HAL
open science

Numerical modelling and High Performance Computing for sediment flows

Jean-Baptiste Keck

To cite this version:

Jean-Baptiste Keck. Numerical modelling and High Performance Computing for sediment flows. Numerical Analysis [math.NA]. Université Grenoble Alpes, 2019. English. NNT : 2019GREAM067 . tel-02433509v2

HAL Id: tel-02433509

https://theses.hal.science/tel-02433509v2

Submitted on 20 Jul 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.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Mathématiques Appliquées

Arrêté ministériel : 25 mai 2016

Présentée par

Jean-Baptiste KECK

Thèse dirigée par **Georges-henri COTTET,** UGA
codirigée par **Iraj MORTAZAVI**, CNAM
et encadrée par **Christophe PICARD**, CNRS

préparée au sein du **Laboratoire Laboratoire Jean Kuntzmann**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Modélisation numérique et calcul haute performance de transport de sédiments

Numerical modelling and High Performance Computing for sediment flows

Thèse soutenue publiquement le **13 décembre 2019**,
devant le jury composé de :

Monsieur GEORGES-HENRI COTTET
PROFESSEUR, UNIVERSITÉ GRENOBLE ALPES,
Directeur de thèse
Monsieur EMMANUEL JEANNOT
DIRECTEUR DE RECHERCHE, INRIA BORDEAUX SUD-OUEST,
Rapporteur
Monsieur FLORIAN DE VUYST
PROFESSEUR, UNIVERSITÉ DE TECHNOLOGIE DE COMPIÈGNE,
Rapporteur
Monsieur IRAJ MORTAZAVI
PROFESSEUR, CNAM - PARIS,
Codirecteur de thèse
Monsieur CHRISTOPHE PICARD
MAÎTRE DE CONFÉRENCES, CNRS, UNIV. GRENOBLE ALPES,
Examinateur
Madame VALÉRIE PERRIER
PROFESSEUR, CNRS, UNIVERSITÉ GRENOBLE ALPES,
Président
Madame DELPHINE DEPEYRAS
DOCTEUR-INGÉNIEUR, SOCIÉTÉ INGELIANCE - MÉRIGNAC,
Examinateur
Monsieur ECKART MEIBURG
PROFESSEUR, UNIV. DE CALIFORNIE - SANTA BARBARA, USA,
Examinateur

UNIVERSITÉ DE GRENOBLE ALPES

MSTII

Mathématiques, Sciences et Technologies de l'Information, Informatique

T H È S E

pour obtenir le titre de

docteur en sciences

de l'Université Grenoble Alpes

Spécialité : Mathématiques appliquées

Présentée et soutenue par

Jean-Baptiste KECK

Numerical modelling and High Performance Computing for sediment flows

Thèse dirigée par Georges-Henri COTTET
codirigée par Iraj MORTAZAVI

préparée au sein du Laboratoire Jean Kuntzmann (LJK)

soutenue le 13 décembre 2019

Composition du Jury :

Président :	Valérie PERRIER	-	CNRS, Université Grenoble Alpes
Rapporteurs :	Emmanuel JEANNOT	-	INRIA, Bordeaux Sud-Ouest
	Florian DE VUYST	-	Université de Technologie de Compiègne
Examinateurs :	Christophe PICARD	-	CNRS, Université Grenoble Alpes
	Delphine DEPEYRAS	-	Ingéliance Technologies, Mérignac
	Eckart MEIBURG	-	University of California, Santa Barbara
Directeurs :	Georges-Henri COTTET	-	Université Grenoble Alpes
	Iraj MORTAZAVI	-	CNAM Paris
Invités :	Chloé MIMEAU	-	CNAM Paris
	Matthieu PUYO	-	Ingéliance Technologies, Mérignac

「お箸は汁物を摑みません。」ミシュラン博士。

« *C'est pas faux !* »

Perceval

Remerciements

Je souhaite remercier en premier lieu mon directeur de thèse, Georges-Henri Cottet, pour m'avoir permis de réaliser cette thèse dans son équipe au sein du laboratoire Jean Kuntzmann. Je lui suis également reconnaissant pour ses qualités scientifiques, sa patience et sa franchise. J'aimerais ensuite remercier mon codirecteur de thèse, Iraj Mortazavi, pour les nombreux séjours au CNAM de Paris, sa sympathie et pour tout son enthousiasme dans ce travail.

Je souhaite également remercier mon encadrant de thèse, Christophe Picard, sans qui ce travail ne serait peut-être pas arrivé à son terme. Je le remercie pour sa grande disponibilité, sa bonne humeur et tout le soutien qu'il m'a apporté durant ces quatre longues années de thèse. J'espère qu'une fois son habilitation à diriger les recherches en poche il pourra continuer à transmettre cette ambiance de travail à la future génération de thésards de l'équipe.

Je remercie mes deux rapporteurs de thèse, à savoir Emmanuel Jeannot et Florian De Vuyst, pour avoir pris le temps de relire ce document conséquent et pour leurs retours très positifs et pertinents. Je tiens également à remercier tous les autres membres du jury : la présidente du jury Valérie Perrier ainsi que tous les autres examinateurs Delphine Depeyras et Eckart Meiburg pour avoir accepté de parcourir ce travail et pour avoir tenté de rejoindre Grenoble malgré une situation quelque peu compliquée.

Je remercie chaleureusement Eckart Meiburg d'avoir fait le déplacement depuis la Californie pour la soutenance et pour les nombreux échanges autour de son modèle de sédimentation. Je remercie également son ancien étudiant de thèse Peter Burns pour les précisions qu'il a pu m'apporter quant à l'aspect numérique du problème. J'espère que ce travail n'est que la première étape d'une longue collaboration entre Grenoble et Santa Barbara.

Je remercie Delphine Depeyras, Matthieu Puyo, Quentin Desbonnets et Florent Braure de l'accueil qu'ils m'ont réservé au sein du groupe Ingéliance sur le site de Mérignac, et de m'avoir permis de découvrir le monde de la simulation numérique en entreprise par le biais du dispositif de doctorat-conseil. Cette collaboration s'est avérée très enrichissante, avec en prime l'obtention conjointe du prix de la simulation numérique de l'usine nouvelle.

J'adresse aussi mes remerciements aux personnes qui ont gravité autour de ce projet de thèse, Chloé Mimeau, Jean-Matthieu Etancelin et Franck Pérignon, pour ces nombreux échanges qui ont grandement contribué à la complétion de travail. Mention spéciale à Chloé pour son soutien et sa positivité sans failles, à Jean-Matthieu pour ses retours et contributions dans la librairie et à Franck pour m'avoir mis le pied à l'étrier (et le café, faute de quoi ce travail ne serait bien évidemment pas arrivé à terme).

Je n'oublie pas bien sur tout le personnel du laboratoire pour m'avoir permis de mener ce travail sans accrocs. Tout d'abord je remercie le personnel informatique, notamment Fred, Patrick et Bruno, pour la résolution rapide de tous ces petits tracas informatiques du quotidien. Je remercie aussi tout le personnel administratif et en particulier Laurence, Gaëlle, Aurore et Cathy de nous simplifier la vie au jour le jour.

Je pense également aux autres membres de l'équipe, notamment à Christophe, amateur de vins et autres spiritueux, au passé texan, qui tente de maintenir tant bien que mal l'équipe à flot; à Brigitte, qui malgré les plaisanteries dont elle est souvent la cible, propose tout de même quelques douceurs pour accompagner le café, voire même, dans son immense bonté, un graphe des relations de publication sur HAL; à Emmanuel, toujours vêtu d'une belle chemise (et qui oublie parfois son polycopié dans l'ascensseur), qui m'a permis de mettre en place la visioconférence en ce vendredi 13 sans transports; à Clément qui a dû s'incliner sur l'épreuve des clous lors de la journée de rentrée du laboratoire et à Ksenia qui a su appliquer chaque jour avec vigueur la méthode scientifique pour finalement devenir développeuse CFD sur Poitiers. Toutes mes pensées vont bien évidemment aux développeurs `Fortran 77` et à leurs familles. Je pense enfin au dernier venu, Simon, un grand artiste qui m'a tout de suite pris sous son aile et qui a accepté de devenir mon coach de vie. Sa fameuse méthode TPLM a sans doute été la clé de la rédaction de ce manuscrit, même si Olivier Minne, les cartes à trous, le `PEP-8`, le reblochon, la chartreuse M.O.F. et le télex y sont aussi sûrement pour quelque chose.

Je remercie ensuite mes collègues de bureau, fussent-ils contemporains de la tour irma ou du batimag. Je pense notamment à Quentin, ce bon vieux Matthias des familles (toujours célibataire), Thomas le web développeur aguerri (qui fera un très bon ingénieur), la maman du LJK (avec ses plats succulents et ses petites gâteries), bouclette l'ours du LJK fan d'échecs et bien évidemment *el famoso* Dr. Meyron qui peut vous faire miroiter tout et n'importe quoi (et qui apprécie les fruits au sirop). Grâce à vous tous, cette période a été riche d'échanges : échanges de quantité de mouvement tout d'abord grâce aux 54 balles habilement éparpillées dans tous les recoins du bureau, échanges musicaux avec la découverte de nombreuses superstars (le chœur universitaire, Damien Jean, Khaled Freak, Salut c'est cool, Philippe Katerine, Vald, Kaaris, Liza M., Al K. et autres grands artistes en 3 lettres), échanges culturels (Kaamelott et le cul de chouette, Coincoin et les z'inhumains, la scène underground), échanges sportifs (le chamboule-tout et le championnat du monde de fléchettes), échanges d'opportunités business avec la création de nombreuses jeunes pousses (dont la licorne frigofit) et échanges scientiques enfin avec la fameuse étude sur l'angle optimal de rebond balistique et l'étude approfondie de la tribologie. Je pense également aux trois générations de stagiaires qui se sont succédé et qui n'ont eu d'autre choix que de baigner dans cette ambiance : les louveteaux de la meute (Nathan et Gregouze), les deux supernanas (Mélanie et Anne) et mes étudiants de compétition (Nicolas et Joël). Tous auront été placés astucieusement sur les trajectoires balistiques habituelles du bureau et, chose surprenante, aucun n'y est resté. Sophie la girafe est, de tou · te · s mes collègues de bureau, la seule qui a survécu à mon passage au LJK. Sa datation au carbone 14 indique qu'elle proviendrait au moins de l'ère irma, où les abeilles butinaient encore en plein air afin d'y produire le fameux miel mural amianté bio tant prisé des connaisseurs.

Je dois également beaucoup à tous mes collègues de proximité, présent ou passé : Émilie (la tigresse), Arnaud (un grand homme), Mr. Bălăşoiu*z* (en trajectoire ballistique), Lionel (petit ange parti trop tôt), David (le hipster), Alexis (les maths marchent), Zhu (Dédé), Jean-Baptiste (qui se tape des barres), Nicolas (le béret noir) et François (le kiné) qui ont su ambiancer avec brio le couloir central du premier étage du batimag. Je remercie également les startuppers pour leur curiosité quant à la présence de peluches poulpiques dans le bureau.

Aussi, j'aimerais remercier tous les autres jeunes du laboratoire avec qui j'ai pu échanger quelques mots, une bière, un café ou encore une session jeu durant toutes ces années : Kévin, Meriem, Charles, Philomène, Abdelkader, Chloé, Victor, Claudia, Olivier, Fairouz, Modibo, Sophie, Musaab, Flora, Arthur, Maria Belen, Matthieu, Mariliza, Hubert, Alexandra, Anatole, Yassine, Anna, Nils et tous les autres que j'ai malheuresement pu oublier par mégarde. J'ai pu trouver en dehors du bureau certaines personnes qui sont devenues de vrais amis. J'ai une pensée particulière pour Alexandra, avec qui j'ai partagé de nombreuses heures de discussion, de confection de Weihnachtsgebäck, restaurants et autres thés aux perles; à David qui finira bien par soutenir un jour et à Émilie, la caïd de Mulhouse, subitement devenu maman d'un beau bébé panda de 2m de haut pour 12kg; et à beaucoup d'autres qui se reconnaîtront.

Merci à tous ceux qui ont plébiscité les nombreux montages de maître Gimps, à ceux qui ont encensé les diverses citations de Jibouze, et enfin à ceux qui ont dû supporter les divers accents et jeux de mots de Jean Bof. J'ai en effet une pensée émue pour la boîte aux lettres de la commission parité qui risque de souffrir d'une longue période de chômage technique après mon départ. Merci à Poulpy et à Poulpine pour avoir apporté une aura de paix dans le bureau, tout en faisant office de vulgaires projectiles. J'espère que votre union marquera une nouvelle ère de prospérité dans le bureau 124, avec notamment l'arrivée de nouveaux membres de votre espèce. Merci à Aix pour avoir tenu si longtemps. Tu étais notre cible favorite jusqu'à ce que tu tombes sur la mauvaise personne au mauvais moment. Merci, encore, à Dimitri, pour cette sombre histoire de palette. Enfin, mille mercis à Boris pour ses nombreuses offrandes de pinces à linges et autres objets divers qui tapissent dorénavant le bureau.

J'aimerais ensuite remercier mes amis de l'Ensimag : Carl, Romain, Lionel, Jocelyn, Alexandre, Guillaume, Gauthier, Joël, Antoine et Thomas avec qui j'ai pu passer de très bons moments durant cette thèse, que ce soit durant nos escapades en Espagne, dans les montagnes, sur des îles, dans le canton de Vaud ou encore dans le temple du fromage ou autres accélérateurs de particules. Il faut aussi citer nos nombreuses escapades vidéoludiques sur combat de baton, attaque de pigeons, tours délicates ou encore cheval de poulet ultime. Ces sessions furent complémentées avec un apport conséquent en contre-attaque : source, zone frontalière 3, tuerie de sol 2 et Mordhau (pour les amateurs). Je souhaite également remercier mes amis restés en Alsace, en particulier Cédric, Florian et Martin. Je pense aussi à ceux se sont installés un peu plus loin et que je n'ai malheureusement pas pu beaucoup côtoyer durant cette période.

En parlant des personnes qui sont restées en Alsace, je pense bien évidemment aussi à ma famille qui m'a toujours soutenu, notamment mon frère François et ma soeur Madeline. Je pense également à mes parents et mes grands-parents qui sont heureux d'accueillir un docteur dans la famille. Merci également à tous ceux qui ont participé à la préparation, à la mise en place et au bon déroulement de mon pot de soutenance (et pour les cadeaux).

Enfin je remercie la personne sans qui cela n'aurait tout simplement pas été possible. Merci à toi Clarisse, pour tous ces instants de partage au quotidien.

Dr Michelin

Modélisation numérique et calcul haute performance de transport de sédiments

Résumé — La dynamique des écoulements sédimentaires est un sujet qui concerne de nombreuses applications en géophysiques, qui vont des questions d'ensablement des estuaires à la compréhension des bassins sédimentaires. Le sujet de cette thèse porte sur la modélisation numérique à haute résolution de ces écoulements et l'implémentation des algorithmes associés sur accélérateurs. Les écoulements sédimentaires font intervenir plusieurs phases qui interagissent, donnant lieu à plusieurs types d'instabilités comme les instabilités de Rayleigh-Taylor et de double diffusivité. Les difficultés pour la simulation numérique de ces écoulements tiennent à la complexité des interactions fluides/sédiments qui font intervenir des échelles physiques différentes. En effet, ces interactions sont difficiles à traiter du fait de la grande variabilité des paramètres de diffusion dans les deux phases et les méthodes classiques présentent certaines limites pour traiter les cas où le rapport des diffusivités, donné par le nombre de Schmidt, est trop élevé. Cette thèse étend les récents résultats obtenus sur la résolution directe de la dynamique du transport d'un scalaire passif à haut Schmidt sur architecture hybride CPU-GPU et valide cette approche sur les instabilités qui interviennent dans des écoulements sédimentaires. Ce travail revisite tout d'abord les méthodes numériques adaptées aux écoulements à haut Schmidt afin de pouvoir appliquer des stratégies d'implémentations efficaces sur accélérateurs et propose une implémentation de référence open source nommée HySoP. L'implémentation proposée permet, entre autres, de simuler des écoulements régis par les équations de Navier-Stokes incompressibles entièrement sur accélérateur ou coprocesseur grâce au standard OpenCL et tend vers des performances optimales indépendamment du matériel utilisé. La méthode numérique et son implémentation sont tout d'abord validées sur plusieurs cas tests classiques avant d'être appliquées à la dynamique des écoulements sédimentaires qui font intervenir un couplage bidirectionnel entre les scalaires transportés et les équations de Navier-Stokes. Nous montrons que l'utilisation conjointe de méthodes numériques adaptées et de leur implémentation sur accélérateur permet de décrire précisément, à coût très raisonnable, le transport sédimentaire pour des nombres de Schmidt difficilement accessibles par d'autres méthodes.

Mots clés : Transport sédimentaire, couplage bidirectionnel, méthodes particulaires, double diffusivité, HPC, GPU, HySoP

Laboratoire Jean Kuntzmann
Bâtiment IMAG - Université Grenoble Alpes
700 Avenue Centrale
Campus de Saint Martin d'Hères
38401 Domaine Universitaire de Saint-Martin-d'Hères
France

Numerical modelling and High Performance Computing for sediment flows

Abstract — The dynamic of sediment flows is a subject that covers many applications in geophysics, ranging from estuary silting issues to the comprehension of sedimentary basins. This PhD thesis deals with high resolution numerical modeling of sediment flows and implementation of the corresponding algorithms on hybrid calculators. Sedimentary flows involve multiple interacting phases, giving rise to several types of instabilities such as Rayleigh-Taylor instabilities and double diffusivity. The difficulties for the numerical simulation of these flows arise from the complex fluid/sediment interactions involving different physical scales. Indeed, these interactions are difficult to treat because of the great variability of the diffusion parameters in the two phases. When the ratio of the diffusivities, given by the Schmidt number, is too high, conventional methods show some limitations. This thesis extends the recent results obtained on the direct resolution of the transport of a passive scalar at high Schmidt number on hybrid CPU-GPU architectures and validates this approach on instabilities that occur in sediment flows. This work first reviews the numerical methods which are adapted to high Schmidt flows in order to apply effective accelerator implementation strategies and proposes an open source reference implementation named HySoP. The proposed implementation makes it possible, among other things, to simulate flows governed by the incompressible Navier-Stokes equations entirely on accelerator or coprocessor thanks to the OpenCL standard and tends towards optimal performances independently of the hardware. The numerical method and its implementation are first validated on several classical test cases and then applied to the dynamics of sediment flows which involve a two-way coupling between the transported scalars and the Navier-Stokes equations. We show that the joint use of adapted numerical methods and their implementation on accelerator makes it possible to describe accurately, at a very reasonable cost, sediment transport for Schmidt numbers difficult to reach with other methods.

Keywords: Sediment transport, dual-way coupling, particle methods, double diffusivity, HPC, GPU, HySoP

Laboratoire Jean Kuntzmann
Bâtiment IMAG - Université Grenoble Alpes
700 Avenue Centrale
Campus de Saint Martin d'Hères
38401 Domaine Universitaire de Saint-Martin-d'Hères
France

Contents

Introduction **1**

1 High performance computing for sediment flows **5**

1.1 Computational fluid dynamics 7

1.1.1 Derivation of the Navier-Stokes-Fourier equations 7

1.1.2 Vorticity formulation 12

1.1.3 Newtonian fluids 12

1.1.4 Compressible Navier-Stokes equations 13

1.1.5 Incompressible flows 13

1.1.6 The Boussinesq approximation 14

1.1.7 Dimensionless numbers 16

1.1.8 Numerical resolution 18

1.1.9 Vortex methods 19

1.1.10 Scalar transport in high Schmidt number flows 20

1.2 Sediment flows 21

1.2.1 Sediment transport mechanisms 21

1.2.2 Sediments-laden flows classification 24

1.2.3 Numerical models 26

1.3 Density-driven convection 29

1.3.1 Gravity currents 29

1.3.2 Double diffusive convection 31

1.4 High Performance Computing 41

1.4.1 Hierarchy of a computing machine 41

1.4.2 Evaluation of performance 42
1.4.3 Accelerators and coprocessors 44
1.4.4 Parallel programming models 47
1.4.5 The `OpenCL` standard 48
1.4.6 Target `OpenCL` devices 49
1.4.7 The roofline model 53
1.4.8 High performance fluid solvers 55

2 Numerical method **61**

2.1 Outline of the method 63
2.1.1 Operator splitting 63
2.1.2 Temporal resolution 65
2.1.3 Directional splitting 67
2.2 Spatial discretization 70
2.2.1 Cartesian grids 70
2.2.2 Field discretization, interpolation and restriction 72
2.3 Hybrid particle-grid method 73
2.3.1 Semi-lagrangian methods 74
2.3.2 Explicit Runge-Kutta methods 80
2.3.3 Construction of the remeshing formulas 82
2.4 Finite difference methods 85
2.4.1 Spatial derivatives 86
2.4.2 Finite difference schemes 88
2.4.3 Performance considerations 90
2.5 Directional splitting 91
2.5.1 Advection 92
2.5.2 Diffusion 93

2.5.3 Stretching 94
2.5.4 External forces and immersed boundaries 97
2.6 Fourier spectral methods 98
2.6.1 Discrete Fourier transform 98
2.6.2 Spectral diffusion 105
2.6.3 Vorticity correction and computation of velocity 107
2.6.4 Real to real transforms and homogeneous boundary conditions 110
2.7 Extension to general non-periodic boundary conditions 118
2.7.1 General spectral methods 118
2.7.2 Chebyshev collocation method 120
2.7.3 Chebyshev-Tau method 123
2.7.4 Chebyshev-Galerkin method 125
2.7.5 Comparison of the different methods 128

3 Implementation and High Performance Computing 131

3.1 The HySoP library 133
3.1.1 Origin and current state of the library 133
3.1.2 Design of the library 137
3.1.3 `HySoP` from a user point of view 143
3.1.4 Graph of operators and task parallelism 146
3.1.5 Integration of numerical methods 148
3.2 Implementation on hardware accelerators 149
3.2.1 Arrays, buffers and memory handling 150
3.2.2 `OpenCL` code generation framework 152
3.2.3 Permutations and automatic tuning of kernel runtime parameters . . . 159
3.2.4 Directional advection operator 164
3.2.5 Finite differences operators 172

3.2.6 Spectral transforms 175
3.2.7 Navier-Stokes runtime distribution 178
3.2.8 Validation of the solver 180
3.3 Distributed computing 186
3.3.1 Domain decomposition 186
3.3.2 Interprocess communications 187
3.3.3 Field discretization 192
3.3.4 Spectral solvers 192

4 Sediment-laden fresh water above salt water 195

4.1 Statement of the problem 197
4.1.1 Governing equations 198
4.1.2 Boussinesq approximation 199
4.1.3 Boundary conditions 200
4.1.4 Initial conditions 200
4.1.5 Interfacial instabilities 201
4.2 Setup and validation of the solver 202
4.2.1 Outline of the method 202
4.2.2 Timestep criterias 203
4.2.3 Numerical setup 204
4.2.4 Flow characteristics 204
4.2.5 Comparison with two-dimensional reference solution 206
4.2.6 Comparison with three-dimensional reference solution 210
4.3 High performance computing for sediment flows 213
4.3.1 Ensemble averaged two-dimensional simulations 213
4.3.2 Three-dimensional simulations 216
4.3.3 Achieving higher Schmidt numbers 218

Conclusion and perspectives **221**

Acknowledgments **224**

A Mathematical notations **225**

A.1 Function spaces 225

A.2 Functions and symbols 226

A.3 Vector operations 226

A.4 Vector calculus identities 227

B HySoP dependencies and tools **229**

B.1 FFT libraries 229

B.2 Python modules 230

B.3 Build tools 231

B.4 Benchmarking and debugging tools 232

C `Oclgrind` plugin and kernel statistics **233**

C.1 Requirements, build and usage 233

C.2 Output sample and obtained kernel statistics 234

Bibliography **266**

List of Figures

1 Le Havre, Seine estuary, France - February 23rd, 2019 1

1.1 Three-dimensional turbulent scalar transport at high Schmidt number 20

1.2 Illustration of the different sediment transport mechanisms 22

1.3 Plot of the Hjulström curve in logarithmic scale 24

1.4 Map of sediment laden flow regimes 25

1.5 Example of 2D gravity current 31

1.6 Effect of Rayleigh numbers on the evolution of double-diffusive salt fingers . . 32

1.7 Potential temperature versus salinity plot 34

1.8 Initial configuration for a thermohaline convection 34

1.9 Settling velocity with respect to grain grain size 38

1.10 Semi-empirical model [Segre et al. 2001] for ϕ between 0.1 and 10% 39

1.11 Estimation of the diffusivity ratio between salinity and particles 40

1.12 Exponential growth of supercomputing power - `TOP500` list 42

1.13 Treemap representing `TOP500` coprocessor share. 44

1.14 Coprocessor share in the supercomputers containing accelerators 45

1.15 Evolution of the cost of `1 TFLOPS` of double-precision arithmetic 46

1.16 Evolution of energy consumption in terms of `GFLOPS/W` 46

1.17 Achieved memory bandwidth vs. memory transaction size 51

1.18 Achieved memory bandwidth versus float vector type for all devices 52

1.19 Achieved memory bandwidth versus transaction granularity 53

1.20 Roofline of mixbench achieved single-precision performance 54

1.21 Roofline of mixbench achieved double-precision performance 54

2.1 Illustration of the discretization of rectangle-shaped domain 70

2.2 Example of domain decomposition in 24 subdomains 71

2.3 Local interpolation by using an interpolating kernel 72

2.4 Illustration of the two classes of semi-lagrangian methods 75

2.5 Advection with first and second order explicit Runge-Kutta schemes 76

2.6 Example of multiscale advection by using bilinear interpolation 77

2.7 Example of two-dimensional remeshing by tensor product 78

2.8 One-dimensional remeshing kernel 79

2.9 Illustration of some semi-lagrangian remeshing kernels 84

2.10 Stencils approximating the 2D Laplacian 87

2.11 Explicit 2D finite difference scheme used to solve the heat equation 89

2.12 Directional splitting of the advection-remeshing procedure 92

2.13 Spectral interpolation of 1-periodic functions of varying regularity 102

2.14 Spectral convergence of 1-periodic functions of varying regularity 103

2.15 Spectral interpolation by using the eight real-to-real transforms 111

2.16 Zoom on the symmetries exhibited by the different transforms 113

2.17 Spectral Chebyshev interpolation of functions of varying regularity 121

2.18 Convergence of the GLC Chebyshev interpolation method 122

2.19 Matrix layouts associated to the different methods 128

2.20 Convergence of the different methods for a one-dimensional homogeneous Dirichlet Poisson problem 129

3.1 `HySoP` logo representing an hyssop (*Hyssopus officinali*) plant 134

3.2 Evolution of the programming languages used in the `HySoP` library. 135

3.3 Different computing backends present in the `HySoP` library. 136

3.4 Simplified `UML` diagram representing the `Domain` class hierarchy. 137

3.5 Simplified `UML` diagram of the `ContinuousField` class hierarchy. 138

3.6 Simplified `UML` diagram of the `DiscreteField` class hierarchy. 142

3.7 Example of directed graph of operators obtained after step one 147

3.8 Example of DAG of operators obtained after step four 147

3.9 Three dimensional array benchmark on `CPU` 157

3.10 Three dimensional array benchmark on `GPU` 158

3.11 Interactions between the kernel code generator and kernel autotuner 159

3.12 Kernel statistics obtained for `2D` transposition of arrays of size 16384^2 161

3.13 Overview of the process to gather kernel execution statistics 166

3.14 Roofline of 2D and 3D, single- and double-precision advection 167

3.15 Roofline of 2D and 3D, single- and double-precision remeshing 168

3.16 Distribution of runtime for n-dimensional single-precision transport 170

3.17 Distribution of runtime for single-precision transport on the `CPU` 171

3.18 Overview of the directional symbolic code generation process 172

3.19 Rooflines of single- and double-precision directional stretching 174

3.20 `FFT` benchmark, 2D complex-to-complex discrete Fourier transforms 177

3.21 Distribution of runtime for single-precision Navier-Stokes on `CPU` 178

3.22 Distribution of runtime for single-precision Navier-Stoke on `GPU` 179

3.23 Mean runtime per timestep for the resolution of the Taylor-Green vortex problem. 181

3.24 Evolution of enstrophy with respect to time 182

3.25 Comparison of obtained enstrophy to reference enstrophies 182

3.26 Slice of vorticity at x=0 and comparison of isocontours 183

3.27 Evolution of enstrophy with respect to time - variable timestep 184

3.28 Volume rendering of the norm of the vorticity $\|\boldsymbol{\omega}\| \geq 10$ 185

3.29 Example of uniform grid distribution 188

3.30 Extension of the subdomain with ghosts nodes 188

3.31 Local grid data layout 191

3.32 Example of slab decomposition of a three-dimensional domain 193

3.33 Example of pencil decomposition of a three-dimensional domain 193

4.1 River delta of Irrawaddy, Myanmar (European Space Agency) 196

4.2 Two-dimensional problem configuration 197

4.3 Initial vertical sediment profile for $z \in [-200, 200]$ and $l_0 = 1.5$. 200

4.4 Sketch of the different kind of instabilities the flow can develop 201

4.5 Snapshots of horizontally averaged sediment and salt profiles 206

4.6 Evolution of horizontally averaged interface locations and thicknesses 207

4.7 Comparison of 2D sediment slices with reference data 208

4.8 Comparison of 2D salt slices with reference data 209

4.9 Evolution of the ratio of the nose region height to the salinity interface thickness 210

4.10 Comparison of 3D sediment slices with reference data 211

4.11 Comparison of 3D sediment isocontours with reference data 212

4.12 Ensemble averaged quantities at varying Schmidt number 214

4.13 3D sediment isocontours at moderate Schmidt number 215

4.14 Prediction of the nose thickness ratio H/l_s versus obtained results 217

4.15 Example of the physical scales obtained with $Sc = 7$ and $\tau = 25$ 218

4.16 3D sediment isocontours for varying Schmidt number 219

List of Tables

1.1 Characteristics and compute capabilities of the considered devices 50

1.2 Peak performance metrics obtained for the four device configurations 50

1.3 Designed arithmetic intensity for usual floating point operations 50

2.1 Comparative of some high order semi-lagrangian remeshing kernels 83

2.2 Finite differences coefficients for first and second derivatives 86

2.3 Chebyshev-Galerkin coefficients (2.131) for usual boundary conditions 126

3.1 Comparison of some features between `hysop-origin` and `hysop-current`. . . 136

3.2 Kernel autotuning results for `2D` and `3D` `OpenCL` permutation kernels 162

3.3 Performance of transposition on a quad socket NUMA architecture 163

3.4 Approximative arithmetic intensity of the advection kernels 165

4.1 Ensemble averaged statistics for different Schmidt numbers 213

4.2 Dimensionless group values and grid properties for the parametric study . . . 216

4.3 Obtained nose region ratio for three-dimensional test cases (E) 217

C.1 2D and 3D single-precision directional advection kernel statistics 235

C.2 2D and 3D double-precision directional advection kernel statistics 236

C.3 2D and 3D single-precision directional remesh kernel statistics 237

C.4 2D and 3D double-precision directional remesh kernel statistics 238

C.5 3D single- and double-precision directional diffusion kernel statistics 239

C.6 3D single- and double-precision directional stretching kernel statistics 240

List of Algorithms

1 Simulation of the one-way coupling of a scalar with incompressible Navier-Stokes 66

2 First order Strang splitting applied to operator $L = L_1 + \cdots + L_p$ 68

3 Second order Strang splitting applied to operator $L = L_1 + \cdots + L_p$ 68

4 Algorithm to solve the turbulent scalar transport problem by using viscous splitting and first order Strang splitting 69

5 Remeshing procedure using a remeshing kernel W with support $\boldsymbol{s}$ 79

6 Stencil convolution algorithm 87

7 Spectral resolution of a periodic Poisson problem 107

8 Algorithm to compute divergence-free vorticity and velocity on a 2D fully periodic domain discretized on $\boldsymbol{\mathcal{N}} = (\mathcal{N}_y, \mathcal{N}_x)$ points where $\mathcal{N}_x$ and $\mathcal{N}_y$ are odd. . 109

9 Algorithm to compute divergence-free vorticity and velocity on a 3D fully periodic domain discretized on $\boldsymbol{\mathcal{N}} = (\mathcal{N}_z, \mathcal{N}_y, \mathcal{N}_x)$ points with $\mathcal{N}_* \in 2\mathbb{N} + 1$. 109

10 Spectral diffusion with mixed periodic and homogeneous Dirichlet/Neumann boundaries 116

11 Algorithm to compute divergence-free vorticity and velocity on a 2D domain with a mix of periodic and homogeneous boundary conditions. 117

Acronyms and abbreviations

HPC	High Performance Computing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
MIC	Many Integrated Core
GPGPU	General-Purpose computing on Graphics Processing Units
JIT	Just-In-Time
OpenCL	Open Computing Language
CUDA	Compute Unified Device Architecture
MPI	Message Passing Interface
CFD	Computational Fluid Dynamics
DNS	Direct Numerical Simulation
LES	Large Eddy Simulation
RANS	Reynolds-Averaged Navier-Stokes equations
EE	Eulerian-Eulerian
LE	Lagrangian-Eulerian
DAG	Directed Acyclic Graph
LOC	Lines of Code
BC	Boundary Condition

Introduction

Continental erosion is controlled by climate and tectonics and can be enhanced by human activities such as large-scale agriculture, deforestation, and sand extraction. Erosion produces sediments that are mostly evacuated by rivers. Downstream, the supply of sediment carries organic matter and nutrients to the ocean, which are fundamental for marine ecosystems and for oil and gas reservoirs [Mouyen et al. 2018]. Estimation of the global sediment discharge in the ocean remains unknown and its measurement still represents a difficult challenge in earth sciences. While it has been estimated that the largest river sediment discharges probably reach one billion of metric tons per year [Milliman et al. 2013], the processes by which this sediment settles out from buoyant river plumes are still poorly understood. A better understanding of the dominant sediment settling mechanisms will help to determine the location of sediment deposits on the sea floor. This represents important information for predicting the location of certain hydrocarbon reservoirs [Meiburg et al. 2010].

During the seventies, a simple laboratory experiment showed that when a layer of fluid, made denser by the presence of suspended material, was put above colder, distilled water, a fingering phenomenon similar to thermohaline convection occurred, result of double-diffusive instabilities [Houk et al. 1973]. In the nature, this event has been observed when a muddy river enters a stratified lake or ocean [Schmitt et al. 1978], as shown on figure 1. The density stratification of lakes and oceans can be due to thermal or compositional density gradients such as salinity gradients.

Figure 1 – Le Havre, Seine estuary, France - February 23rd, 2019. Picture adapted from Sentinel Hub. It is licensed under CC BY 2.0 and contains modified Sentinel data.

The density of the seawater is greater than the density of fresh water. When a river plume loaded with fine sediments enters the sea, the river often continues to flow above the seawater (hypopycnal flow) and sediment fingering can occur [Mulder et al. 2003]. Hypopycnal flows supposes that the sediment concentration remains dilute (less than $40kg/m^3$), else the density of the river is more than the density of sea, in which case the river continues to flow near the bottom of the sea floor (hyperpycnal flow). For dilute suspensions of particles, Stokes law predicts the settling velocity of small particles in fluid [Ferguson et al. 2004].

In the passed years, most oceanographic sedimentation models relied on Stokes settling as the main mechanism of sediment transfer out of river plumes. Aware that double-diffusive instabilities could greatly increase the effective settling velocity of fine sediments, [Burns et al. 2012] performed a numerical investigation to the explore the effect of double-diffusive mechanisms on sediment settling. Through extensive three-dimensional simulations, it was determined that when sediment-laden water flows over clearer and denser water, both double-diffusive and Rayleigh-Taylor instabilities may arise [Burns et al. 2015]. In their analysis both of those convective instabilities led to downward sediment fluxes that could be orders of magnitude above the ones that would normally be obtained with gravitational settling alone. They however faced a major problem when trying to apply their model to fine sediments such as clay and fine silts. The very low diffusivity of small particles [Segre et al. 2001] with respect to the momentum diffusivity (the kinematic viscosity of the carrier fluid), prevented them to perform the corresponding numerical simulations. This fluid-to-particle diffusivity ratio is often reduced to a dimensionless Schmidt number, and high Schmidt number flows are known to be very computationally demanding [Lagaert et al. 2012]. This work aims to achieve high Schmidt number sediment-laden flow simulations by using adapted numerical methods in a high performance computing context.

In the field of fluid mechanics, the evolution of numerical simulations greatly contributes to improve the understanding of such complex physical phenomena. A numerical simulation makes it possible to obtain a solution to a mathematical model that is generally not possible to solve analytically. These models consist of equations modeling a phenomenon to study. One of the objective of numerical simulations is to allow the validation of the physical and mathematical models used for the study of a phenomenon, compared to the theory and to possible experimental data. Since the thirties, the use of calculators has continued to grow with the rapid development of computing machines. In recent years, advances in computing enabled a large enhancement of numerical simulations related to computational fluid dynamics (CFD). These simulations allow to predict the physical behavior of complex flow configurations, possibly coupled with other physical models. The large increase in computing power over the last decades allows the computation of numerical simulations of increasing accuracy and complexity while maintaining comparable computation times. Numerical simulations are based on one or more numerical methods and a computer code that implements these methods on computing machines. The choice of the numerical methods depends essentially the problem to be solved. The numerical efficiency of a code is measured, among other things, by the calculation time required to obtain the solution according to the desired precision. Thus, the implementation of a method requires special attention in order to make the best use of available computing resources. The development of an efficient code in a context of high

performance computing is a real challenge on nowadays massively parallel machines. Therefore, numerical methods must not only be adapted to the characteristics of the problems to be solved, but also to the architectures of the machines on which they will be implemented. In particular, the eventual presence of multiple compute nodes and/or accelerators such as `GPUs` or other coprocessors must be taken into account from the resolution algorithm to the development of the numerical method.

In this work we limit ourselves to the study of non-cohesive particle-laden hypopycnal flows above salt water. The physics of this type of problem is characterized by the presence of several phenomena at different physical scales that is due to the large difference in the diffusivities of the transported components [Batchelor 1959]. In this thesis, we consider the problem of active scalar transport at high Schmidt number as an application framework. The transported scalars, representing particles concentration and salinity, will be coupled to the carrier fluid bidirectionally. The approach considered here is that of a resolution by an hybrid particle-grid method [Lagaert et al. 2014]. Remeshed particle methods makes it possible to naturally solve conservation equations without imposing a Courant-Friedrichs-Lewy stability constraint, but a less restrictive stability condition allowing the use of larger time steps [Cottet 2016]. The use of this method leads, through the presence of an underlying grid, to algorithms and regular data structures particularly adapted to heavily vectorized hardware such as `GPUs`[Rossinelli et al. 2008][Rossinelli et al. 2010][Etancelin et al. 2014].

Numerically speaking, the starting point of this work is the `HySoP` library, a `Python`-based solver based on hybrid `MPI-OpenCL` programming that targets heterogeneous compute platforms, originally developed in the context of passive scalar transport in high Schmidt number flows [Etancelin 2014]. In particular, we strive to follow the original library mantra, that is the development of non-architecture-specific, performance-portable and easily reusable numerical code. This helps to support the rapid evolution of hardware, favours code reusability and problem expressiveness. A particular attention will be paid to code generation techniques and runtime optimization for `GPU`-enabled numerical routines. By pursuing the `OpenCL` porting efforts of the original developer, we will be able to solve custom user-specified numerical problems such as incompressible Navier-Stokes fully on accelerator. The resulting solver will be generic and modular enough to be able to run high performance `GPU` accelerated numerical simulation of particle-laden sediment flows.

The sequence of the four chapters of this manuscript is the following. In a first chapter, the mathematical models required for the modeling of sediment flows are introduced. The second chapter is dedicated to the different numerical methods required to simulate high Schmidt number flows. The existing numerical methods are adapted to fit `GPU`-computing by using operator splitting techniques. Chapter three is dedicated to the numerical code and the high performance implementation of the method, giving technical details on the development of performance-portable multi-architecture numerical routines. In particular, we will see how code generation techniques simplify the use of accelerators from a user point of view. The fourth chapter is dedicated to the simulation of particle-laden clear water above salt-water with the resulting solver. Finally, we will conclude this manuscript giving rise to a critique of the work carried out as well as a statement of the perspectives revealed by this study.

Chapter 1

High performance computing for sediment flows

Contents

1.1 Computational fluid dynamics . . . 7

1.1.1 Derivation of the Navier-Stokes-Fourier equations . . . 7

1.1.2 Vorticity formulation . . . 12

1.1.3 Newtonian fluids . . . 12

1.1.4 Compressible Navier-Stokes equations . . . 13

1.1.5 Incompressible flows . . . 13

1.1.6 The Boussinesq approximation . . . 14

1.1.7 Dimensionless numbers . . . 16

1.1.8 Numerical resolution . . . 18

1.1.9 Vortex methods . . . 19

1.1.10 Scalar transport in high Schmidt number flows . . . 20

1.2 Sediment flows . . . 21

1.2.1 Sediment transport mechanisms . . . 21

1.2.2 Sediments-laden flows classification . . . 24

1.2.3 Numerical models . . . 26

1.3 Density-driven convection . . . 29

1.3.1 Gravity currents . . . 29

1.3.2 Double diffusive convection . . . 31

1.4 High Performance Computing . . . 41

1.4.1 Hierarchy of a computing machine . . . 41

1.4.2 Evaluation of performance . . . 42

1.4.3 Accelerators and coprocessors . . . 44

1.4.4 Parallel programming models . . . 47

1.4.5 The `OpenCL` standard . . . 48

1.4.6 Target `OpenCL` devices . . . 49

1.4.7 The roofline model . . . 53

1.4.8 High performance fluid solvers . . . 55

Introduction

The context of this work is in applied mathematics, at the intersection of a physical problem, numerical methods and a high performance numerical code. From observations of natural or artificial sedimentation phenomena, empirical physical models are elaborated and put into equations in the form of mathematical models. For numerical simulations, mathematical models are discretized according to one or more numerical methods leading to the expression of algorithms that are implemented in numerical codes. A given model generally depends on several parameters, and can be mapped to a given physical situation by specifying these parameters as well as the initial and boundary conditions of the model. The execution of these codes on computers makes it possible to validate both the numerical methods and the models with respect to the physical phenomena.

Once a mathematical model and associated numerical method are considered valid for a certain range of simulation parameters, the numerical code can be used to explore parameter configurations that are not easily observable or measurable. As it will be the case for our sediment-related problem, the required computational power can be largely dependent on those parameters. When a given computational problem requires more than trillion operations per second (10^{12} `OP/s`) in order to finish in acceptable time, it becomes a necessity to distribute the computations on multiple compute nodes containing multiple `CPU` ressources. When those nodes also contain accelerators, such as `GPUs`, it quickly becomes a necessity to adapt the numerical codes to harness their additional compute power. In order to support a wide diversity of hardware efficiently, associated technologies must be taken into account from the design stage of the numerical methods, especially in the perspective of the exascale.

This chapter is dedicated to the presentation of the mathematical background of fluid mechanics in the context of sediment flows and high performance computing. The first section is dedicated to the construction of the model that describes the motion of an incompressible viscous fluid. Some classical resolution methods will be given and a particular attention will be paid to high Schmidt number flows. The second part present some physical models associated to sedimentary processes and lead to the physical model of interest in this work. The last part is focused on high performance computing and introduces distributed computing and coprocessors.

1.1 Computational fluid dynamics

The goal of computational fluid mechanics is to analyze and understand behaviors of fluids in motion, possibly in the presence of obstacles or structures with which they interact. The major part of fluid mechanics problems arise from engineering problems encountered in the aviation industry and more generally in the fields of transport, energy, civil engineering and environmental sciences. As an example, CFD[1] can be used in sedimentology to study sediment dynamics in an estuary system.

Mathematical modeling of these problems leads to systems of equations too complex to be solved formally. For most problems, the existence of solutions to these systems is still an open problem. Numerical modeling of these problems makes it possible to approach a solution by the use of numerical methods. The Navier-Stokes-Fourier equations provide a general description of the behavior of a viscous fluid based on physical principles of conservation of mass, momentum, and energy.

1.1.1 Derivation of the Navier-Stokes-Fourier equations

The Navier-Stokes equations, named after Claude-Louis Navier and George Gabriel Stokes, are a set of nonlinear partial differential equations that describe the motion of a viscous fluid. This set of equations is a generalization of the equation devised by Leonhard Euler in the 18th century to describe the flow of an incompressible and frictionless fluid [Euler 1757]. In 1821, the French engineer Claude-Louis Navier introduced friction for the more complex problem of viscous fluids [Navier 1821]. Throughout the middle of the 19th century, British physicist and mathematician Sir George Gabriel Stokes improved on this work, though complete solutions were obtained only for the case of simple two-dimensional flows [Stokes 1851][Stokes 1880].

Way later, in 1934, Jean Leray proved the existence of weak solutions to the Navier-Stokes equations, satisfying the equations in mean value, not pointwise [Leray 1934]. One had to wait until the 1960s to obtain results about existence of regular solutions to the Navier-Stokes equations in two dimensions thanks to the work of [Ladyzhenskaya 1969]. To this day, the complex vortices and turbulence that occur in three-dimensional fluid flows as velocities increase have proven intractable to any but approximate numerical analysis methods. The existence and uniqueness of classical solutions of the three-dimensional Navier-Stokes equations is still an open mathematical problem and thus constitues an active research area in mathematics.

The balance of equations arise from applying Isaac Newton's second law to fluid motion, together with the assumption that the stress in the fluid is the sum of a diffusing viscous term proportional to the gradient of velocity and a pressure term. Supplementing the Navier-Stokes equations with the conservation of energy leads to the Navier-Stokes-Fourier equations. This section is dedicated to give the idea behind those famous centuries old equations.

[1] Computational Fluid Dynamics

Integral form of the Navier-Stokes equations

Mass conservation principle: There is no creation, nor annihilation of mass. Put in other words it means that the rate of mass accumulation within a control volume Ω is equal to the rate of mass flow that goes into the control volume minus the rate of mass flow that leaves the control volume:

$$\frac{\mathrm{d}}{\mathrm{dt}}\left[\int_{\Omega} \rho \ \mathrm{dv}\right] + \oint_{\partial\Omega} \rho \boldsymbol{u} \cdot \boldsymbol{n} \ \mathrm{ds} = 0 \tag{1.1}$$

where ρ is the density of the fluid, $\boldsymbol{u}$ the velocity of the fluid, Ω the fluid control volume, $\partial\Omega$ the boundaries of the control volume and $\boldsymbol{n}$ the normal to the boundary at integration point pointing outwards Ω.

Momentum conservation principle: This equation is given by Isaac Newton's second law applied on the control volume. As Ω is an open system, we need to take into account the momentum flux due to the particles entering and leaving the control volume:

$$\frac{\mathrm{d}}{\mathrm{dt}}\left[\int_{\Omega} \rho \boldsymbol{u} \ \mathrm{dv}\right] + \oint_{\partial\Omega} (\rho \boldsymbol{u} \otimes \boldsymbol{u})\boldsymbol{n} \ \mathrm{ds} = \int_{\Omega} \boldsymbol{F} \ \mathrm{dv} \tag{1.2}$$

where $\boldsymbol{F}$ represents the forces acting on the fluid in Ω and $\boldsymbol{u} \otimes \boldsymbol{u} = \boldsymbol{u}\boldsymbol{u}^T$ is the outer product of the velocity vector with itself. This equation means that the accumulation of momentum within the control volume is due to the rate of momentum flow going into the control volume minus the rate of momentum flow leaving the control volume supplemented by forces acting the fluid.

The forces applied on the fluid can be expressed as the sum of two forces: $\boldsymbol{F} = \nabla \cdot \overline{\overline{\sigma}} + \boldsymbol{F_{ext}}$ where the divergence of the tensor is defined as the vector of the divergence of all its lines. Here $\boldsymbol{F_{ext}}$ represents all the external forces acting on the fluid and $\overline{\overline{\sigma}}$ is the Cauchy stress tensor that completely define the state of stress at any point inside the control volume in a deformed state. The Cauchy stress tensor can further be decomposed into $\overline{\overline{\sigma}} = \overline{\overline{\tau}} - PI_d$ where $\overline{\overline{\tau}}$ is the viscous stress tensor, I_d is the identity tensor in dimension d and $P = (1/d)\ Tr(\overline{\overline{\tau}})$ the hydrostatic pressure field. The viscous stress tensor accounts for the stress that can be attributed to the strain rate at a given point (the rate at which it is deforming) whereas the pressure field accounts for the local mean compression due to normal stresses.

The decomposition of $\boldsymbol{F}$ as volumetric forces and surface forces becomes straightforward when using the divergence theorem (A.2c):

$$\int_{\Omega} \boldsymbol{F} \ \mathrm{dv} = \int_{\Omega} (\nabla \cdot \overline{\overline{\sigma}} + \boldsymbol{F_{ext}}) \ \mathrm{dv} = \oint_{\partial\Omega} \overline{\overline{\sigma}}\boldsymbol{n} \ \mathrm{ds} + \int_{\Omega} \boldsymbol{F_{ext}} \ \mathrm{dv} \tag{1.3}$$

The **constitutive equation** of a given fluid gives the expression of the viscous stress tensor $\overline{\overline{\tau}}$ with respect to other physical quantities. Most commonly encountered fluids, such as water, are Newtonian. In this case, the viscous stress is linear with respect to the strain rate. More complex fluids are known as non-Newtonian and can display a wide range of behaviors like paint or honey. Once $\overline{\overline{\tau}}$ is specified, the deduction of the total stress tensor $\overline{\overline{\sigma}}$ follows.

Energy conservation principle: The first law of thermodynamics states the change in the internal energy of a closed system is equal to the amount of heat Q supplied to the system, minus the amount of work W done by the system on its surroundings. Let $\boldsymbol{q}$ be the conductive heat flux vector accounting for thermal conduction on the boundaries of the control volume. We also consider an additional volumetric heat source $\mathcal{Q}$ that may arrise from radiation or chemical reactions. Because Ω is an open system, we need to take into account the energy flux due to the particles entering and leaving the control volume:

$$\frac{\mathrm{d}}{\mathrm{dt}}\left[\int_{\Omega}\rho e \ \mathrm{dv}\right]+\oint_{\partial\Omega}\rho e\left(\boldsymbol{u}\cdot\boldsymbol{n}\right)\ \mathrm{ds}=\oint_{\partial\Omega}\left(\overline{\overline{\sigma}}\boldsymbol{n}\right)\cdot\boldsymbol{u}\ \mathrm{ds}+\int_{\Omega}\boldsymbol{F_{ext}}\cdot\boldsymbol{u}\ \mathrm{dv}-\oint_{\partial\Omega}\boldsymbol{q}\cdot\boldsymbol{n}\ \mathrm{ds}+\int_{\Omega}\mathcal{Q}\ \mathrm{dv} \tag{1.4}$$

where k is the thermal conductivity of the fluid, T the absolute temperature of the fluid, $e = U + \frac{1}{2}\left\|\boldsymbol{u}\right\|^2$ the total energy defined as the sum of the internal energy U and the kinetic energy.

For compressible flows the relation between density, pressure and temperature is given by a special equation called **equation of state**. The most commonly used one is the ideal gas relation $P = \rho RT$, R being the gas constant. This additional equation closes the set of equations.

Differential form of the Navier-Stokes equations

As equations (1.1), (1.2) and (1.4) remain valid for a control volume Ω as small as we want, we can apply the divergence theorem (A.2) to get equivalent local equations. This relies on the continuum hypothesis, stating that a fluid can be regarded as a continuum rather than a collection of individual molecules. Flows where molecular effects are of significance are known as rarefied flows. Most liquids can be considered as a continua, as can gases under ordinary circumstances.

Conservation of mass:

$$\frac{\partial\rho}{\partial t}+\nabla\cdot\left(\rho\boldsymbol{u}\right)=0 \tag{1.5}$$

Conservation of momentum:

$$\frac{\partial\rho\boldsymbol{u}}{\partial t}+\nabla\cdot\left(\rho\boldsymbol{u}\otimes\boldsymbol{u}\right)=\nabla\cdot\overline{\overline{\sigma}}+\boldsymbol{F_{ext}} \tag{1.6}$$

Conservation of energy:

$$\frac{\partial\rho e}{\partial t}+\nabla\cdot\left(\rho\boldsymbol{u}\right)=\nabla\cdot\left(\overline{\overline{\sigma}}\boldsymbol{u}\right)+\boldsymbol{F_{ext}}\cdot\boldsymbol{u}-\nabla\cdot\boldsymbol{q}+\mathcal{Q} \tag{1.7}$$

The principle of conservation of angular momentum implies that the total stress tensor is symmetric, which means that $\overline{\overline{\sigma}}$ defines a self-adjoint linear operator. The local equation for

energy (1.7) is obtained by using this additional equality:

$$\oint_{\partial\Omega} \left(\overline{\overline{\sigma}}\boldsymbol{n}\right) \cdot \boldsymbol{u} \text{ ds} = \oint_{\partial\Omega} \boldsymbol{n} \cdot \left(\overline{\overline{\sigma}}\boldsymbol{u}\right) \text{ ds} = \int_{\Omega} \nabla \cdot \left(\overline{\overline{\sigma}}\boldsymbol{u}\right) \text{ dv} \tag{1.8}$$

Equations (1.6) and (1.7) can be simplified using equation (1.5), yielding the following non-conservative local forms of momentum and energy equations:

$$\rho \left[\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u}\right] = \nabla \cdot \overline{\overline{\tau}} - \nabla P + \boldsymbol{F_{ext}} \tag{1.9a}$$

$$\rho \left[\frac{\partial e}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, e\right] = \nabla \cdot \left(\overline{\overline{\tau}}\boldsymbol{u}\right) - \nabla \cdot (P\boldsymbol{u}) + \boldsymbol{F_{ext}} \cdot \boldsymbol{u} - \nabla \cdot \boldsymbol{q} + \mathcal{Q} \tag{1.9b}$$

In addition, taking the dot product of equation (1.9a) with the velocity $\boldsymbol{u}$ and subtracting it to equation (1.9b) together with equations (A.1a) and (A.1c) gives the equation relating to the internal energy $U = e - \frac{1}{2}(\boldsymbol{u} \cdot \boldsymbol{u})$:

$$\rho \left[\frac{\partial U}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, U\right] = \overline{\overline{\tau}} : \nabla \boldsymbol{u} - P(\nabla \cdot \boldsymbol{u}) - \nabla \cdot \boldsymbol{q} + \mathcal{Q} \tag{1.10}$$

In this equation, the : operator denotes the double dot product between two tensors. This new equation allows us to define a new equation based on the enthalpy h which is related to the internal energy through $h = U + P/\rho$:

$$\rho \left[\frac{\partial h}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, h\right] = \overline{\overline{\tau}} : \nabla \boldsymbol{u} + \left[\frac{\partial P}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, P\right] - \nabla \cdot \boldsymbol{q} + \mathcal{Q} \tag{1.11}$$

This equation is easier to obtain by going back to the conservative formulation of equation (1.10) using equation (1.5) and by using usual divergence formulas (A.1).

Finally we can relate enthalpy h to the temperature of the fluid T using the following differential equation:

$$\text{dh} = C_p \text{dT} + \frac{1}{\rho}(1 - \alpha T)\text{dP} \tag{1.12}$$

where C_p denotes the heat capacity at constant pressure and α the bulk expansion coefficient also known as the cubic thermal expansion coefficient. We also consider that the heat transfer by conduction is governed by Fourier's law $\boldsymbol{q} = -k\nabla T$, giving the following equation for the temperature:

$$\rho C_p \left[\frac{\partial T}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, T\right] = \overline{\overline{\tau}} : \nabla \boldsymbol{u} + \alpha T \left[\frac{\partial P}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, P\right] + \nabla \cdot (k\nabla T) + \mathcal{Q} \tag{1.13}$$

For an ideal gas the variation of enthalpy dh is independent of the variation of pressure dP and thus $\alpha T = 1$. Liquids generally have higher expansivities than solids because their bulk expansion coefficient greatly vary with temperature. For example water has an anomalous thermal expansion coefficient as it is densest at 3.983 $°C$ where it reaches $\rho = 999.973\ kg/m^3$.

Summary of unknowns in a d-dimensional space:

Physical quantity	Variable	Number of scalar unknowns
Fluid velocity	$\boldsymbol{u}$	d
Fluid pressure	P	1
Fluid density	ρ	1
Thermal conductivity	k	1
Thermal expansion	α	1
Absolute temperature	T	1
Conductive heat flux	$\boldsymbol{q}$	d
Total stress tensor	$\overline{\overline{\tau}}$	$d(d+1)/2$
	Total	$d(d+5)/2+5$

Summary of equations in a d-dimensional space:

Physical equation	Expression	Number of scalar equations
Conservation of mass	(1.5)	1
Conservation of momentum	(1.6)	d
Conservation of energy	(1.13)	1
Equation of state for density	$\rho(P,T)$	1
Eq. of state for thermal conductivity	$k(P,T)$	1
Eq. of state for thermal expansion	$\alpha(P,T)$	1
Fourier's law	$\boldsymbol{q} = -k\nabla T$	d
Constitutive equation	$\overline{\overline{\tau}}(\boldsymbol{u}, P, T, ...)$	$d(d+1)/2$
	Total	$d(d+5)/2+5$

Full Navier-Stokes-Fourier system of equations:

The Navier-Stokes-Fourier system describes the motion of a compressible, viscous and heat conducting fluid, and can be expressed as the following:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{1.14a}$$

$$\rho \left[\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} \right] = \nabla \cdot \overline{\overline{\tau}} - \nabla P + \boldsymbol{F_{ext}} \tag{1.14b}$$

$$\rho C_p \left[\frac{\partial T}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, T \right] = \overline{\overline{\tau}} : \nabla \boldsymbol{u} + \alpha T \left[\frac{\partial P}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, P \right] + \nabla \cdot (k \nabla T) + \mathcal{Q} \tag{1.14c}$$

The Navier-Stokes equations are a simplification of this model for a Newtonian isothermal fluid. When temperature is not involved, the energy equation (1.14c) is not required anymore.

1.1.2 Vorticity formulation

The vorticity is a field that describes the local spinning motion of a continuum near some point, as would be seen by an observer located at that point and traveling along with the flow. Mathematically, the vorticity $\boldsymbol{\omega}$ is a pseudovector field defined as the curl of the velocity:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{1.15}$$

Some phenomena are more readily explained in terms of vorticity, rather than the basic concepts of pressure and velocity. It is possible to rewrite the momentum equation (1.14b) in terms of vorticity with the help of equation (1.15):

$$\boxed{\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,\boldsymbol{\omega} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} - \boldsymbol{\omega}(\nabla \cdot \boldsymbol{u}) + \frac{1}{\rho^2}(\nabla\rho \times \nabla P) + \nabla \times \left(\frac{\nabla \cdot \overline{\overline{\tau}} + \boldsymbol{F_{ext}}}{\rho}\right)} \tag{1.16}$$

Note that if the density ρ is constant, the baroclinic term that contains the pressure vanishes.

1.1.3 Newtonian fluids

A fluid is considered to be Newtonian only if the tensors that describe the viscous stress and the strain rate are related by a constant fourth order viscosity tensor $\overline{\overline{\overline{\overline{\mu}}}}$ that does not depend on the stress state and velocity of the flow:

$$\overline{\overline{\tau}}(\boldsymbol{u}, T) \;=\; \overline{\overline{\overline{\overline{\mu}}}}(T) : \nabla \boldsymbol{u} \tag{1.17}$$

If the fluid is also isotropic, this viscosity tensor reduces to two scalar coefficients, the molecular viscosity coefficient $\mu(T)$ and the bulk viscosity coefficient $\lambda(T)$. Those constants describe the fluid's resistance to shear deformation and compression, respectively:

$$\overline{\overline{\tau}}(\boldsymbol{u}, T) \;= \mu\left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T\right) + \lambda(\nabla \cdot \boldsymbol{u}) I_d \tag{1.18}$$

Stokes made the hypothesis that $\lambda = -(2/3)\mu$ which is frequently used but which has still not been definitely confirmed to the present day [Stokes 1880][Hak 1995]. This is not a problem because, as we will see later, the bulk viscosity λ can be neglected when the fluid can be regarded as incompressible.

For an isotropic Newtonian fluid, the divergence of the viscous stress tensor becomes:

$$\nabla \cdot \overline{\overline{\tau}} = \nabla \cdot \left[\mu\left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T\right) + \lambda(\nabla \cdot \boldsymbol{u}) I_d\right] \tag{1.19}$$

The divergence of the strain rate can be rearranged as:

$$2(\nabla \cdot \dot{\gamma}) = \nabla \cdot \left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T\right) = \Delta \boldsymbol{u} + \nabla(\nabla \cdot \boldsymbol{u}) \tag{1.20}$$

Expanding equation (1.19) using (1.20) and divergence formulas gives:

$$\nabla \cdot \overline{\overline{\tau}} = \mu \Delta \boldsymbol{u} + (\mu + \lambda)\nabla(\nabla \cdot \boldsymbol{u}) + \left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T\right)\nabla\mu + (\nabla \cdot \boldsymbol{u})\nabla\lambda \tag{1.21}$$

1.1.4 Compressible Navier-Stokes equations

Under the Stoke's hypothesis and considering an isotropic Newtonian fluid at constant temperature T_0 and constant molecular viscosity μ we finally obtain the compressible Navier-Stokes equations from equations (1.5), (1.6) and (1.21):

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{1.22a}$$

$$\rho \left[\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} \right] = \mu \left[\Delta \boldsymbol{u} + \frac{1}{3} \nabla (\nabla \cdot \boldsymbol{u}) \right] - \nabla P + \boldsymbol{F_{ext}} \tag{1.22b}$$

As stated before this set of equation is valid for non-rarefied flows where the representative physical length scale of the system is much larger than the mean free path of the molecules. As a side note we will only consider isotropic Newtonian fluids throughout the manuscript.

By dividing by the density, equations (1.22a) and (1.22b) can be rewritten as the following:

$$\frac{1}{\rho} \left[\frac{\partial \rho}{\partial t} + (\boldsymbol{u} \cdot \nabla) \rho \right] + \nabla \cdot \boldsymbol{u} = 0 \tag{1.23a}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} = \nu \left[\Delta \boldsymbol{u} + \frac{1}{3} \nabla (\nabla \cdot \boldsymbol{u}) \right] - \frac{\nabla P}{\rho} + \boldsymbol{f_{ext}} \tag{1.23b}$$

where $\nu = \mu / \rho$ is the **cinematic viscosity** and $\boldsymbol{f_{ext}} = \boldsymbol{F_{ext}} / \rho$ represents the **external mass forces** acting on the fluid.

1.1.5 Incompressible flows

An incompressible flow refers to a flow in which the material density is constant within a infinitesimal control volume that moves with the flow velocity. From the continuity equation (1.23a), an equivalent statement that implies incompressibility is that the divergence of the flow velocity is zero:

$$\frac{\partial \rho}{\partial t} + (\boldsymbol{u} \cdot \nabla) \rho = -\rho (\nabla \cdot \boldsymbol{u}) = 0 \tag{1.24}$$

Under the incompressibility hypothesis the viscous stress tensor reduces to $\overline{\overline{\tau}} = \mu \left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T \right)$ for an isotropic Newtonian fluid, removing the need to provide the bulk viscosity λ. This allows us to define the incompressible Navier-Stokes equations with variable density and viscosity:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.25a}$$

$$\rho \left[\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} \right] = \mu \Delta \boldsymbol{u} + \left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T \right) \nabla \mu - \nabla P + \boldsymbol{F_{ext}} \tag{1.25b}$$

Note that this hypothesis also simplifies the energy conservation equation (1.14c):

$$\rho C_p \left[\frac{\partial T}{\partial t} + (\boldsymbol{u} \cdot \nabla) T \right] = 2\mu (\dot{\gamma} : \nabla \boldsymbol{u}) + \alpha T \left[\frac{\partial P}{\partial t} + (\boldsymbol{u} \cdot \nabla) P \right] + \nabla \cdot (k \nabla T) + \mathcal{Q} \tag{1.26}$$

In addition, if the density and the viscosity are assumed constant, is follows that the flow is incompressible and we can obtain the following set of equations:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.27a}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} = \nu \Delta \boldsymbol{u} - \frac{\nabla P}{\rho} + \boldsymbol{f_{ext}} \tag{1.27b}$$

that can also be expressed in terms of vorticity:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{1.28a}$$

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{\omega} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} + \nu \Delta \boldsymbol{\omega} + \nabla \times \boldsymbol{f_{ext}} \tag{1.28b}$$

It is important to note that an incompressible flow does not imply that the fluid itself is incompressible. Homogeneous fluids with constant density always undergo flow that is incompressible, but the reverse is not true. As an example it is possible to have an incompressible flow of a compressible fluid in the case of low Mach numbers or in stratified flows.

1.1.6 The Boussinesq approximation

The Boussinesq approximation is a way to solve the incompressible Navier-Stokes equations with a slightly varying density, without having to solve for the full compressible formulation of the Navier-Stokes-Fourier equations (1.14) applied to an isotropic Newtonian fluid (1.21). This approximation, introduced by Joseph Boussinesq in 1877, assumes that variations in density have no effect on the flow field, except that they give rise to buoyancy forces [Boussinesq 1877]. The Boussinesq approximation is mostly used to simplify the equations for non-isothermal flows, such as natural convection problems, but can also be used for problems where the density variations are not due solely to temperature variations inside the fluid.

General framework

Suppose that tiny density variations $\delta\rho \ll \rho_0$ are created by some physical phenomenon such that the density can be expressed as $\rho = \rho_0 + \delta\rho$ where ρ_0 is a constant. Injecting ρ into the compressible continuity equation (1.23a) gives:

$$\underbrace{\frac{1}{\rho}\left[\frac{\partial \delta\rho}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \delta\rho\right]}_{\simeq\, 0} + \nabla \cdot \boldsymbol{u} = 0 \tag{1.29}$$

Under the Boussinesq approximation, the continuity equation reduces to the incompressible form $\nabla \cdot \boldsymbol{u} = 0$ because the first term is assumed to be small compared to the velocity gradients.

Rewriting the momentum equation (1.22b) for an incompressible flow and using the buoyancy as only external force leads to:

$$\rho \left[\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} \right] = \mu \Delta \boldsymbol{u} - \nabla P + \rho \boldsymbol{g} \tag{1.30}$$

Taking into account that the density variations have only an effect on buoyancy forces yields:

$$\rho_0 \left[\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} \right] = \mu \Delta \boldsymbol{u} - \nabla P + \rho \boldsymbol{g} \tag{1.31}$$

Finally equation (1.31) can be rewritten as:

$$\rho_0 \left[\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} \right] = \mu \Delta \boldsymbol{u} - \nabla \mathcal{P} + \delta \rho \boldsymbol{g} \tag{1.32}$$

where $\boldsymbol{g} = -g\boldsymbol{e_z}$ and $\mathcal{P} = P + \rho_0 g z$ is referred to as a **pressure shift**. A pressure shift can be done whenever the external force vector field is conservative, meaning that there exists some scalar field ϕ such that $\boldsymbol{F_{ext}} = -\nabla \phi$, by defining the pressure term as $\mathcal{P} = P + \phi$.

Special case for temperature dependent density variations

If density variations are only due to temperature variations δT around some base temperature T_0, we can rewrite the buoyancy term $\delta \rho \boldsymbol{g}$ as $-\alpha \rho_0 (T - T_0) \boldsymbol{g}$ where α is the coefficient of thermal expansion. In this special case, the momentum equation becomes:

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} = \nu \Delta \boldsymbol{u} - \frac{\nabla \mathcal{P}}{\rho_0} - \alpha (T - T_0) \boldsymbol{g} \tag{1.33}$$

Keep in mind that this equation is only valid if the viscosity μ does not depend on temperature, an hypothesis that is generally assumed, along with constant thermal conductivity k. Under those additional assumptions and considering the energy conservation equation (1.14c) we get the following set of equations:

$$T = T_0 + \delta T \tag{1.34a}$$

$$\rho = \rho_0 \left(1 - \alpha \delta T\right) \tag{1.34b}$$

$$\mathcal{P} = P - \rho_0 g z \tag{1.34c}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.34d}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} = \nu \Delta \boldsymbol{u} - \frac{\nabla \mathcal{P}}{\rho_0} - \alpha \delta T \boldsymbol{g} \tag{1.34e}$$

$$\rho C_p \left[\frac{\partial \delta T}{\partial t} + (\boldsymbol{u} \cdot \nabla) \delta T \right] = \underbrace{2\mu \left(\dot{\gamma} : \nabla \boldsymbol{u}\right)}_{\text{viscous heating}} + \underbrace{\alpha T \left[\frac{\partial P}{\partial t} + (\boldsymbol{u} \cdot \nabla) P \right]}_{\text{work of pressure forces}} + k \Delta \delta T + \mathcal{Q} \tag{1.34f}$$

Most of the time, the work of pressure forces and the heat generated by viscous friction are neglected. Those terms however constitute intrinsic components in many physical problems like buoyancy-induced natural convection [Pons et al. 2007]. When those terms and heat $\mathcal{Q}$ can be neglected we can rewrite equation (1.14c) with the thermal conductivity κ_t as the following:

$$\frac{\partial \delta T}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \delta T = \kappa_t \Delta \delta T \tag{1.35}$$

1.1.7 Dimensionless numbers

In this subsection we quickly present some computational fluid dynamics related dimensionless numbers that will be used throughout the manuscript.

Reynolds number

The Reynolds number is used to predict the transition from laminar to turbulent flow, and may be used in the scaling of similar but different-sized flow situations. The predictions of onset of turbulence and the ability to calculate scaling effects can be used to help predict fluid behaviour on a larger scale.

- Laminar flow occurs at low Reynolds numbers, where viscous forces are dominant, and is characterized by smooth, constant fluid motion.
- Turbulent flow occurs at high Reynolds numbers and is dominated by inertial forces, which tend to produce chaotic eddies, vortices and other flow instabilities.

The Reynolds number is defined as:

$$Re = \frac{\text{inertial forces}}{\text{viscous forces}} = \frac{\rho u L}{\mu} = \frac{uL}{\nu}$$

where ρ is the density of the fluid, u is the velocity of the fluid with respect to the object, L is a characteristic length, μ the dynamic viscosity of the fluid and ν the kinematic viscosity of the fluid.

This concept was introduced by George Stokes in 1851 [Stokes 1851], but this number was named by Arnold Sommerfeld in 1908 [Sommerfeld 1908] after Osborne Reynolds who popularized its use in 1883 [Reynolds 1883].

Schmidt number

The Schmidt number Sc is a dimensionless number defined as the ratio of kinematic viscosity (momentum diffusivity) to the mass diffusivity of some transported component:

$$Sc = \frac{\text{viscous diffusion rate}}{\text{molecular diffusion rate}} = \frac{\nu}{\kappa} = \frac{\mu}{\rho\kappa}$$

where κ is the mass diffusivity, ρ is the density of the fluid, μ the dynamic viscosity of the fluid and ν the kinematic viscosity of the fluid.

A Schmidt number of unity indicates that momentum and mass transfer by diffusion are comparable, and velocity and concentration boundary layers coincide with each other. Thus it is used to describe whether momentum or diffusion will dominate mass transfer.

Prandtl and Lewis numbers

The Prandtl number Pr, named after the German physicist Ludwig Prandtl, is the heat transfer analog of the Schmidt number. It is defined as the ratio of momentum diffusivity to thermal diffusivity:

$$Pr = \frac{\text{viscous diffusion rate}}{\text{thermal diffusion rate}} = \frac{\nu}{\kappa_t} = \frac{\mu}{\rho\kappa_t}$$

The ratio between the thermal diffusivity and the mass diffusivity is defined as the Lewis number $Le = \frac{\kappa_t}{\kappa} = \frac{Sc}{Pr}$ and was named after Warren K. Lewis [Lewis 1922].

Rayleigh number

The Rayleigh number Ra for a fluid is a dimensionless number associated with buoyancy-driven flow (natural convection) named after Lord Rayleigh [Rayleigh 1916]. This number can be used to describe flows when the mass density of the fluid is non-uniform. When below a critical value, there is no flow and heat transfer is purely achieved by conduction, otherwise heat is transferred by natural convection [Çengel et al. 2001].

When the density difference is caused by temperature difference, it is defined as the following:

$$Ra = \frac{\text{time scale for thermal transport by diffusion}}{\text{time scale for thermal transport by convection}} = \frac{gL^3}{\nu\kappa_t}\Delta\rho \simeq \frac{\rho g L^3}{\nu\kappa_t}\alpha\Delta T$$

where α is the thermal expansion coefficient of the fluid, ρ is the density of the fluid, g is the acceleration due to gravity, L is a characteristic length, ν the kinematic viscosity of the fluid, κ_t is the thermal diffusivity of the fluid and $\Delta\rho$ and ΔT represent respectively density and temperature differences.

Kolmogorov microscales

Kolmogorov microscales are the smallest scales in turbulent flow. At the Kolmogorov scale, viscosity dominates and the turbulent kinetic energy is dissipated into heat:

- Kolmogorov length scale: $\eta_K = \left(\frac{\nu^3}{\epsilon}\right)^{\frac{1}{4}}$
- Kolmogorov time scale: $\tau_K = \left(\frac{\nu}{\epsilon}\right)^{\frac{1}{2}}$
- Kolmogorov velocity scale: $u_K = (\nu\epsilon)^{\frac{1}{4}}$

where ν is the kinematic viscosity and ϵ is the average rate of dissipation of turbulence kinetic energy per unit mass. The Kolmogorov length scale ν can be obtained as the scale at which the Reynolds number is equal to 1, $Re = \frac{uL}{\nu} = 1$, $L = \frac{\nu}{U} = \frac{\mu}{\rho U}$.

1.1.8 Numerical resolution

The simulation of turbulent flows by numerically solving the Navier-Stokes-Fourier equations (1.14) requires resolving a very wide range of time and length scales, all of which affect the flow field. Such a time and space resolution can be achieved with Direct Numerical Simulation (DNS), but is computationally expensive. Its computational cost can prohibit simulation of practical engineering systems with complex geometry or flow configurations. To circumvent those limitations, various class of mathematical models have been developed to numerically solve complex fluid problems.

Direct Numerical Simulation (DNS)

A direct numerical simulation is a simulation in computational fluid dynamics in which the Navier-Stokes equations are numerically solved without any turbulence model. This means that the whole range of spatial and temporal scales of the turbulence must be resolved. All the spatial scales of the turbulence must be resolved on the computational mesh, from the smallest dissipative scales (Kolmogorov microscales), up to the integral scale L associated with the motions containing most of the kinetic energy. In 3D we have that the number of mesh points satisfying this condition is proportional to $Re^{\frac{9}{4}} = Re^{2.25}$ and the number of time steps is proportional to $Re^{\frac{3}{4}}$ implying that the number of floating point operations grows as Re^3. Therefore, the computational cost of a fully resolved DNS is very high, even at low Reynolds numbers.

Large Eddy Simulation (LES)

The principal idea behind LES is to reduce the computational cost by ignoring the smallest length scales, which are the most computationally expensive to resolve, via low-pass filtering of the Navier-Stokes equations. Such a low-pass filtering, which can be viewed as a time- and spatial-averaging, effectively removes small-scale information from the numerical solution. This information is not irrelevant, however, and its effect on the flow field must be modeled, a task which is an active area of research for problems in which small-scales can play an important role, such as near-wall flows, reacting flows, and multiphase flows.

Reynolds-averaged Navier-Stokes equations (RANS)

The Reynolds-averaged Navier-Stokes equations are time-averaged equations of motion of fluid flow. The idea behind the equations is Reynolds decomposition, whereby an instantaneous quantity is decomposed into its time-averaged and fluctuating quantities, an idea first proposed by Osborne Reynolds. The RANS equations are primarily used to describe turbulent flows. These equations can be used with approximations based on knowledge of the properties of flow turbulence to give approximate time-averaged solutions to the Navier-Stokes equations.

1.1.9 Vortex methods

Since their first use in the 1930s [Prager 1928][Rosenhead 1931], vortex methods have gained attraction in the numerical fluid mechanics community by their ability to model accurately, robustly and naturally flows where the convective phenomenon is dominant. However, their widespread use was hampered by their difficulty to take into account adhesion conditions on the boundaries of immersed solids and to treat effects due to the viscosity of the flow [Cottet et al. 2000]. This last point is explained by the phenomenon of grid distortion which is characteristic of lagrangian methods. This distorsion is due to the accumulation or rarefaction of the particles in areas of strong velocity gradients.

Despite multiple studies until the 1990s in order to remedy these weaknesses, the vortex method did not achieve to impose themselves in their original context [Chorin 1973][Leonard 1975][Chorin 1978][Leonard 1985]. The emergence of particle remeshing schemes and their use in hybrid eulerian/lagrangian solvers enabled the vortex methods to circumvent their intrinsic difficulties while maintaining their original strengths [Anderson et al. 1985][Cottet 1988]. The remeshing procedure allowed to redistribute the particles on an underlying grid, ensuring a distortionless procedure with uniform particle distribution, and the use of eulerian methods on Cartesian grid facilitated the treatment of viscous effects. The gain of competitiveness obtained with this new semi-lagrangian framework then aroused many efforts, particularly in terms of algorithmic development and high performance computing, resulting in multicore distributed parallel solvers [Sbalzarini et al. 2006][Lagaert et al. 2014] and efficient implementations of high order remeshing kernels on coprocessors and accelerators [Rossinelli et al. 2008][Etancelin et al. 2014]. The remeshed particle method is described in section 2.3.

1.1.10 Scalar transport in high Schmidt number flows

The prediction of the dynamics of a scalar transported in a turbulent flow constitute an important challenge in many applications. A scalar field can describe a temperature field, the concentration of chemical species or even a levelset function that captures the interface in multiphase flows [Shraiman et al. 2000]. For a passive scalar θ, the transport equation is an advection-diffusion equation that is coupled with the velocity of the carrier-fluid:

$$\boxed{\frac{\partial \theta}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,\theta = \kappa \Delta \theta} \tag{1.36}$$

where κ is the molecular scalar diffusivity and $\boldsymbol{u}$ the turbulent velocity of the carrier fluid.

Similarly to the Kolmogorov length scale η_K defined in section 1.1.7, the Batchelor scale η_B describes the smallest physical length scales that the scalar will develop before being dominated by molecular diffusion [Batchelor 1959]. The Schmidt number is a dimensionless parameter defined as the viscosity-to-diffusivity ratio:

$$S_c = \frac{\nu}{\kappa} \tag{1.37}$$

The Batchelor and Kolmogorov scales are related by $\eta_B = \eta_K/\sqrt{S_c}$. For a Schmidt number larger than one, the Batchelor scale is thus smaller than the Kolmogorov scale and scalar dynamics can occur at scales smaller than the smallest velocity eddies. Direct numerical simulations of turbulent transport with a pseudo-spectral methods have been conducted by [Donzis et al. 2010] to study universal scaling laws of passive scalars. For Schmidt numbers higher than one, they investigated three-dimensional grid discretizations up to $N = 2048$ in each direction with a Fourier-based spectral solver to determine the effects of the grid resolution on small-scale scalar statistics. Their main conclusion is that the grid resolution should effectively be of the order of the Batchelor scale $\Delta x \simeq \eta_K$.

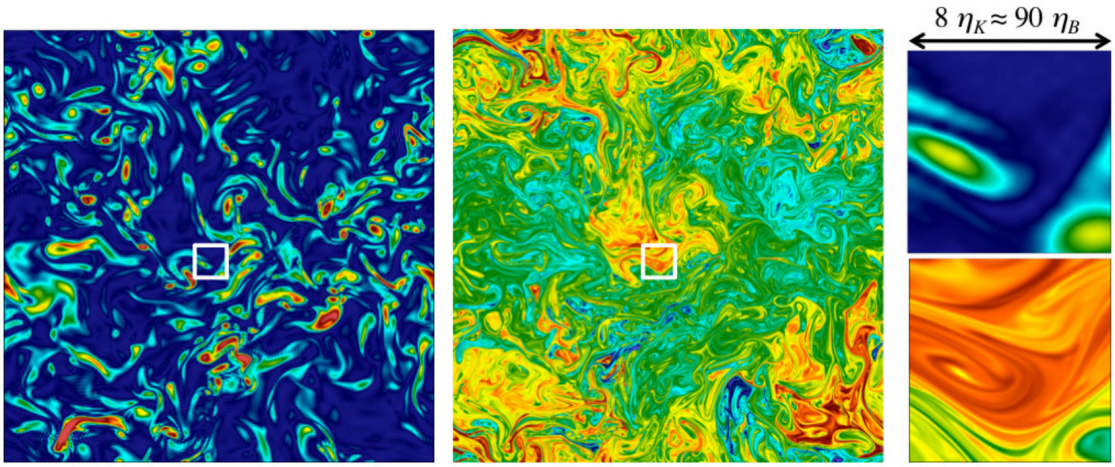


Figure 1.1 – Three-dimensional turbulent scalar transport at high Schmidt number: Slice of the norm of the vorticity (left, coarse grid 256^3) and passive scalar (right, fine grid 3064^3) obtained by [Lagaert et al. 2014] for a Schmidt number $S_c = 128$.

For numerical simulations, the dependency of the Batchelor scale on the Schmidt number suggests that the prediction of scalar dynamics for high Schmidt numbers is more demanding in terms of spatial resolution than the prediction of momentum. The two-scale nature of turbulent scalar transport makes it natural to use different grids [Gotoh et al. 2012], however, classical advection schemes for the scalar equation impose a CFL condition which requires adaptation of the time step to the finest scales, thus increasing the computational cost. Particle methods, that are not constrained by any CFL condition, can overcome this limitation. This technique has been introduced by [Cottet et al. 2009a] in the context of LES simulations of homogeneous isotropic turbulence where both the momentum and scalar equations were solved by particle methods at different resolutions. This method leads to a significative speed up over more conventional grid-based methods and allows to address challenging Schmidt numbers [Lagaert et al. 2012]. A distributed hybrid spectral-particle method has also been proposed in [Lagaert et al. 2014] where three-dimensional simulations up to $S_c = 128$ were performed (see figure 1.1). In this case, the grid resolution was 256^3 for the flow variables $(\boldsymbol{u}, \boldsymbol{\omega})$ and 3064^3 for the scalar θ. The speed-up provided by the spectral-particle method over a purely spectral method was estimated to be around $\mathcal{O}\left(10^2\right)$.

1.2 Sediment flows

Dispersion of solid particles in turbulent shear flows is of importance in many natural and engineering processes. The transport of sediments in oceans and slurries in pipes are typical examples. At sufficiently high mass loading, the particles have an effect on the turbulence structure of the carrier fluid and thus modify the transport of physical quantities. To predict these transport phenomena, the knowledge of the coupling between the particles and the fluid is required. This section introduces sediment transport processes and aims to give a state of the art of existing fluid-particles coupling models.

1.2.1 Sediment transport mechanisms

A sediment is a naturally occurring material that originates from the alteration of continental geological formations. Sedimentary erosion, transport and deposit are driven by many physical processes, their transport being mainly due to wind, water and gravity. As an example, sand can be carried in suspension by rivers, reaching the sea bed where they will be deposited by sedimentation once the flow becomes sufficiently quiescent. Their geological origin determining their chemical composition, sediments exhibit a huge variety of physical and mechanical properties including varying sizes, shapes and densities.

In the literature, many sediment size classifications have been proposed [Schmid 1981] [Moncrieff 1989]. Those classifications are important because the granulometry plays a great role into the transport mechanisms: the same sediment calibers will not be found near or far from the erosion zone as it can be seen on figure 1.2. As we may expect, the finest sediments will be transported as suspensions and the coarsest sediments that are heavier will travel the least distance from the erosion zone. The granulometry is not the only factor that should be taken

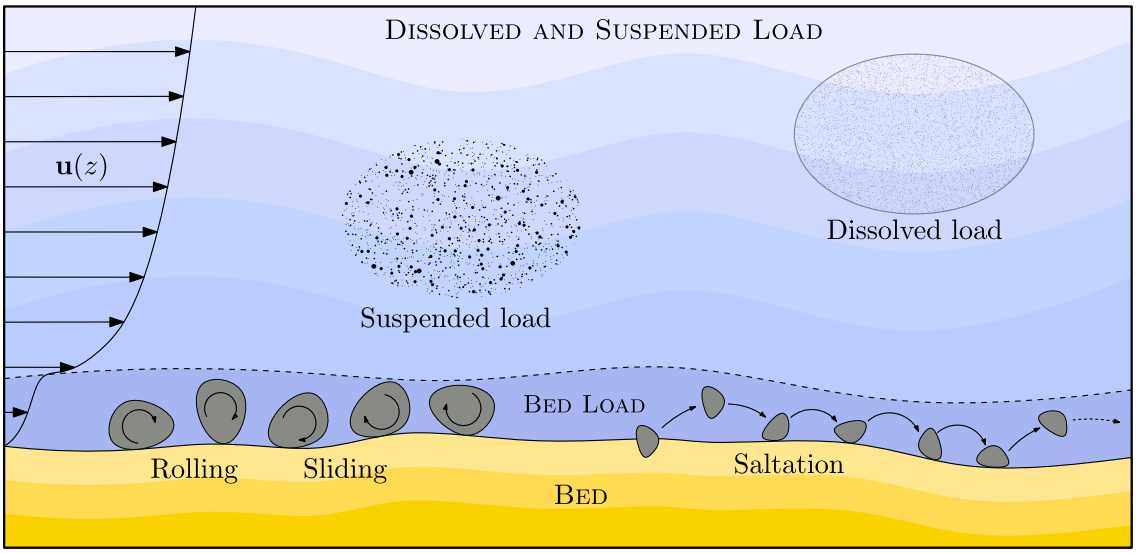


Figure 1.2 – Illustration of the different sediment transport mechanisms: the upper layer is composed of ions that are transported as dissolved load and light sediments like clay, silt and sand that are transported as suspended loads. The bed load is composed of intermediate size sediments like gravel alternating between transport and sedimentation phases, jumping on the bed (saltation) and pebbles and boulders rolling and sliding on the sea floor. A typical mean vertical flow velocity profile $\boldsymbol{u}(z)$ is also provided, showing the effect of the high concentration of sediments in the bed load, acting as a porous media, on the flow velocity.

in account, the amount of clay and organic matter is also of importance because it determines whether the sediments exhibit a cohesive behaviour or not. Gravels and coarse to medium sands are non-cohesive sediments that are constituted of particles that are independent from each other. In cohesive sediments however, particles are attracted between each other and form higher density aggregates that will settle [Teisson 1991] having significant effects on the flow [Edmonds et al. 2010]. Polydisperse flows are sediment-laden flows that contains sediment of different sizes and shapes. Polydispersity of particles or droplets introduces a wide range of length and time scales that introduce further modeling challenges.

The ability to transport sediments is constrained by the energy available in the flow. When water flows over the sediment bed, it generates a shear stress τ that may trigger the sediment transport if above a critical value. The comparison of the destabilizing force due to shear $\|\boldsymbol{F_s}\| \propto \tau d^2$ to the stabilizing gravity force acting on a single particle $\|\boldsymbol{F_g}\| \propto (\rho_s - \rho_f) g d^3$ gives a nondimensional number, the Shields number, that can be used to calculate the initiation of motion of sediments [Shields 1936]. It is defined as the following:

$$\tau_* = \frac{\tau}{(\rho_s - \rho_f) g d}$$

where ρ_s and ρ_f are the particle and fluid density. As the transport is facilitated when particles lay on an inclined bed, the formula can be adapted to depend on the slope angle [Fredsøe 1992].

Depending on the value of the Shields number, sediments will move differently. We usually distinguish four layers of sediment loads as represented on figure 1.2:

- **Dissolved load:** Dissolved matter is invisible, and is transported in the form of chemical ions, the simplest example being salt (NaCl). Water can disrupt the attractive forces that hold the sodium and chloride in the salt compound together and dissolve it. All flows carry some type of dissolved load as water is a universal solvent. The total dissolved solid load transported in major world rivers has been estimated to 3843 megatons per year [Zhang et al. 2007].

- **Suspended load:** The suspended load is composed of fine particles that are transported in suspension in the flow. Those particles are too large to be dissolved, but too light to lie on the sea bed. However, suspended loads can settle if the flow is quiescent or in the presence of cohesive particles that will agglomerate and thus settle on the bed.

- **Bed load:** The bed load is constituted of sediments that are in continuous or intermittent contact with the bed. One can distinguish pebbles and boulders that can only slide and roll on the bed and coarse sands and gravels that moves by successively jumping on the bed in the flow direction. This last process is called saltation. The fluid velocity rapidly decreasing near the sea bed, stratification occurs and the near bed sediment concentration rises, resulting in non-Newtonian rheologies [Faas 1984].

- **Bed:** When the sediment concentration exceeds a critical value, the structural density, a porous network of sediments is formed and a weak saturated soil is formed. Due to the continuous deposit of sediments, this structure will slowly collapse and compress on its own weight, increasingly consolidating the soil. The soil then consolidates even further due to time dependent processes such as thixotropy [Freundlich et al. 1936].

In reality, the transition between these four zone is rather gradual. This highlights the fact that sediment transport is a vertically continuous varying phenomenon. The Hjulström curve (fig. 1.3) can be used to determine whether a river flowing at a certain velocity will erode, transport, or deposit sediments of a certain size.

This simple model, proposed in the early 20th century, determines the state of a particle at a certain depth by the knowledge of its size and the velocity of the carrier fluid. On this curve we can already see that for cohesive sediment (clay and silt) the erosion velocity increases with decreasing grain size, highlighting the fact that the cohesive forces become more and more important as particles sizes get smaller. On the other hand, the critical velocity for deposition depends only on the settling velocity which decreases with decreasing particle size.

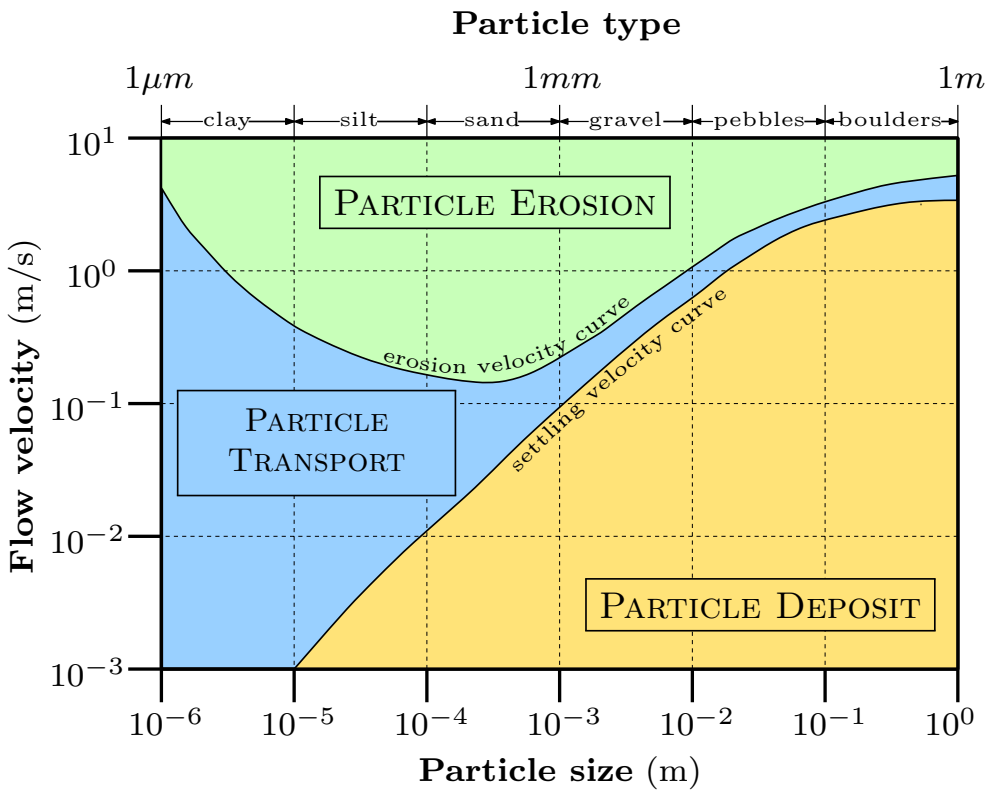


Figure 1.3 – Plot of the Hjulström curve in logarithmic scale. At high flow velocity, gravel, pebbles and boulders are transported on the sea bed whereas clay, silt and sand are already transported as a suspension at smaller velocities. The erosion velocity curve corresponds to the critical velocity where sediments begins to erode from the bed. Eroded sediments may then be transported in the carrier fluid at lower velocities until the critical settling velocity is reached, where they will deposit. Figure adapted from the original [Hjulstrom 1935] curve.

1.2.2 Sediments-laden flows classification

Physical models can be split into three classes depending on the volume fraction of particles in the carrier fluid and the ratio between particle response time and the Kolmogorov time scale [Elghobashi 1991] as it can be seen on figure 1.4. The particle response time τ_p characterizes how quickly a particle responds to turbulent fluctuations in the carrier fluid. In this subsection, ϕ_p is the volume fraction of particles in the carrier fluid, S represents the average distance between the centers of two neighboring particles, d is the diameter of a particle, τ_k is the Kolmogorov time scale and τ_e is the turnover time of large eddy.

When the volume fraction of particles is very small $\phi_p \leq 10^{-6}$, the particles effect on turbulence can be neglected. In this first case, the particles behaves as tracers: the particle dispersion only depends on the carrier fluid, and there is not feedback from particles to the fluid. This situation is often referred as "one way coupling".

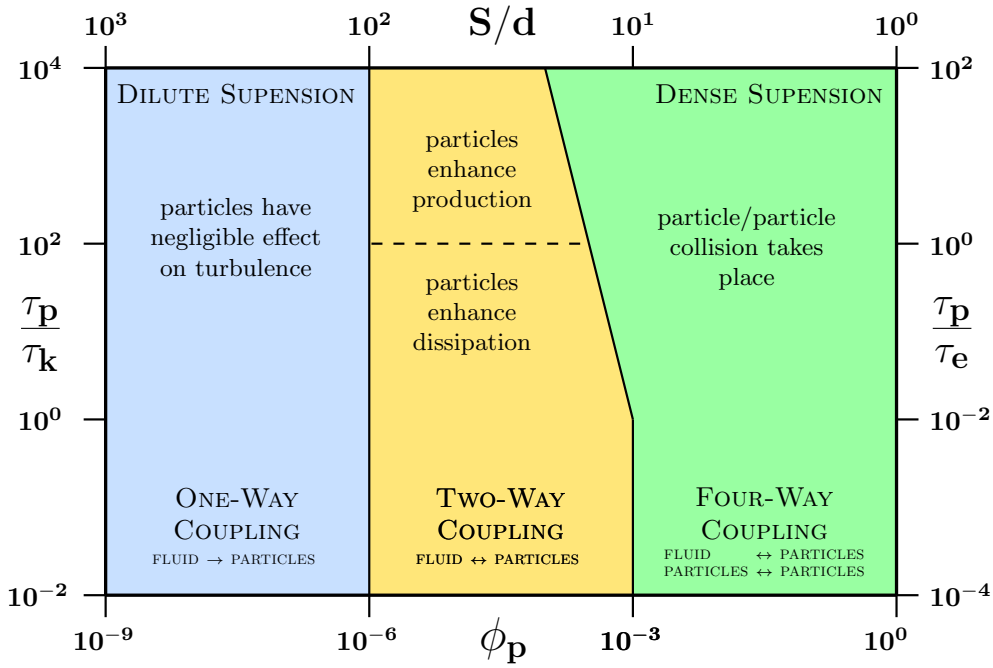


Figure 1.4 – Map of sediment laden flow regimes as a function of the volume fraction of the particles $\phi_p \propto \frac{d}{S}$ and the ratio between particle response time τ_p and the Kolmogorov time scale τ_k. Here d represents the particle diameter, S the distance between two neighboring particles and τ_e is the eddy turnover time. Figure adapted from [Elghobashi 1994].

The second regime characterizes dilute suspensions where the volume fraction ϕ_p is comprised between 10^{-6} and 10^{-3} In this case, the mass loading is high enough to modify the turbulence of the carrier fluid and "two way coupling" becomes necessary as particles modify the fluid characteristics. Depending on their size, particles can enhance either the production or the dissipation of turbulence energy:

1. Lowering the particles diameter d diminishes their response time τ_p and increases the total surface area of the particles. As a consequence, particles have a tendency to increase the dissipation rate of the turbulence energy.

2. Increasing the particles diameter d increases their response time τ_p and diminishes the total surface area of the particles. This increases the particles Reynolds number Re_p and vortex shedding builds up. As a result, particles increase the production of turbulence energy in the carrier fluid.

The third regime characterizes dense suspensions where the particle loading is greater than 0.1%. Because of the increased particle loading, we also have to take into account interactions between particles, hence the term "four way coupling". Increasing further the particle loading leads to collision-dominated flows (fluidized beds) followed by contact-dominated flows (nearly settled bed). When ϕ_p reaches 100%, there is no carrier fluid anymore and the flow becomes fully granular.

1.2.3 Numerical models

In this work, we are only interested in suspensions that are dilute enough to be accurately modelled by either one-way or two-way coupling ($\phi_p \leq 10^{-3}$), evolving as dissolved or suspended loads. One-way coupling just consists into adding an additional scalar advection-diffusion equation to the Navier-Stokes equations (see section 1.1.10). Some modelling difficulties arrise when considering suspended loads of sediments where two-way coupling becomes mandatory. Such flow configurations are referred to as sediment-laden flows and are included into the more general framework of particle-laden flows. In a particle-laden flow, one of the phases is continuously connected (the carrier phase) and the other phase is made up of small immiscible particles (the particle phase). This section presents the two main approches commonly used to mathematically represent particle-laden flows.

Eulerian-eulerian approach

In the EE[2] approach the carrier flow and the particles are modelled as a continuum, allowing the use of the same discretization techniques for both the fluid and the particles. This class of methods can further be split into two subclasses depending on how the particulate phase is modelled (deterministically or statistically):

1. **Deterministic separated-fluid formulation**: When the particles can be considered as a continuum behaving like the carrier fluid, is is possible to perform a two-way fluid-fluid coupling through two sets of Navier-Stokes equations modelling each continuum:

$$\phi_f + \phi_p = 1 \tag{1.38a}$$

$$\boldsymbol{F_{f\to p}} + \boldsymbol{F_{p\to f}} = 0 \tag{1.38b}$$

$$\frac{\partial \phi_k \rho_k}{\partial t} + \nabla \cdot (\phi_k \rho_k \boldsymbol{u}_k) = 0 \tag{1.38c}$$

$$\frac{\partial \phi_k \rho_k \boldsymbol{u}_k}{\partial t} + \nabla \cdot (\phi_k \rho_k \boldsymbol{u}_k \otimes \boldsymbol{u}_k) = \nabla \cdot (\phi_k \overline{\overline{\tau}}_k) + \phi_k \rho_k \boldsymbol{g} + \boldsymbol{F_{k'\to k}} \tag{1.38d}$$

where ϕ_k, ρ_k, $\boldsymbol{u}_k$ and $\overline{\overline{\tau}}_k$ represent the volume fraction, density, velocity and total stress tensor of phase $k \in \{f, p\}$. This two-fluid model is based on the particle-phase continuum assumption of [Drew et al. 2006]. Note that for the particulate phase, those quantities are to be determined for a huge number of particles in a small control volume which imposes $d \ll dx$. The fluid-particles interactions are taken into account through the interphase hydrodynamic force $\boldsymbol{F_{k'\to k}}$ resulting in a two-way coupling of the model. The total stress tensor of the particles can take into account the pressure gradient due to the carrier flow and for dense suspensions where $\phi_p \geq 0.1\%$, complex particle-particle interactions like collisions and particle viscous stress due to altered flow characteristics around each particle. If the particles are polydisperse, it is also possible to consider independently each particle group as an individual phase [Schwarzkopf et al. 2011].

[2] Eulerian-Eulerian

2. **Deterministic mixed-fluid formulation**: This formulation is a simplification of the separated-fluid formulation where the relative velocities between the fluid and particles $\boldsymbol{u}_f - \boldsymbol{u}_p$ are small compared to the predicted flow velocity field $\boldsymbol{u} = \boldsymbol{u}_f$. This simplification results in a unique set of Navier-Stokes equations for the two phases [Faeth 1987]:

$$\phi_f + \phi_p = 1 \tag{1.39a}$$

$$\rho = \phi_f \rho_f + \phi_p \rho_p \tag{1.39b}$$

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{1.39c}$$

$$\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) = \nabla \cdot \overline{\overline{\tau}} + \rho \boldsymbol{g} \tag{1.39d}$$

3. **Statistical kinetic method**: Statistical methods also make the assumption that particles can be seen as a continuum. They can be used to track the evolution of ensemble-averaged particle properties. In this first statistical method, the particles are described by a probability density function $W(\boldsymbol{u}_p, \boldsymbol{x}_p, t)$ which represents the probability that a particle has a certain velocity and position at given time. This framework being more general, the resulting transport equations are harder to derive but it becomes easier to handle near-wall boundary conditions. It has been first introduced by [Buyevich 1971], and since has been continuously improved by many authors including notably [Reeks 1980], [Derevich et al. 1988], [Swailes et al. 1997] and [Hyland et al. 1999]. Moreover, this framework tends to invalidate the traditional assumption that the particles behaves as a Newtonian fluid [Elghobashi 1983].

4. **Statistical generalized Langevin model**: This model is a generalization of the statistical kinetic model where the probability density function now also depends on the velocity of the carrier fluid $W(\boldsymbol{u}_f, \boldsymbol{u}_p, \boldsymbol{x}_p, t)$. It was first introduced by [Simonin et al. 1993] and is built upon the work of [Haworth et al. 1986] on the generalized Langevin model. However statistical approches are most suited for gases and are not relevant for sediment-laden flows were all the particles are solid.

Lagrangian-eulerian approach:

The LE[3] approach denotes a group of models where the carrier fluid is modelled as a continuum and the particles are described in a lagrangian frame. Within this framework all particles are tracked individually within the carrier fluid, allowing a better handling of particle-particle collisions [Subramaniam 2013]. We usually split those methods into the following categories:

1. **Fully-resolved DNS:** The exact Navier-Stokes equations are solved by imposing no-slip boundary conditions at each particle surface. In this case, particle collisions are usually modelled by using either a soft-sphere or hard-sphere collision model [Hoomans et al. 1996] [Matuttis et al. 2000]. This kind of simulation is very demanding in terms of compute resources but no additional closure models are required to compute the solution.

[3] Lagrangian-Eulerian

This method has been successfully applied for a limited amount of particles by [Zhang et al. 2005] and [Uhlmann 2005]. As the domain space step dx should be much smaller than the particle diameter to capture the surface of each particle d, increasing the number of particles quickly becomes a problem when considering thousands particles. However, boundary handling can be simplified by introducing an immersed boundary method to easily enforce the particle boundary conditions. By using the penalization method of [Kempe et al. 2012], particle-resolved DNS has been used to perform fluid simulations of a polydisperse sediment bed made up of thousands of particles, sheared by a viscous flow [Biegert et al. 2017]. More recently the same technique was used to study the settling of cohesive sediments [Vowinckel et al. 2019]. Simulations using levelsets and vortex penalization were performed in [Coquerelle et al. 2008] and [Jedouaa et al. 2019].

2. **Point-particle DNS:** If the particle size is smaller than the Kolmogorov scale, the particle can be considered as a point particles, relieving the $dx \ll d$ condition. The main difference with the fully-resolved DNS is that this model requires an additional closure model for interphase momentum transfer. As each particle can be evolved independently, it is possible to handle millions of particles [Kuerten 2016]. This category englobes two subcategories depending on the type of computational particles that is considered:

   - **PP-DNS with physical particles:** each particle represents a physical particle [Squires et al. 1991] [Sundaram et al. 1997].
   - **PP-DNS with stochastic particles:** each particle represents some particle density that interacts with the carrier fluid [Boivin et al. 1998].

   In point-particle based models statistical treatment of collisions is usually employed [O'Rourke 1981] [Schmidt et al. 2000].

3. **Point-particle LES:** It is also possible to take into account the carrier fluid using Large Eddy Simulation instead of DNS, leading to two new subcategories similar to the previous ones: point-particle LES with physical particles [Apte et al. 2003] and point-particle LES with stochastic particles [Sommerfeld 2001] [Okong'o et al. 2004].

From a numerical point of view, the lagrangian-eulerian approach minimizes numerical diffusion in the particulate-phase fields such as volume fraction or mean velocity when compared to eulerian-eulerian approaches. This comes at the cost of higher computational requirements compared to the EE averaged equations due to the particle-based representation of the multiphase flow, even when considering stochastic point particle based models.

1.3 Density-driven convection

For the rest of this chapter, we place ourselves into the eulerian-eulerian framework. This section describes some sedimentation problems that are buoyancy-driven such as gravity currents and double-diffusive fingering of salt and particle-laden flows.

1.3.1 Gravity currents

Gravity currents or density currents are a primarily horizontal flows in a gravitational field that is driven by a density difference [Benjamin 1968]. Gravity currents can be finite volume like in the event of a dam break, or continuously supplied like for lava flows. Depending on the parameters, mostly the density difference and the concentration, different approaches that can account for this kind of two-way coupling are to be considered. An overview of those approaches can be found in [Sinclair 1997] as well as in [Schwarzkopf et al. 2011]. In his book, [Simpson 1999] describes and illustrates a whole range of naturally occurring density-driven flows such as turbidity currents in the ocean, sea breeze fronts propagating inland from the coast, salt water wedges in river estuaries, and snow avalanches. The density difference that drives this kind of currents may be cause by the presence of additional chemicals in some regions of the flow, like in the case of fresh water flowing into salty sea water [Huppert 1982]. Thermal differences can also be the cause of density gradients creating gravity currents [Bercovici 1994]. Turbidity currents in the ocean are driven by suspended particles increasing the bulk density of the surrounding fluid [Bonnecaze et al. 1993]. In this specific case, the density can change through sedimentation and the entrainment of the lighter carrier fluid. [Britter et al. 1978] provides experimental and analytical data for the case of a heavy fluid released next to a lighter fluid which show that gravity currents mix trough Kelvin-Helmholtz billows generated at the moving head of the mixing layer.

The density difference between the two fluids can range from very small to very large. In many geophysical situations such as sea water and fresh water the density difference is very small (within 5%). In cases of small density difference, density variations can be neglected in the inertia term, but retained in buoyancy term. Hence, for dilute suspensions of sediments that are small enough such that their inertia can be neglected, the most convenient model uses a **single-phase Boussinesq fluid** whose density ρ depends only on carrier fluid base density ρ_0 augmented by the local particle concentration C.

The particulate phase is driven by a convection-diffusion equation for its concentration:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.40a}$$

$$\mathcal{P} = P - \rho_0 g z \tag{1.40b}$$

$$\rho = \rho_0 \left(1 + C\right) \tag{1.40c}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) \boldsymbol{u} = \nu \Delta \boldsymbol{u} - \frac{\nabla \mathcal{P}}{\rho_0} + C \boldsymbol{g} \tag{1.40d}$$

$$\frac{\partial C}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) C = \kappa_c \Delta C \tag{1.40e}$$

This simple model can directly be obtained from the incompressible eulerian-eulerian mixed-fluid formulation (1.39) in its Boussinesq approximation (1.34) where we used the so called pressure shift (1.32). This model is particularly adapted for the velocity-vorticity formulation as the gradient of shifted pressure disappears when taking the curl of the momentum equations leading to two scalar convection-diffusion equations in 2D. For three-dimensional problems we get three convection-diffusion-stretch equations for the vorticity and one convection-diffusion equation for the scalar concentration C:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.41a}$$

$$\rho = \rho_0 \left(1 + \alpha C\right) \text{ with } \alpha \ll 1 \text{ and } C \in [0, 1] \tag{1.41b}$$

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) \boldsymbol{\omega} = \nu \Delta \boldsymbol{\omega} + \nabla \times \left(C \boldsymbol{g}\right) \tag{1.41c}$$

$$\frac{\partial C}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) C = \kappa_c \Delta C \tag{1.41d}$$

This model has extensively been used by [Necker et al. 1999] to investigate particle-driven currents and to compare the results to a eulerian-lagrangian numerical model as described in subsection (1.2.3). Their results indicate that this model is a good candidate in the presence of low particle Stokes number. [Härtel et al. 2000] than further used direct numerical simulation of this model to perform a detailed analysis of the flow structure exhibited at the front of this kind of flows, where Kelvin-Helmholtz instabilities appear.

To illustrate the fact that simpler models than the full Navier-Stokes equations for the carrier fluid can be used we can take as an example [Bonnecaze et al. 1999]. Using only the shallow-water equations, they were able to solve for a turbulent current flowing down a uniform planar slope from a constant-flux point source of particle-laden fluid. This work has later been extended by [Ross 2000]. Another example of this kind would be [Monaghan et al. 1999] who used SPH to model gravity currents descending a ramp in a stratified tank.

There can be practical situations of interest where the density difference between the two fluids forming the gravity current is larger than a few percent. In order to study such flows, we cannot use the above Boussinesq approximation. We instead need to solve the mixed formulation involving the full incompressible Navier-Strokes equations with variable density (1.39) along with convection-diffusion equation for the particles concentration:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.42a}$$

$$\mathcal{P} = P - \rho_0 g z \tag{1.42b}$$

$$\rho = \rho_0 \left(1 + \alpha C\right) \text{ with } \alpha = \mathcal{O}\left(1\right) \text{ and } C \in [0, 1] \tag{1.42c}$$

$$\rho \left[\frac{\partial \boldsymbol{u}}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) \boldsymbol{u}\right] = \mu \Delta \boldsymbol{u} - \nabla \mathcal{P} + C \rho \boldsymbol{g} \tag{1.42d}$$

$$\frac{\partial C}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) C = \kappa_c \Delta C \tag{1.42e}$$

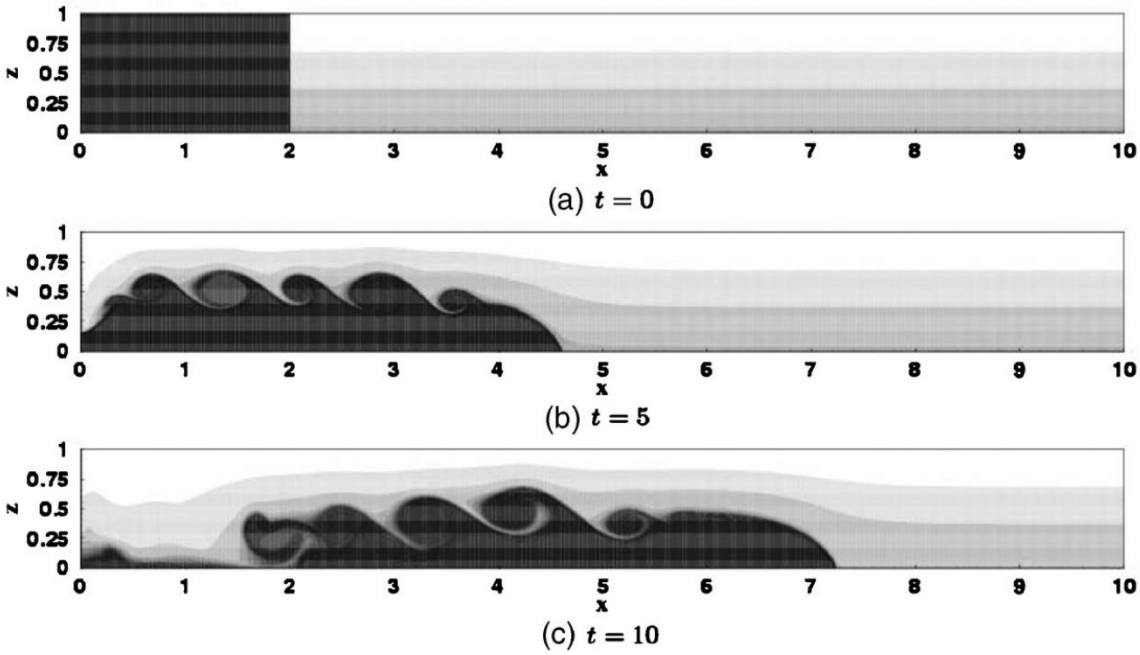


Figure 1.5 – Example of 2D gravity current obtained by [Birman et al. 2007].
A dense fluid is released in a linearly stratified ambient of lesser density. The density contours show the formation of an advancing front, behind Kelvin-Helmholtz instabilities giving rise to strong vortices.

By using equations 1.42, detailed numerical simulations of gravity currents released from a lock and propagating at the bottom have been first studied by [Ungarish et al. 2002] and extended later by [Birman et al. 2006] and [Birman et al. 2007] in the case of linearly stratified ambients, using this more complete model as illustrated on figure 1.5. Hoffmann and its collaborations than extended the numerical investigation to 2D buoyancy-driven flows in the presence of slopes [Hoffmann et al. 2015]. This model has also been used by Meiburg and collaborators to take into account complex 3D sea floor topologies like bumps [Nasr-Azadani et al. 2014][Meiburg et al. 2015].

More recently, DNS and LES simulations of three-dimensional non-Boussinesq gravity currents using a discontinuous Galerkin finite elements method have been conducted by [Bassi et al. 2018]. Multiple levels of stratifications have also been investigated by [Khodkar et al. 2018] using this model.

1.3.2 Double diffusive convection

Double diffusive gravity currents constitute a specific set of problems where two fluids, initially in a lock-exchange configuration, exhibit different densities that depends on some of their physical properties (concentration, temperature, ...) that diffuses at different rates creating two different density gradients. In this kind of configuration, the horizontal movement

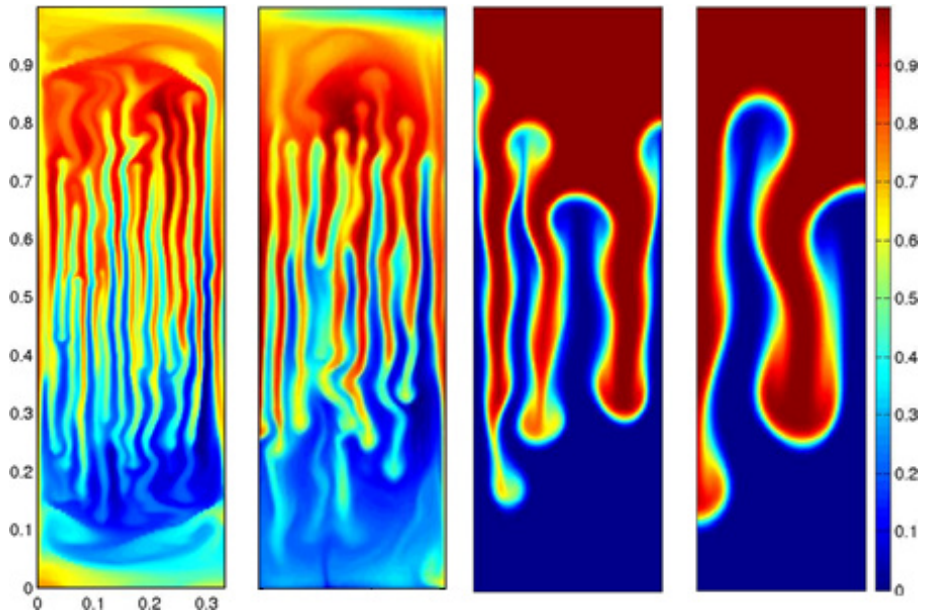


Figure 1.6 – Effect of Rayleigh numbers on the evolution of double-diffusive salt fingers: Numerical simulations results obtained by [Sreenivas et al. 2009a] that show concentration fields of a thermohaline system at different thermal Rayleigh numbers for fixed value of $R_\rho = 6$. Finger characteristics such as width, evolution pattern are a function of Rayleigh numbers. Image adapted from Salt fingers by Faria Rehman and licensed under CC BY-SA 4.0.

is initiated by initial density differences between each side of the lock as in classical lock-exchange configurations that have been presented in subsection (1.3.1). After the lock-gate is withdrawn, the light fluid flows over the heavy fluid transporting dissolved chemicals, physical particles or heat by convection down through the interface between the two fluids. This forms vertical density gradients, generating double diffusive convection oriented perpendicularly to the flow direction. In practice, double-diffusive processes are interesting enough that they are often studied independently without the initial horizontal lock-exchange configuration.

In double-diffusive problems, two basic type of convective instabilities may arrise: diffusive and fingering configurations (see figure 1.6 for the latter case). Such transport processes have long been ignored because their effects have been thought to be always overwhelmed by turbulent transport, but in both cases, the double diffusive fluxes can be much larger than the vertical transport in a single-component fluid because of the coupling between diffusive and convective processes [Turner 1985]. As an example, double diffusion convection plays an important role in upwelling of nutrients and vertical transport of heat and salt in oceans. Thus, finger convection helps to support flora and fauna. It also plays an important role in controlling the climate.

The form of the resulting vertical motions depends on whether the driving energy comes from the component having the high or lower diffusivity. When one layer of fluid is placed above another denser layer having different diffusive properties, two types of instabilities may arise:

- If the stable stratification is provided by the component with the lower molecular diffusivity, the stratification will be of "diffusive" type. This can happen when small pockets of saline water are trapped near the ocean bottom in the trenches where there is a supply of geothermal heat from hot vents [Kelley et al. 2003].
- On the other hand if the stable stratification is provided by the most diffusive component, the stratification will be called to be of "finger" type. This configuration is occurring frequently in oceanographic studies as salt fingers [Stern 1969].

Double diffusive convection has been actively studied during the last six decades as its effects are not limited to oceanography. It is also of importance in the understanding in a number of other phenomena that have multiple causes for density variations [Huppert et al. 1981] [Chen et al. 1984]. This includes convection in magma chambers [Huppert et al. 1984] and formation of basalt fingers [Singh et al. 2011] in geology, solidification of metallic alloys [Beckermann et al. 1988] in metallurgy, convection of solar winds due to temperature gradients and magnetic buoyancy [Hughes et al. 1988] and more recently convection in Jupiter due to hydrogen-helium phase separation [Nettelmann et al. 2015] in astrology. What is interesting in these systems is the wide variety of length scales and generated structures that are controlled by large variations in the governing parameters. Salt fingering in oceans operates on a length scale of centimeters over days while for the basalt columns formation the convective structures scale up to meters over time scales of decades [Singh et al. 2014].

Thermohaline convection and salt fingers

Historically double diffusive convection has been studied first in oceanography where heat and salt concentrations exist with different gradients and diffuse at differing rates. The term **thermohaline convection** has already been introduced in 1945 to describe this specific heat-salt system by Deacon and his collaborators when they studied water circulation and surface boundaries in the oceans [Deacon 1945]. A decade later, Stommel and its collaborators tried to describe an "oceanographic curiosity" relative to thermohaline convection that they named the salt fountain [Stommel 1956]. In 1960 Stern and its collaborators studied those salt fountains and observed that gravitationally stable stratification of salinity and temperature can be explained by the fact that the molecular diffusivity of heat κ_t is much greater than the diffusivity of salt κ_s [Stern 1960]. Some years later, experimental data obtained by [Ingham 1965] suggested that the relationship between temperature and salinity was much better described by a curve of constant density ratio R_ρ rather than by a straight line as showed on figure (1.7). Those early experimental observations were later taken as an evidence of double diffusive mixing by [Schmid 1981] using the empirical model proposed by [Chen et al. 1977].

Salt fingers appear when a hot and salty water lies over cold and fresh water of higher density (see figure 1.8). Salt fingers are formed because the temperature T, which is the fastest diffusing component, contributes to the negative density gradient ($\frac{\partial \rho_t}{\partial z} < 0$, stable stratification) and slow diffusing salinity S contributes to the positive density gradient ($\frac{\partial \rho_s}{\partial z} > 0$, unstable stratification) with overall density stratification remaining gravitationally stable ($\frac{\partial \rho}{\partial z} < 0$).

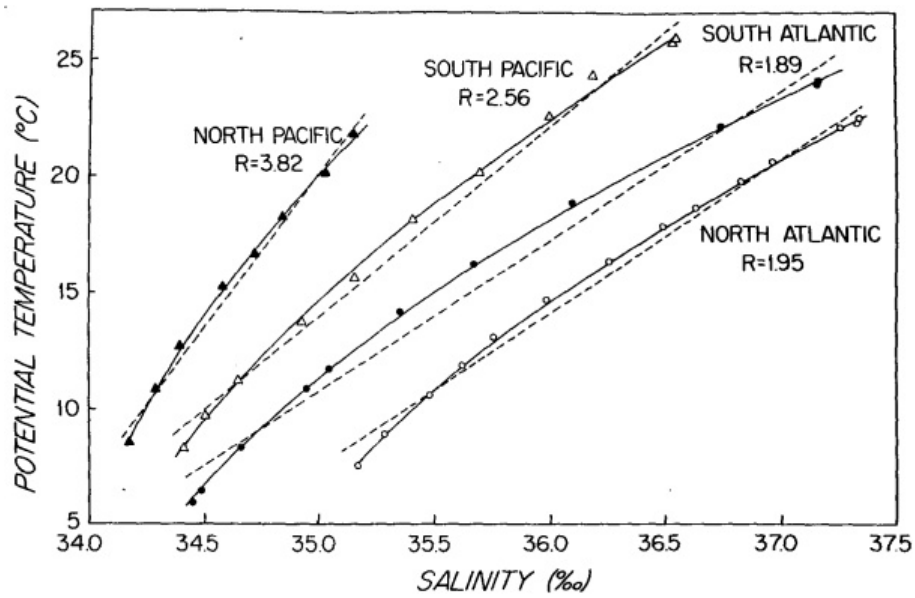


Figure 1.7 – Potential temperature versus salinity plot of four hydrographic stations in the North and the South Atlantic and the North and the South Pacific obtained by [Schmid 1981] based on the experimental work of [Ingham 1965] and the numerical model proposed by [Chen et al. 1977] to estimate density expansion parameters α and β. Dotted lines represents linear fit of data whereas solid lines represents optimal least squares fit of constant density ratio $R_\rho = \dfrac{\alpha \Delta T}{\beta \Delta S}$. See equations (1.43) for the modelization of thermohaline convection.

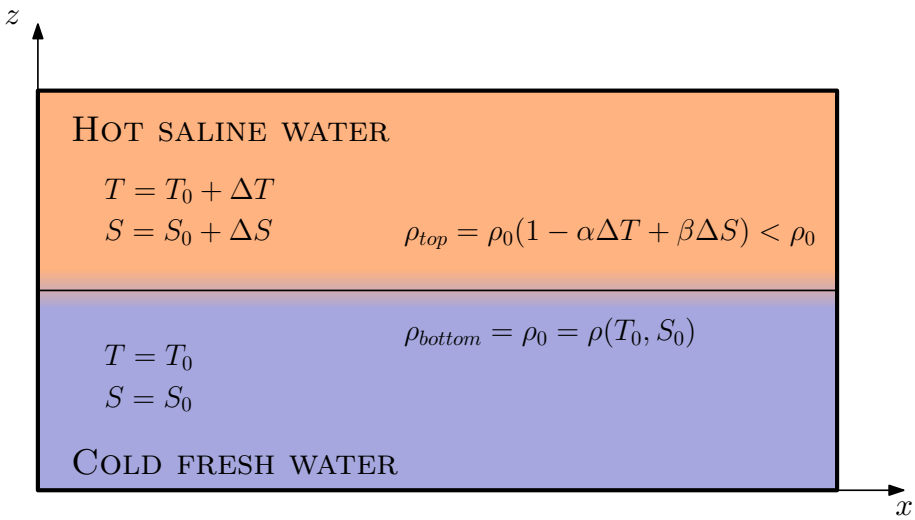


Figure 1.8 – Initial configuration for a thermohaline convection: hot and salty water lies over cold and fresh water of higher density ρ_0. Initial temperature and salinity differences, denoted ΔT and ΔS modify the bulk density of water ρ_0 by a factor $1 - \alpha\Delta T + \beta\Delta S < 1$ such that the system is initially gravitationally stable. The density stability ratio $R_\rho > 1$ is defined as the ratio between the thermal and salt density contributions. The temperature, diffusing 100 times faster than salt, will reduce density bellow the initial interface making the system gravitationally unstable before salinity can compensate for the sudden density change.

On the contrary, diffusive stratification occur when a deep layer of cold fresh water is placed above a warm salty (and heavier) layer. For example, such situation can occur in the case of a melting iceberg that inputs cold freshwater from above [Stern 1975]. See [Kelley et al. 2003] for a complete discussion about the diffusive configuration.

The usual model for salt fingering can be found in [Schmitt Jr 1979] and uses a single-phase Boussinesq fluid whose density ρ depends only on carrier fluid base density ρ_0 augmented by the local salinity S and temperature T analog to the one presented in section (1.1.6) and equations (1.40):

$$T = T_0 + \delta T \tag{1.43a}$$

$$S = S_0 + \delta S \tag{1.43b}$$

$$\rho = \rho_0 \left(1 - \alpha \delta T + \beta \delta S\right) \tag{1.43c}$$

$$\mathcal{P} = P - \rho_0 g z \tag{1.43d}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.43e}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) \boldsymbol{u} = \nu \nabla^2 \boldsymbol{u} - \frac{\nabla \mathcal{P}}{\rho_0} + (\beta \delta S - \alpha \delta T)\boldsymbol{g} \tag{1.43f}$$

$$\frac{\partial \delta T}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) \delta T = \kappa_t \nabla^2 \delta T \tag{1.43g}$$

$$\frac{\partial \delta S}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) \delta S = \kappa_s \nabla^2 \delta S \tag{1.43h}$$

In this simple model, the work of pressure forces and the heat generated by viscous friction are neglected, the notation ∇^2 has been used instead of Δ for the Laplacian operator, α corresponds to usual thermal expansion coefficient and β to the density expansion parameter taking into account the presence of additional salt in the flow. As stated before, κ_t represents the thermal diffusivity, κ_s is the diffusion coefficient for salt in water and we applied a pressure shift to get $\mathcal{P}$ from constant bulk density $\rho_0 = \rho(T_0, S_0)$.

Within this specific setup we can estimate that around $T_0 = 20°C$, the kinematic viscosity of water is approximately $\nu = 1.003 \times 10^{-6} m^2/s$, the thermal diffusivity of water is $\kappa_t = 1.430 \times 10^{-7} m^2/s$ and from [Caldwell 1973] the diffusivity of salt in sea water is $\kappa_s = 1.286 \times 10^{-9} m^2/s$ (which seems to agree with [Poisson et al. 1983]). This configuration yields the following dimensionless numbers comparing the different diffusivities that are involved in the double diffusive thermohaline convection process:

- **Schmidt number:** $Sc = \dfrac{\nu}{\kappa_s} = 7.799 \times 10^2 = \mathcal{O}\left(10^3\right)$
- **Prandtl number :** $Pr = \dfrac{\nu}{\kappa_t} = 7.014 \times 10^0 = \mathcal{O}\left(10\right)$
- **Lewis number :** $Le = \dfrac{\kappa_t}{\kappa_s} = \dfrac{Sc}{Pr} = 1.112 \times 10^2 = \mathcal{O}\left(10^2\right)$
- **Diffusivity ratio :** $\tau = Le^{-1} = \dfrac{\kappa_s}{\kappa_t} = 8.993 \times 10^{-3} = \mathcal{O}\left(10^{-2}\right)$

With those numbers, it is easy to see that in heat-salt systems, heat is the faster diffusing component (about 100 times) and that both heat and salt diffuse slower than velocity through viscous friction (by a factor 10 and 1000 respectively). The evolution of salt fingers is known to be influenced by many other non-dimensional parameters such as the thermal Rayleigh number Ra_t, the salinity Rayleigh number Ra_s and the initial density stability ratio R_ρ:

- **Initial stability ratio:** $R_\rho = \dfrac{\alpha \Delta T}{\beta \Delta S} \in]1, 10]$
- **Salinity Rayleigh number:** $Ra_s = \dfrac{\rho g L^3}{\nu \kappa_t} \beta \Delta S$
- **Thermal Rayleigh number:** $Ra_t = R_\rho Ra_s = \dfrac{\rho g L^3}{\nu \kappa_t} \alpha \Delta T \in [10^8, 10^{12}]$

It can be noted that when $R_\rho < 1$, the density gradient becomes gravitationally unstable. The initial density stability ratio R_ρ, the Schmidt number Sc and diffusivity ratio τ have been observed to be the most important non-dimensional parameter that controls the fingering behavior, see [Taylor et al. 1989] [Piacsek et al. 1980] [Traxler et al. 2011]. However, first decades of research on salt fingers has been done in systems where Rayleigh numbers are high (10^8 or higher, see [Kelley 1990]) but it has been recently observed that variation of flux ratios (the density fluxes anomaly due to the presence of heat and salt) also greatly depend on the Rayleigh numbers [Sreenivas et al. 2009b]. Smaller Rayleigh numbers than the ones naturally occurring for heat-salt systems were experimentally investigated by [Krishnamurti et al. 2002] and numerically by [Singh et al. 2014]. Finally it has been shown by [Huppert et al. 1973] that the stability criterion for finger formation in a two-layer system is $1 \ll R_\rho \ll Le^{3/2}$. Note that Pe can be much smaller for other configurations like salt-sugar systems where $Le \simeq 3$.

While forty years ago it was unclear that salt-fingering even existed outside of the laboratory, it is now well understood that the vertical flux of sea salt is dominated by double diffusive convection [Schmitt et al. 1978]. It was also observed that a small amount of horizontal water movement or turbulence could destroy the fingers [Linden 1971][Konopliv et al. 2018]. However, the fingers usually reformed within a few minutes when the flow became laminar again.

Double diffusive sediment-laden flows

During the seventies, a simple laboratory experiment showed that a similar fingering phenomenon occurred when a warm fluid, made denser by the presence of suspended material, was put above colder, distilled water [Houk et al. 1973]. As for thermohaline convection, the presence of an originally sharp horizontal interface between two such fluid layers, the system is double-diffusive and vertical sediment fingering motion occurs [Green 1987]. This sediment fingering can also occur when vertical sediment concentration gradients are associated with vertical temperature gradients instead of salinity gradients [Maxworthy 1999]. Temperature

can also be complementary to salinity as observed in hypersaline lakes [Ouillon et al. 2019]. The vertical motion of suspended particles being fundamental to sedimentation, it was then anticipated that the resulting double-diffusive instabilities could greatly contribute to the distribution of nearshore sediments when compared to gravitational settling alone in laminar flows.

In the nature, this event has been observed when a muddy river enters a stratified lake or ocean [Schmitt et al. 1978]. Such sediment-laden riverine outflows can be classified as either hypopycnal or hyperpycnal currents. An hyperpycnal current is a flow in which the density of the river is more than that of the stratified ambiant, so that the river continues to flow near the bottom of the sea floor [Mulder et al. 2003]. Such flows have been numerically investigated in [Schulte et al. 2016]. The flows of interest here are hypopycnal currents, where the combined density fresh water and suspended sediments is less than the density of the ocean. In this case the river continues to flow along the top of the stratified ambiant and sediment fingering can occur. As predicted, the resulting downward sediment flux can be orders of magnitude above the one that would normally be obtained with only gravitational settling [Hoyal et al. 1999][Parsons et al. 2000]. It has been later observed that double-diffusive fingering was not the only type of instabilities that could arrise. When sediment-laden water flows over clearer and denser water, the dominant instability mode may instead become Rayleigh-Taylor like, in which case the instabilities are located at the lower boundary of the particle-laden flow region [Burns et al. 2012].

The important parameter that drives the instabilities has been identified as being the ratio of the particle settling velocity to the diffusive spreading velocity of salinity or temperature layer that initially provides the stable stratification [Burns et al. 2015]. While the particle settling velocity can be estimated from physical particle properties [Gibbs 1985], the diffusivity of salinity κ_s or temperature κ_t is fully determined by the knowledge of associated Schmidt number between the considered field and water. The upward layer spreading velocity depends on this Schmidt number but also the other dimensionless parameters of the system, such as the stability ratio R_ρ and the diffusivity ratio τ between the salinity and sediments (or between temperature and sediments). In order for sediment fingers to form in hypopycnal flows, the particles have to diffuse slower than the considered agent ($\tau > 1$).

The numerical models used to explore the nonlinear behavior of convective sedimentation in hypopycnal flows are similar to the one of the heat-salt system (1.43). Those models consider dilute monodisperse distributions of fine particles with negligible inertia that can be considered as a continuum. The particle concentration C contributes positively to the density and is transported into the carrier fluid with respect to some relative particle settling velocity V_{st} that is usually the same everywhere in the flow [Yu et al. 2013]. In the case of the particle-salt system, equations (1.43c) and (1.43g) are thus replaced by the following expressions:

$$\boxed{\begin{aligned} \rho &= \rho_0 \left(1 + \alpha\delta S + \beta\delta C\right) && (1.44a) \\ \frac{\partial \delta C}{\partial t} + \left[\left(\boldsymbol{u} - V_{st}\boldsymbol{e_z}\right)\cdot\nabla\right]\delta C &= \kappa_c \nabla^2 \delta C && (1.44b) \end{aligned}}$$

The full set of equations relating to this particle-salt system is introduced in chapter 4.

The particle settling velocity can be determined directly from the grain diameter d for given values of carrier fluid viscosity ν_0 and density ρ_0. For dilute suspensions, Stokes law predicts the settling velocity of small spheres in fluid:

$$V_{st} = \frac{Rgd^2}{C_1 \nu_0} \tag{1.45a}$$

$$R = \frac{\rho_c - \rho_0}{\rho_0} \tag{1.45b}$$

where ρ_c is the density of particles, R represent the submerged specific gravity and C_1 is a constant that has a value of 12 for smooth spheres, and that is larger for non-spherical or rough particles [Raudkivi 1998]. Stokes law holds for particle Reynolds numbers bellow unity, else the settling of large particles is slowed down by the turbulent drag of the wake behind each particle:

$$V_{st} = \sqrt{\frac{4Rgd}{3C_2}} \tag{1.46a}$$

with $C_2 = 0.4$ for smooth spheres and $C_2 = 1$ for natural grains. The expressions for Stokes flow and turbulent drag law can be combined into a single expression that works for all sizes of sediment [Ferguson et al. 2004]. The fall velocity of various sizes of particle estimated with this law is shown on figure 1.9.

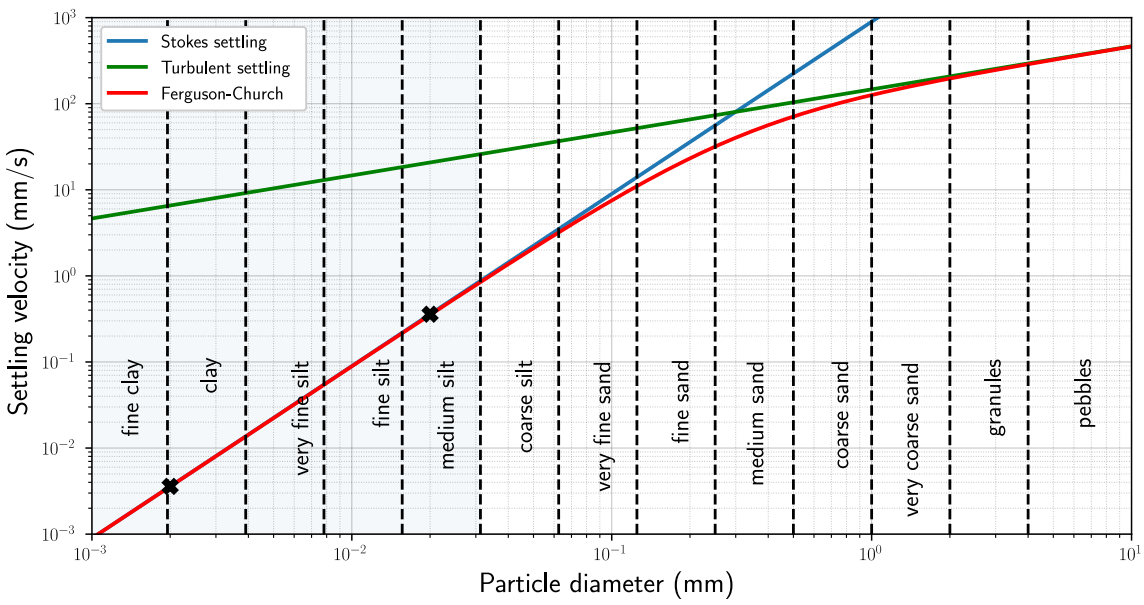


Figure 1.9 – Settling velocity with respect to grain grain size. The blue zone corresponds to the dilute suspensions considered here. Figure adapted from [Sylvester 2013].

The only considered particles being the ones with negligible inertia (clay and fine silts), the settling velocity of a single particle will always be determined by Stokes law (1.45). The small diffusion coefficient κ_c associated to the particles accounts for Brownian motion or the mixing that would occur in real polydisperse suspensions. In practice, for grain sizes between 1 and

$20\mu m$, the particle diffusivity is mainly induced by the velocity fluctuation due to the presence of neighboring particles [Yu et al. 2014]. Depending on the volume fraction of particles ϕ, the neighboring particles also have an influence on effective viscosity and settling velocity. A semi-empirical formulation based on the particle diameter d and particles Stokes velocity V_{st} has been proposed by [Segre et al. 2001]:

$$\nu_c(\phi) = \nu_0(1 - \phi/0.71)^{-2} \tag{1.47a}$$

$$V_c(\phi) = V_{st}(1 - \phi)^5 \tag{1.47b}$$

$$\kappa_c(\phi) = K(\phi)V_{st}r \tag{1.47c}$$

$$(\phi) = 11.4\frac{(1-\phi)^2(1-\phi/0.71)^2}{\sqrt{(1+4\phi+4\phi^2-4\phi^3+\phi^4)}} \tag{1.47d}$$

where $r = d/2$ is the particle radius and ν_c and V_c are respectively the effective viscosity of the sediment-laden mixture and the effective settling velocity of the particles.

As shown on figure 1.10, for dilute suspensions under $\phi = 2\%$, the resulting values stay really close to the particle properties obtained at infinite dilution.

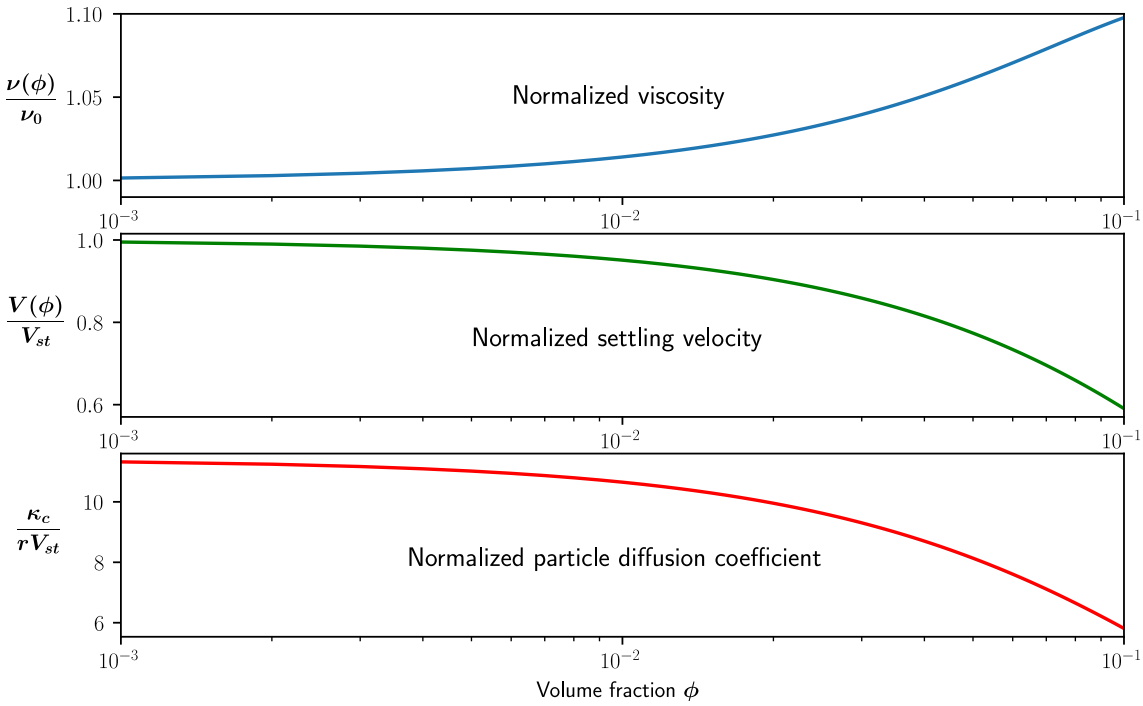


Figure 1.10 – Semi-empirical model [Segre et al. 2001] for ϕ between 0.1 and 10%.

The average yearly sediment concentration of dirty rivers ranges from 10 to $40kg/m^3$ [Mulder et al. 1995]. For quartz ($\rho_c \simeq 2650kg/m^3$) this corresponds roughly to volume fractions between 0.4% and 1.5%. When a river outflow is loaded with a large amount of suspended sediment (more than $40kg/m^3$) the density of the mixture becomes greater than the one of seawater and the flow becomes hyperpycnal. The diffusivity of salt in sea water has already been estimated in the last subsection, leading to $\kappa_s = 1.286 \times 10^{-9}m^2/s$. If we estimate κ_c

from equations (1.47) for a given volume fraction and particle size, we can also compute the diffusivity ratio $\tau = \kappa_s/\kappa_c$. This is illustrated on figure 1.11.

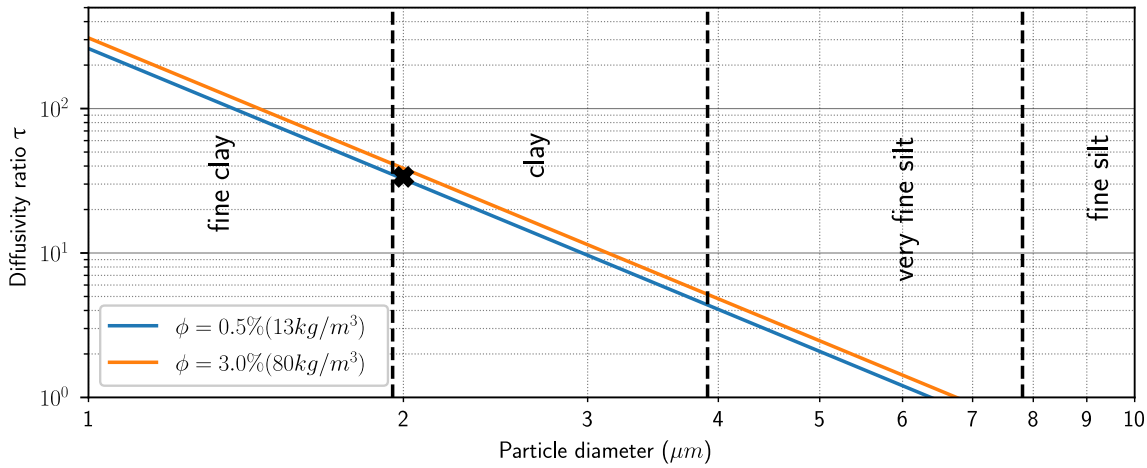


Figure 1.11 – Estimation of the diffusivity ratio between salinity and particles.

From figure 1.11 we can estimate that $\tau \simeq 25$ for a particle diameter of $2\mu m$. This is effectively the usual values that we can find in the literature [Yu et al. 2014][Burns et al. 2015]. The dimensionless parameters corresponding to grain sizes of $2\mu m$ and volume fractions of roughly 1% are usually the following:

- **Dimensionless settling velocity:** $V_p \simeq 0.04$
- **Schmidt number:** $S_c = \dfrac{\nu_0}{\kappa_s} \simeq 700$
- **Diffusivity ratio:** $\tau = \dfrac{\kappa_s}{\kappa_c} \simeq 25$
- **Initial stability ratio:** $R_\rho = \dfrac{\alpha \Delta S}{\beta \Delta C} \simeq 2$

The value $R_\rho = 2$ corresponds to a mass loading of roughly $20kg/m^3$ of sediments and a salinity of 3%. The Schmidt number between sediments and water can be computed as $\nu_0/\kappa_c = \tau S_c \simeq 17500$. Because of the high Schmidt numbers involved and associated Batchelor scales for salinity and particle concentration, numerical simulations with excessively large values of S_c or τ cannot be performed (see section 1.1.10). With $\tau = 25$, simulations up to $S_c = 70.0$ in 2D and $S_c = 7.0$ in 3D have been performed in [Burns et al. 2015]. Accessing higher Schmidt numbers is the current challenge to be able to fully understand sediment fingering naturally occurring near sediment-laden riverine outflows. The Batchelor scale of fine particles being around $\eta_B = \eta_K/\sqrt{S_c} \simeq \eta_K/132$, a multilevel approach with a grid ratio of 132 in each direction could be considered. Even for a two-dimensional flow, this will require important computational ressources and distributed high performance computing is here mandatory. This work aims to achieve high Schmidt number sediment-laden flow simulations by using adapted numerical methods in a high performance computing context.

1.4 High Performance Computing

High performance computing (HPC) is a field of applied mathematics that consists in using supercomputers for applications that generally target scientific computing. The goal is to take full advantage of all the available compute resources to solve a given numerical problem. When high performance computing is used to solve multiple independent problems on a set of resources of a computing machine, the problem is said to be embarrassingly parallel. In this case little to no effort is needed to separate the problem into a number of parallel tasks. When a set of compute resources is used to solve a single problem, the numerical method employed has to be distributed on the computational elements in a collaborative manner. This usually implies communication of intermediate results between the different compute ressources. The implementation is here less straightforward because the problem has to be decomposed into parallel tasks. This often requires to rethink the underlying algorithms and to find a compromise between calculations and communications. As communication between distant compute ressources is usually a costly operation, it is not uncommon to perform redundant calculations to be able to reduce the volume of communications.

1.4.1 Hierarchy of a computing machine

A computing machine is made up of different hierarchical level of resources. It is composed of interconnected compute nodes that generally comprises a hierarchy of memory (multiple levels of caches, `RAM`, local disks), processors and possibly accelerators (coprocessors or graphics cards). Including multiple cores on a single chip has become the dominant mechanism for scaling processor performance. Modern supercomputers are thus hierarchical and the hierarchy depth tends to grow [Smith et al. 1990][Fu et al. 2016]. Structural hierarchy (core, socket, node, rack, system) implies significant differences in communication time [Blaauw et al. 1997]. Memory hierarchy also induces differences in access time: the larger is the size of the level, the slower is the access and the data movement overhead become the most significant factor of inefficiency.

A given compute node containing multiple processors usually exhibit one of the following memory architecture [Lameter 2013]:

- **Uniform memory access:** UMA is a shared memory architecture used in parallel computers where all the processors share the physical memory uniformly. In an UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data.
- **Non-uniform memory access:** On NUMA architectures, memory access time depends on the memory location relative to the processor. A processor can access its own local memory faster than non-local memory.

A compute node usually contain one or more sockets (NUMA nodes), that each can contain one or more physical `CPUs` (corresponding to the physical sockets on the motherboard). A

compute cluster contain one or more compute nodes and effectively constitute a distributed memory architecture. Most usual cluster configurations ranges from one to eight sockets per compute node, each containing a single physical multi-core `CPU`.

1.4.2 Evaluation of performance

Performance of supercomputers is usually evaluated in terms of floating point operations per second (`FLOPS` or `FLOP/s`) because most scientific computations require floating-point calculations. Although the notion of single- and double-precision is architecture specific, we will here always refer to 32 bits floating point arithmetic (`FP32`) as single-precision and 64 bits floating point arithmetic (`FP64`) as double precision. The `TOP500` project was started in 1993 and ranks the 500 most powerful supercomputer systems in the world [Meuer et al. 1993]. Since 1993, performance of the number one ranked position has grown in accordance with the revised Moore's law, doubling roughly every 18 months [Moore 1975]. As of June 2019, `Summit` is the fastest supercomputer with theoretical peak computing performance of `200.8 PFLOPS` of double-precision floating point arithmetic. As a comparison, this is over 1.53×10^6 times faster than the `Connection Machine CM-5/1024`, which was the fastest system in November 1993 with `131.0 GFLOPS` theoretical peak computing performance. This represents a performance doubling every 15 months. Even tough, for the first time, all top 500 systems deliver a petaflop or more on the High Performance Linpack (HPL) benchmark [Dongarra et al. 2003], Moore's law is slowing down as we get closer and closer to physical limitations of current lithography processes [Waldrop 2016].

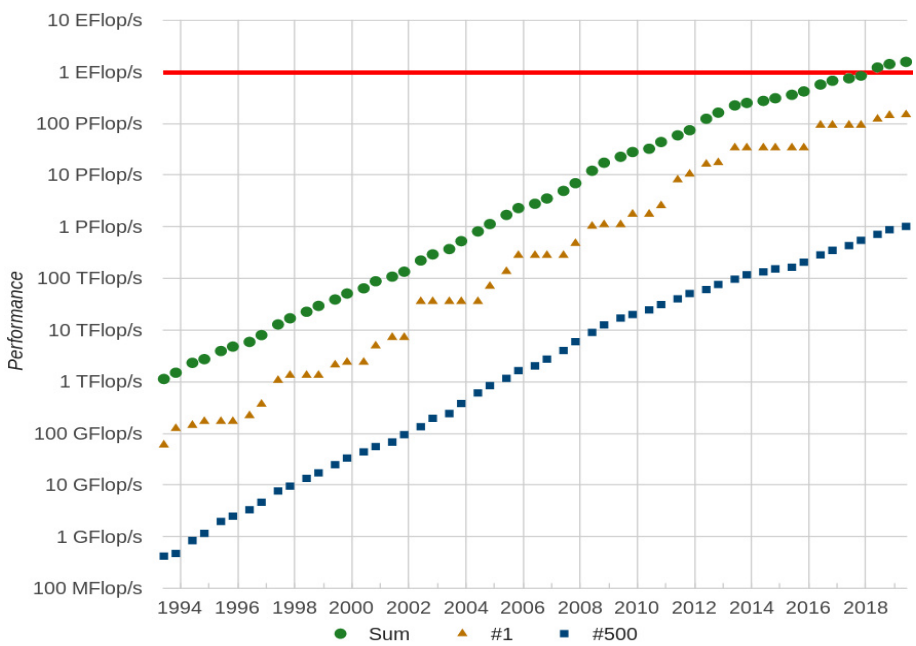


Figure 1.12 – Exponential growth of supercomputing power - `TOP500` list

HPL is a simple benchmark that factors and solves a large dense system of linear equations using Gaussian elimination with partial pivoting. Figure 1.12 shows the evolution of su-

percomputing power as achieved on the HPL benchmark (which is less then the theoretical peak compute). Currently the combined power of all top 500 supercomputers exceeds the exaflop, or one quintillion floating point operations per second. Exascale computing refers to computing systems capable of at least `1 EFLOPS`, and the first systems to break the exascale compute wall are expected in the upcoming years [Geller 2011]. When HPL gained prominence as a performance metric in the early nineties, vendors logically pursued architectural designs that would increase HPL performance. As there was a strong correlation between benched performance and general application performance, this in turn improved real application performance. While microprocessors have increased in speed at a rate of 60%/year in the last decades, access times to memory have only been improving at a rate of less than 10%/year [Carvalho 2002]. Because of the exponential growing disparity between compute and memory performance the HPL metric now gives a skewed picture relative to application performance where the computation-to-data-access ratios (`FLOP/B`) are low. The HPL benchmark is only relevant for high computation-to-data-access ratios as obtained for dense matrix-matrix multiplications [Dongarra et al. 2013]. Since November 2017, a new benchmark has been introduced in the `TOP500` project to evaluate computational and data access patterns that more closely match a broad set of scientific applications, leading to an alternative ranking of the `TOP500` supercomputers.

This new benchmark, named High Performance Conjugate Gradients (HPCG), has been designed to put more stress on memory and interconnect by performing sparse matrix operations. This benchmark is especially designed for HPC applications that require higher memory bandwidth and lower memory latencies. It aims to deter hardware vendors from making design choices that would be detrimental for a wide range of applications, but beneficial for the HPL benchmark. Such applications are usually memory-bound on current architectures, and do not benefit at all from additional floating point compute units. On the contrary, when a given algorithm is limited by computing power instead of memory bandwidth, it is said to be compute-bound. Because HPCG is memory-bound, this benchmark generally achieves only a tiny fraction of the peak `FLOPS` of a given supercomputer. As an example, the `Summit` supercomputer only achieves `2.93 PFLOPS` on HPCG versus `148.6 PFLOPS` on the HPL benchmark. For a given algorithm, the computation-to-data-access ratio in `FLOP/B` is defined as the arithmetic intensity. Particle methods and dense linear algebra (HPL) perform the order of 10 floating point operations per byte loaded or stored in memory. Those algorithms are typically compute-bound on current architectures. Sparse linear algebra (HPCG) and stencil computations perform less than `1 FLOP/B` and are usually memory-bound. Spectral methods and Fast Fourier Transforms (FFT) have intermediate arithmetic intensities and may be either memory- or compute-bound for a given architecture [Williams et al. 2009]. An associated visual performance model is described in section 1.4.7.

Last but not least, the performance can also be evaluated in terms of power efficiency. As power consumption of supercomputers increases, energy efficiency will move from desirable to mandatory in order to push beyond the exascale limit [Hemmert 2010]. The `GREEN500` is a third list that classifies supercomputers in terms of achieved floating point operations per second per watt (`FLOPS/W`). Here `Summit` is ranked second with `14.72 GFLOPS/W`, for a total consumption of roughly `10 MW`.

1.4.3 Accelerators and coprocessors

Since 2010, high performance computing system architectures have shifted from the traditional clusters of homogeneous nodes to clusters of heterogeneous nodes with accelerators [Kindratenko et al. 2011]. Hardware acceleration consists in the use of computing hardware specially designed to perform some operations more efficiently than in software running on general-purpose `CPUs`. Accelerators include many integrated core architectures (`MICs`), graphics processing units (`GPUs`), digital signal processors (`DSPs`), application-specific instruction-set processors (`ASIPs`), field-programmable gate arrays (`FPGAs`) and other processors or hardware accelerators. In the realm of high performance computing, the hardware accelerators have to be reconfigurable and the current tendency is clearly in favour of dedicated `GPUs`, followed to a lesser extent by `MICs` and a slow but increasing adoption of `FPGAs` [Weber et al. 2010]. The `TOP500` of June 2019 lists 134 supercomputers containing accelerators. Those supercomputers represent roughly 40% of the total computing power of the 500 machines (in terms of HPL performance) as exposed on figure 1.13.

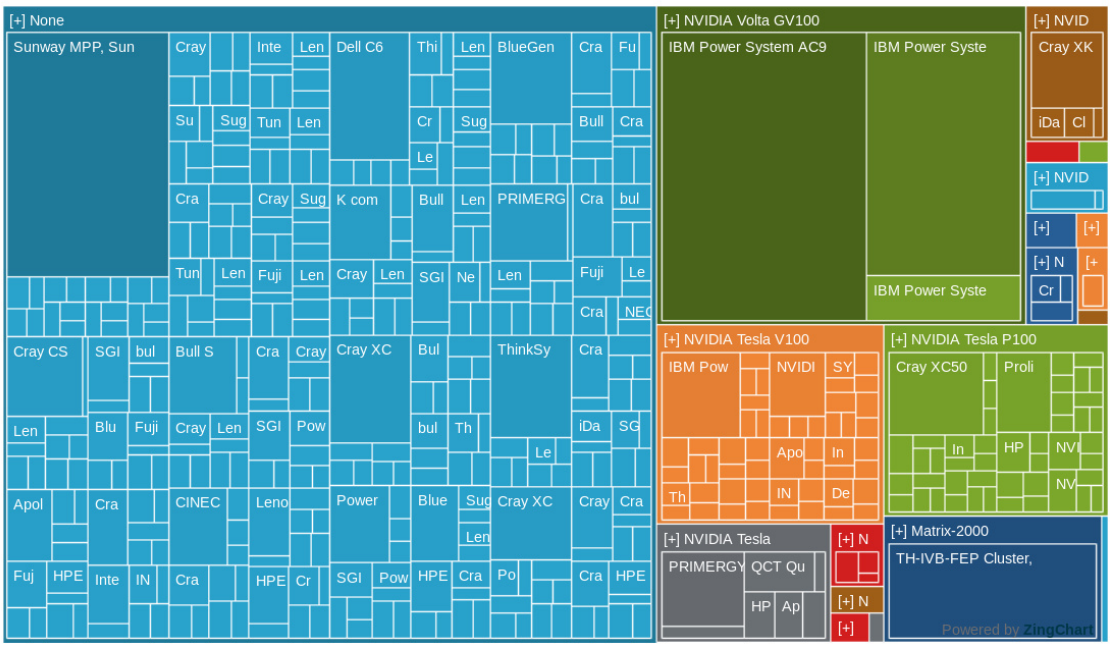


Figure 1.13 – Treemap representing `TOP500` coprocessor share. Each rectangle represent a specific accelerator, and `None` means that no accelerator is present (leftmost part of the plot in blue). Each subrectangle represent supercomputer of the `TOP500` list. The area of the rectangles is proportional to the benchmarked HPL performance on each supercomputer.

Accelerator architectures tend to be either heavily specialized or very generic and both of these models present real challenges when it comes to programming. General-purpose processing on graphics processing units (GPGPU) is the use of a `GPU` as a coprocessor for general-purpose computing [Luebke et al. 2006]. The `GPU` accelerates applications running on the `CPU` by

offloading some of the compute-intensive portions of the code while the rest of the application still runs on the CPU. The share of coprocessors on the 134 supercomputers containing accelerators is further described in figure 1.14. Here the advantage is clearly to Nvidia GPUs, as most of the accelerators present in the TOP500 are last generation Volta GPUs (Volta GV100, Tesla V100, Tesla P100 and variants). Out of those 134 machines, 62 contain Nvidia Volta GPU architectures (46.6%), 50 contain NVidia Pascal GPU architectures (37.6%), 12 contain NVidia Kepler GPU architectures (9%) and 5 contain Intel Xeon Phi MICs (3.8%). The remaining machines contain either custom accelerators or NVidia Fermi GPUs.

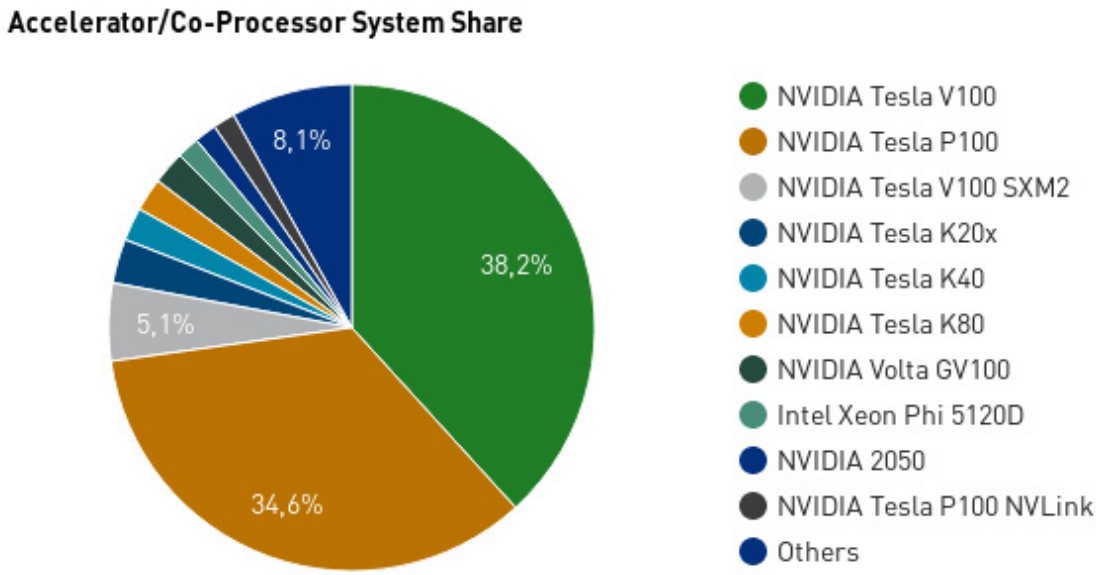


Figure 1.14 – Coprocessor share in the supercomputers containing accelerators as of June 2019. Data represents the ratio of obtained HPL performance with respect to the 134 machines having accelerator support, not the ratio of accelerator computing capabilities.

GPU accelerators are great for providing high performance in terms of floating point operations per second (FLOPS), the performance that is typically measured by the HPL benchmark [Nickolls et al. 2010]. Server-grade GPUs contain large ECC memory (up to 32GB VRAM) along with high performance for double-precision arithmetic. Consumer-grade GPUs are more focused on graphical tasks that generally require only single-precision arithmetic without error control [Arora 2012]. While the difference in performance between single precision and double precision arithmetic is generally 2 for server-grade GPUs, and it can be as high as 32 for consumer-grade GPUs. When their compute capabilities can be fully exploited, GPUs are advantageous both in terms of price ($/TFLOP) and energy consumption (GFLOPS/W) when compared to traditional CPUs. This is shown on figures 1.15 and 1.16. Those figures have been adapted from [Rupp 2013]. The 27648 Nvidia Volta V100 GPUs present on the current top one supercomputer Summit can deliver up to 215.7 PFLOPS of theoretical peak double-precision arithmetic for a power consumption of roughly 8.3 MW, ranking the machine to the second place of the GREEN500 list.

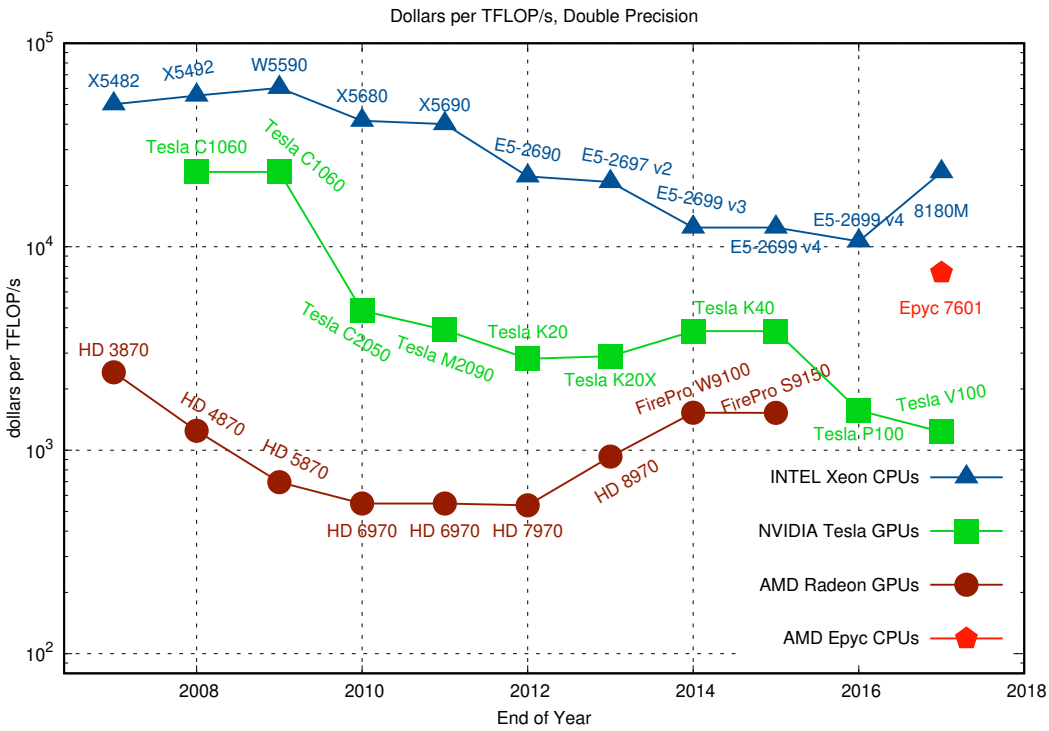


Figure 1.15 – Evolution of the cost of 1 TFLOPS of double-precision arithmetic

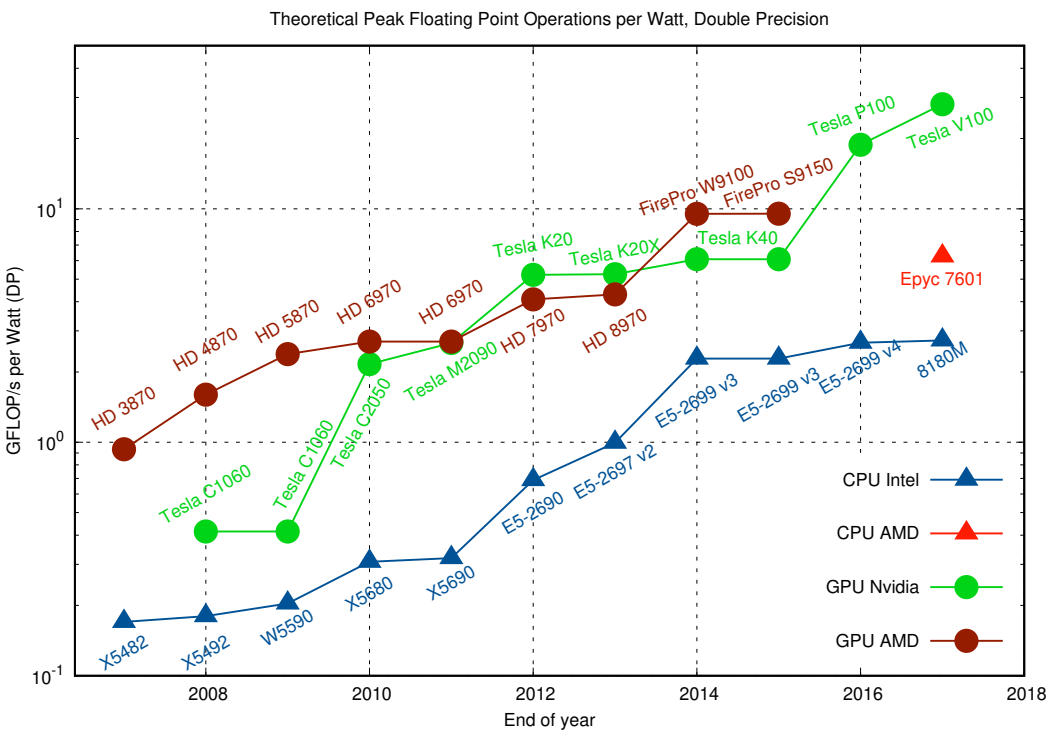


Figure 1.16 – Evolution of energy consumption in terms of GFLOPS/W

1.4.4 Parallel programming models

Due to thermal and energy constraints, the trend in processor and accelerator development has been towards an ever-increasing number of cores with reduced frequency. The exponential growth in the number of cores is expected to lead to consumer grade `CPUs` containing hundreds of cores in the near future [McCool 2008]. Another way to get extend the computing power of a supercomputer is to simply add more compute nodes. As the number of processors and compute nodes increase, efficient interprocessor communication and synchronization on a supercomputer becomes a challenge. In the nineties, two parallel programming open standards grew to dominate the parallel computing landscape:

- **Message Passing Interface:** `MPI` is a specification for a standard library for message passing that was defined by the `MPI` Forum, a group of parallel computer vendors, library writers, and applications specialists. MPI includes point-to-point and collective internode communications routines through communicator abstractions [Barker 2015]. The third version of the standard (`MPI-3`) introduced remote memory accesses [Hoefler et al. 2015]. Because it is a standard, multiple open-source implementations of `MPI` have been developed such as `MPICH` [Walker et al. 1996] and `OpenMPI` [Gabriel et al. 2004]. Implementations optimized for vendor-specific hardware are also available directly from hardware vendors.
- **Open Multiprocessing:** `OpenMP` is an implementation of multithreading. Once spawned, threads run concurrently, with the runtime environment allocating threads to different physical `CPU` cores. It is an industry-standard API for shared-memory programming [Dagum et al. 1998] targeting `C`, `C++` and `Fortran` and consists in a set of compiler directives, library routines, and environment variables that influence runtime behavior. The standard require specific compiler support [Novillo 2006] and as for `MPI`, both open-source and vendor-specific implementations exist. As for `OpenMP 4.1`, the standard began to introduce accelerator support [Antao et al. 2016].

Those two standards still require an in-depth understanding of computing ressource hierarchies. Multicore architectures with `OpenMP` requires an explicit management of the memory hierarchy while distributed computing with `MPI` requires explicit communications between the different processes that are mapped to the physical compute nodes. As `MPI` process placement can play a significant role concerning communication performance [Rashti et al. 2011], tools such as `TopoMatch`, have been developed in order optimize the placement of processes on `NUMA` compute nodes with respect to interprocess data exchange [Jeannot et al. 2010]. Concerning OpenMP, thread and memory placement can be tweaked by using the `numactl` utility [Kleen 2005]. While those two standards still plays an important role in high performance computing, the learning curve can be steep for some programmers. A survey of programming languages that try to hide the complexity of multicore architectures is available in [Kim et al. 2009]

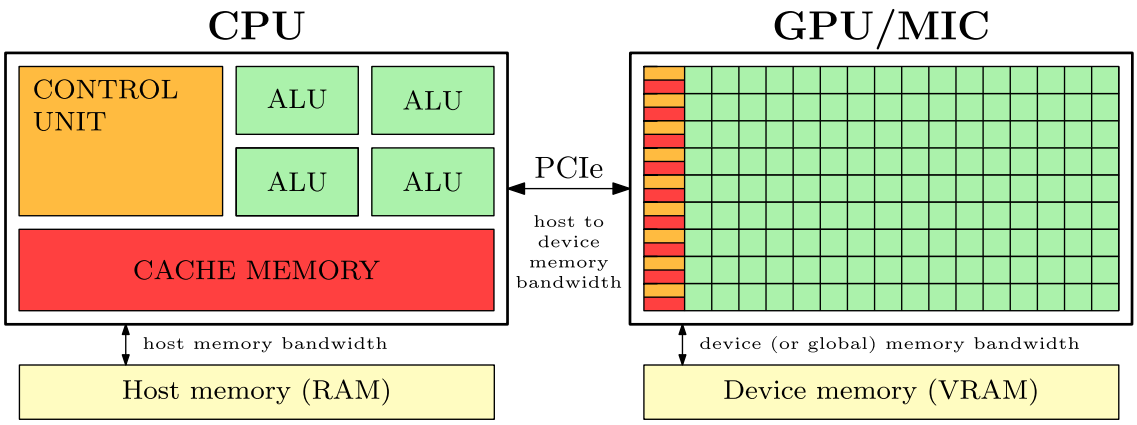
An application built with the hybrid model of parallel programming can run on a cluster using both `OpenMP` and `MPI`. In this case `OpenMP` is used for parallelism within a compute

node (or within a NUMA node) while MPI is used for internode communication. OpenMP can be replaced by other multithreading implementations such as POSIX threads [Barney 2009] and OpenCL. Since 2006, the two industry standards for GPU programming are CUDA [Luebke 2008] and OpenCL [Stone et al. 2010]. While the former has been designed only for Nvidia GPUs, the latter can drive a wide range of CPUs and accelerators. In this work we will focus on hybrid MPI-OpenCL programming.

1.4.5 The OpenCL standard

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators [Stone et al. 2010]. Device vendors usually propose their own implementation of the OpenCL standard, optimized to drive their own devices (AMD, Nvidia, Intel, IBM, ...) but open source implementations of the standard also exist. The Portable Computing Language (POCL) is a portable open source implementation of the OpenCL 1.2 standard based on LLVM [Jääskeläinen et al. 2015] on CPU devices. It also proposes an experimental CUDA backend targeting Nvidia GPUs by directly generating PTX pseudo-assembly from OpenCL [Compute 2010]. Each of those implementations defines what we usually call an OpenCL platform. Each compute node can expose multiple platforms each supporting specific devices through vendor installable client drivers [Trevett 2012]. A key feature of OpenCL is portability, via its abstracted memory and execution model. The ability to directly use hardware-specific features can be provided through vendor-specific extensions. However, performance is not necessarily portable across OpenCL platforms and a given OpenCL code has to be tested and tuned accordingly for each different target device.

The abstract memory model assumes that each OpenCL device can store its associated memory buffers in device memory banks that may differ from host memory (RAM). OpenCL CPU devices can store and interact directly with RAM, and device memory buffers can be seen as host ones through memory mappings. Dedicated GPU devices and other PCIe extension cards usually provide their own physical embedded memory (VRAM).



The OpenCL memory model defines the behavior and hierarchy of memory that can be used by OpenCL applications. It is up to individual vendors to define how the memory model maps to specific hardware. The host memory is defined as the region of system memory that is directly and only accessible from the host processor (RAM). The global memory is indirectly accessible from the host and directly accessible from the device (VRAM for GPUs, RAM for CPUs). Constant memory can be indirectly written by the host with read-only access on the device. Local memory is a region of memory that is local to a work-group and private memory is private to an individual work-item (registers).

A kernel is a function executed on an OpenCL device. The threads are organized hierarchically into one-, two- or three-dimensional grids each composed of equally shaped, one-, two- or three-dimensional thread blocks called work-groups. Depending on the work-group size, each work-group contains a certain amount of individual threads called work-items. Each kernel is executed asynchronously with specific work-space size and work-group size, defined by three-dimensional vectors global_work_size (work-space shape) and local_work_size (work-group shape). Multiple kernels can be enqueued in different command queues. The synchronization between queues and the host CPU is event based. An individual work-item can load and store data in the global memory. Work-items of the same work-group can collaborate by using the shared memory. Work-items cannot directly access to host processor memory, using CPU data thus requires host-to-device copy prior to the execution of the kernel.

1.4.6 Target OpenCL devices

In this work we will focus on performance obtained on Intel and Nvidia OpenCL platforms on the four following compute devices:

- Server grade dual socket Intel Xeon E5-2695 v4 CPUs (36 physical cores, 128GB RAM)
- Server grade GPU: Nvidia Tesla V100 (5120 CUDA cores, 32GB VRAM)
- Consumer grade CPU: Intel Core i7-9700K CPU (8 physical cores, 16GB RAM)
- Consumer grade GPU: Nvidia GeForce RTX 2080 Ti (4352 CUDA cores, 11GB VRAM)

Table 1.1 and 1.2 contain theoretical and achieved performance characteristics of the four considered benchmark configurations.

The speed of the bus interconnect usually constitute the bottleneck for host-device interactions but host-to-host and device-to-device memory bandwidth are also of importance in the design of OpenCL based numerical codes. Actual generation of PCIe buses (PCIe3.0 16x) provides 16 GB/s of theoretical bidirectional memory bandwidth (host-to-device and device-to-host) for dedicated accelerators. This may be slow compared to the device-to-device memory bandwidth, also called global memory bandwidth, provided by the two considered GPUs which is of the order of 1TB/s. When the data can entirely fit in device memory, a given kernel is either compute or memory bound depending on the device global memory bandwidth and compute capabilities for the required type of operations (single or double precision floating point operations, integer operations, ...).

This can be summed up into the designed arithmetic intensity of a given device which is given in operations per byte of memory traffic (OP/B). It can be seen in table 1.3 that the average device will require fifty operations per element loaded from memory in order not to be memory bound.

Device	**Type**	**Transistors**	**Platform**	**BDW** (GB/s)	**FP32** (GFLOPS)	**FP64** (GFLOPS)
i7-9700K	CPU	2.9B	Intel 18.1.0	41.6 (21.3)	588.0	294.0
E5-2695 v4	CPU	7.2B	Intel 18.1.0	76.8 (68.3)	745.85	372.9
RTX 2080Ti	GPU	18.6B	Cuda 10.1.120	616	11750 (16319)	367 (510)
V100-SXM2	GPU	21.1B	Cuda 10.0.141	900	14899 (17671)	7450 (8836)

Table 1.1 – Characteristics and compute capabilities of the considered devices: BDW corresponds to the maximal theoretical device global memory bandwidth (for CPUs , this corresponds to maximum memory frequency and all memory channels in use). For CPU devices, the second value between parenthesis corresponds to the maximal theoretical value enabled by actual memory configuration. FP32 and FP64 correspond to theoretical single and double precision peak floating point operations per second at device base clock. For GPU devices, the second value between parenthesis correspond to same values at maximal boost clock (this provides a performance boost of +38.9% for the 2080Ti and +18.6% for the V100 under sufficient cooling). CPU characteristics are estimated from Intel compliance metrics (CTP).

Device	#	**H2D** (GB/s)	**D2H** (GB/s)	**BDW** (GB/s)	**FP32** (GFLOPS)	**FP64** (GFLOPS)
i7-9700K	1	19.4	8.9	18.1 (85.0%)	514.8 (87.6%)	260.1 (88.5%)
E5-2695 v4	2	13.5	6.1	**67.4 (49.4%)**	1434.1 (96.1%)	713.1 (95.8%)
RTX 2080Ti	1	8.98	6.1	496.13 (80.5%)	14508.6 (88.9%)	451.4 (88.5%)
V100-SXM2	1	7.08	4.9	807.6 (89.7%)	15676.0 (88.7%)	7854.6 (88.9%)

Table 1.2 – Peak performance metrics obtained for the four device configurations: H2D corresponds to host-to-device memory bandwidth, D2H to device-to-host memory bandwidth, BDW to device global memory bandwidth, FP32 and FP64 to floating point operations per second. FLOPS (floating point operations per seconds) are taken as the best result achieved by clpeak [Bhat 2017] for all considered vector types. The results are also given in percentage of theoretical peak values at actual memory bandwidth and max. boost frequency.

	Operations per byte (OP/B)		Operations per element (OP/E)	
Device	**FP32** (FLOP/B)	**FP64** (FLOP/B)	**FP32** (FLOP/E)	**FP64** (FLOP/E)
i7-9700K	14.1 (27.6)	7.1 (13.8)	56.5 (110.4)	56.5 (110.4)
E5-2695 v4	9.7 (10.9)	4.9 (5.5)	38.8 (43.7)	38.8 (43.7)
RTX 2080Ti	19.1 (26.5)	**0.6 (0.8)**	76.3 (106.0)	**4.8 (6.6)**
V100-SXM2	16.6 (19.6)	8.3 (9.8)	66.2 (78.5)	66.2 (78.5)

Table 1.3 – Designed arithmetic intensity for usual floating point operations: Arithmetic intensity represent the minimal number of operations to execute per byte loaded in order to be compute bound. Data corresponds to the worst case scenario where we consider only one memory transaction of the given type. The numbers between parenthesis reflect more realistic values for the considered OpenCL platforms (using actual bandwidth and clock).

Double precision operations on consumer grade GPUs constitute an exception to this rule because of they target graphical tasks that mostly require single-precision operations. In practice because of the small number of operations per byte required for most of the numerical kernels we will develop in this work, the situation is likely to be memory bound. As an example, explicit finite differences based solvers use stencils operations while implicit finite differences solvers use sparse algebra. Both of those methods have very low arithmetic intensity. The performance of memory-bound kernels can quickly drop when the full potential of the memory controller is not used. How to correctly access memory is architecture dependent but most of accelerators use vector operations in a SIMD fashion [Duncan 1990]. It usually entails to access contiguous memory with the good granularity and alignment. Grouping of work-items into work-groups is not only relevant to computation, but also to global memory accesses. The device coalesces global memory loads and stores issued by neighboring work-items of a work-group into as few transactions as possible to minimize required memory bandwidth. For strided global memory access, the effective bandwidth is poor regardless of architecture and declines with increasing stride (distance between two accessed elements).

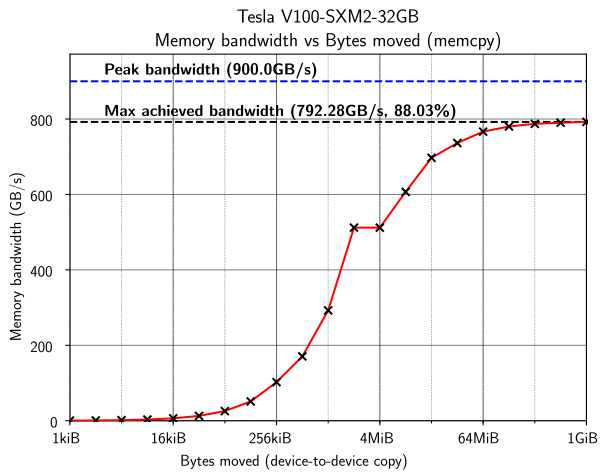


Figure 1.17 – Achieved memory bandwidth vs. memory transaction size
Each result (black cross) corresponds to the median bandwidth obtained for 1024 device-to-device copies performed with the builtin OpenCL copy kernel on the Tesla V100-SXM2 GPU.

Global memory accesses on cache miss have a latency of hundreds device cycles (of the order of $\simeq$ 100ns). In order to achieve the full device bandwidth, this latency needs to be hidden using the following strategies:

- Perform contiguous memory accesses: adapt the data structures and reformulate associated algorithm.
- Perform independent loads and stores from the same work-item: increase problem size or increase work-item workload by playing on the global work-space size (global_size).
- Perform memory transactions from increased number of work-items (local_size)
- Use larger word sizes by using vector transactions: use float4 instead of float when the device is designed to handle 16B per work-item per coalesced memory transaction.

All those strategies are so that there is enough memory transactions in flight to saturate the memory bus (the price of latency for small memory transaction is shown on figure 1.17). Instead of relying only on some compiler automatic vectorization process, the `OpenCL` standard provide vector types and vector intrinsics to manually vectorize kernels. Those vector intrinsics have a great impact on achieved memory bandwidth as shown on figure 1.18 for single precision floating point vectors.

Figure 1.19 clearly shows that for the `Tesla V100 GPU`, `16B` per work-item per memory transaction is optimal. Considering that all types perform equally well at given type size (which is surprisingly not the case here for `64B` transactions) one can deduce from figure 1.18 that the two `CPUs` prefer `64B` transactions and that the consumer grade `GPU` is designed for `32B` transactions. Those values do not match the `preferred` and `native` vector sizes provided by the `OpenCL` platforms and encourages microbenchmarking for the design of any kernel. A given hard-coded kernel will thus be portable across our four platforms but performance will not. The solution to this problem is to implement one kernel per vector type, and this can be done with a simple code generation technique associated to some kernel template. The best kernel can then be determined at runtime by self-tuning [Dolbeau et al. 2013]. In fact, kernel auto-tuning can be extended to determine other runtime parameters such as the local and global grid sizes that have an effect on device occupancy and work-item workload.

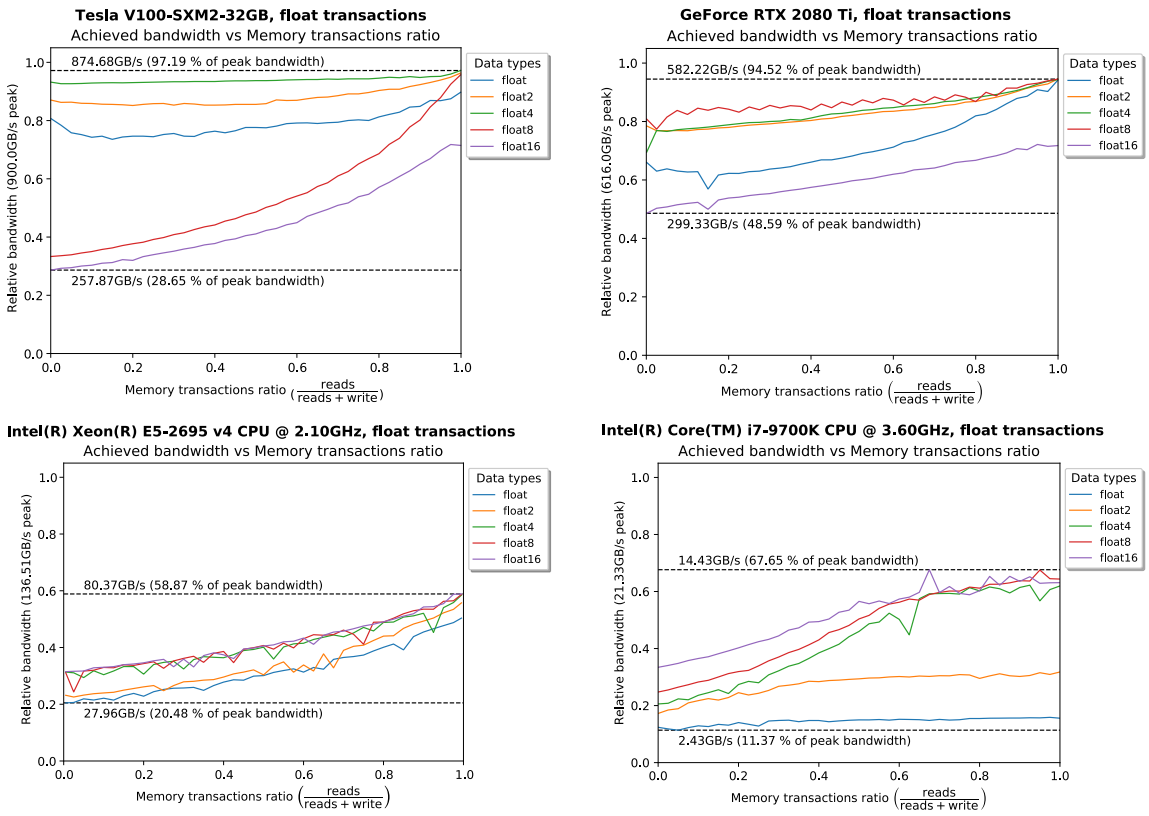


Figure 1.18 – Achieved memory bandwidth versus float vector type for all devices

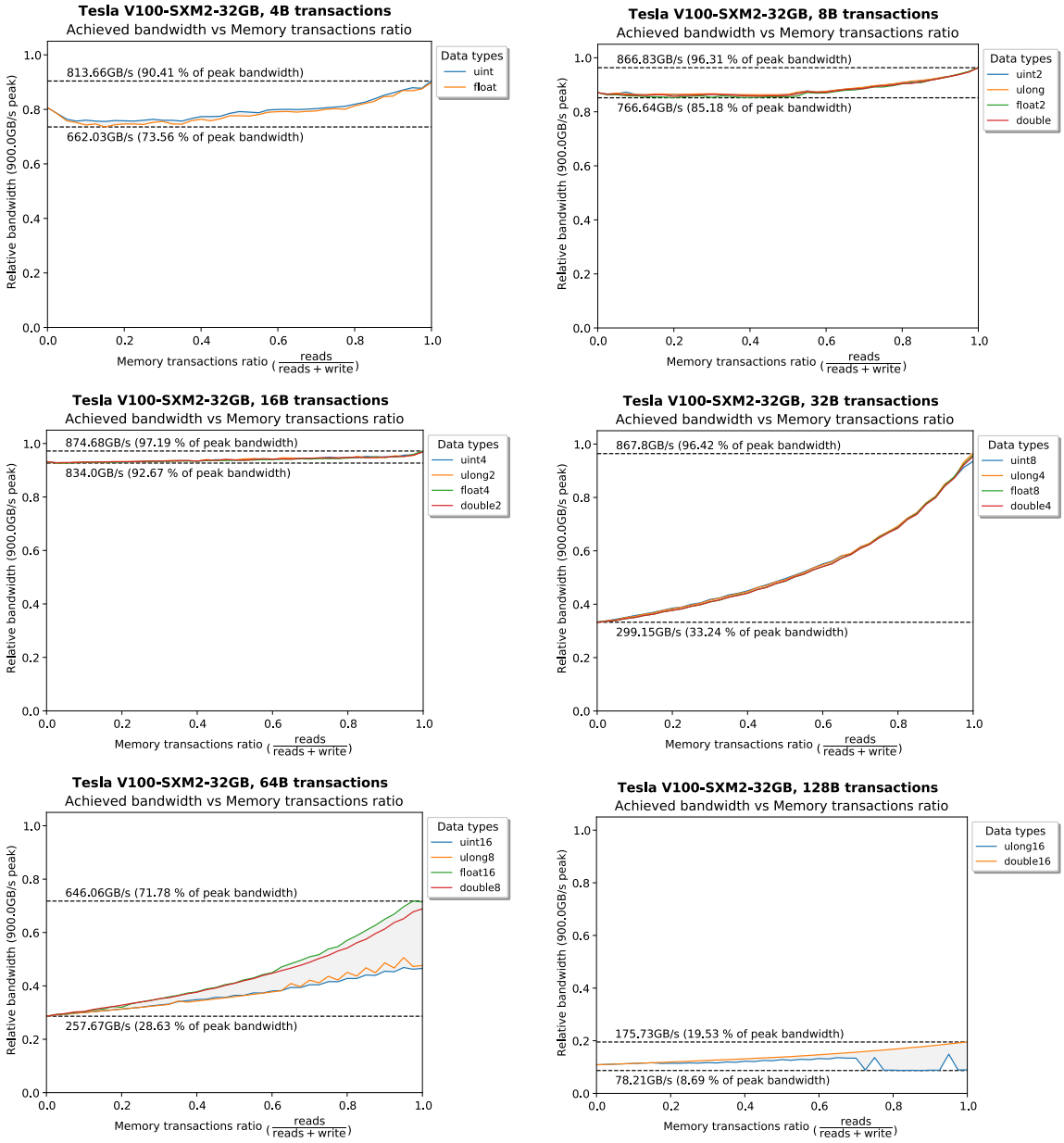


Figure 1.19 – Achieved memory bandwidth versus transaction granularity: Median bandwidth achieved for `1 GiB` buffers at `1 op/E` over 1024 runs per kernel. The memory transaction ratio is the amount of memory loads compared to the total number of memory transactions (loads + stores). A different kernel is generated for every data point.

1.4.7 The roofline model

When a kernel may be compute bound we can switch to performance model that also take into account the compute capabilities of the device. The roofline model is an intuitive visual performance model used to provide performance estimates of a given compute kernel

running on a given architectures by showing inherent hardware limitations [Williams et al. 2009]. The most basic roofline model can be visualized by plotting performance as a function of device peak performance Π and device peak bandwidth β (see table 1.2). For a given hardware, the best attainable performance is expressed by $P = min(\Pi, \beta I)$ where I is the arithmetic intensity in operations per byte. The resulting plot usually exhibits fractions of maximum attainable performance P with both axes in logarithmic scale. Within this model, the performance obtained for a given kernel can be plotted as a single point which coordinates represents the arithmetic intensity I of the considered algorithm (this value is fixed for a given algorithm and do not depend on kernel optimizations) and the obtained performance obtained from microbenchmarking (most of the time in operations per second). A kernel is said to be memory-bound when $I \leq \Pi/\beta$ and compute bound when $I \geq \Pi/\beta$ (see table 1.3).

A reference roofline for single and double precision floating point operations can be obtained with the `mixbench` benchmark [Konstantinidis et al. 2017]. It generates kernel of increasing arithmetic intensity (in `FLOP/B`) and run them on arrays with fixed number of elements (an element has a size `4B` for `float` and `8B` for `double`). The joint knowledge of the total number of elements, the arithmetic intensity and the result of the runtime is required to deduce the obtained performance in `FLOP/s`. The following figure shows the performance obtained with this benchmark on the two `GPUs` for single and double precision operations:

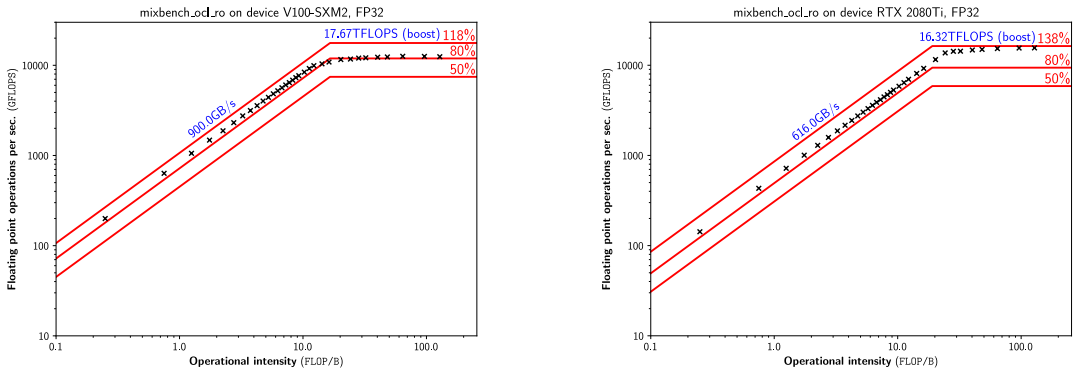


Figure 1.20 – Roofline of mixbench achieved single-precision performance

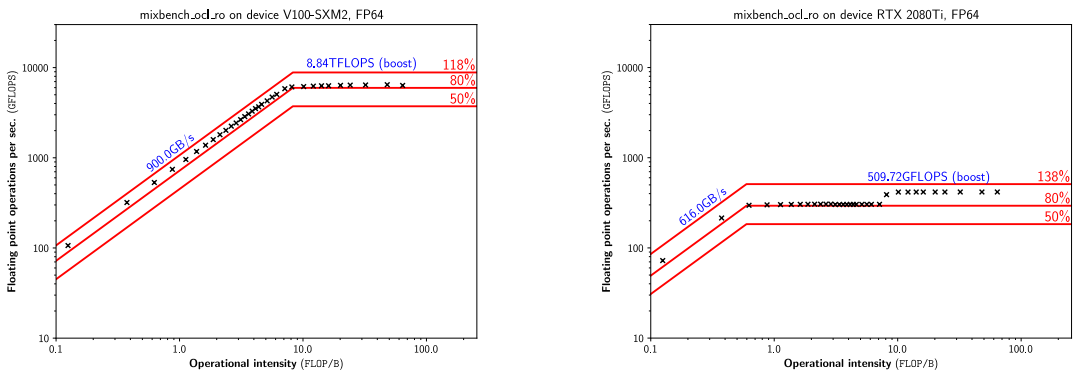


Figure 1.21 – Roofline of mixbench achieved double-precision performance

1.4.8 High performance fluid solvers

All fluid solvers are based on a numerical approximation of the variables and the equations both in terms of time and space. The common principle is to spatially discretize the field of computation in various elements: grid, mesh or particles. In any case, the solution is given on these discrete elements and an increase in the number of elements leads to a better spatial description of the solution. Timesteps are usually chosen to be the largest possible while ensuring the stability of the underlying numerical schemes. Most methods allow an estimation of the approximation errors in terms of spatial and temporal resolutions of the discretization. In practice, obtaining results on finer resolutions generally allows greater accuracy in the resolution of the considered problem, which contributes to a better understanding of the physical mechanisms involved. The consequence is that the computing resources necessary for a refinement of numerical solutions become important. Hence, flow simulation relies heavily on high performance computing to accurately simulate complex fluid problems [Tezduyar et al. 1996].

In the passed years, many high-performance numerical fluid-related solvers have been released and many other research-oriented closed-source solvers are still in active development:

- **Finite element based solvers:**
  - `FEniCS` is a `C++/Python` library to translate finite element variational formulations into efficient distributed finite element code [Logg et al. 2012]. It is based on `DOLPHIN`, an automatic finite element code generator [Logg et al. 2010]. This solver has been used to simulate turbulent flows on massively parallel architectures and scales up to 5000 cores [Hoffmann et al. 2015].
  - `Fluidity` is a general purpose, multiphase computational fluid dynamics code capable of numerically solving the Navier-Stokes equations on arbitrary unstructured finite element meshes. It is parallelised using `MPI` and is capable of scaling to many thousands of cores [Davies et al. 2011].
  - `Feel++` is another `C++` `MPI`-based finite element (and spectral element) solver based on domain decomposition [Prud'Homme et al. 2012]. It implements one-, two- and three-dimensional Galerkin methods and can be applied to fluid mechanics. It scales up to thousands of cores [Abdelhakim 2018].
- **Finite-volume based solvers:**
  - `OpenFOAM` is a `C++` toolbox for the development of custom numerical solvers targeting continuum mechanics problems and most prominently computational fluid dynamics [Chen et al. 2014]. Although the solver is `MPI`-based, multiple parts of the library have been accelerated by introducing `GPU` support [Malecha et al. 2011][He et al. 2015]. This solver has been used to solve the three-dimensional two-phase eulerian model for sediment transport [Chauchat et al. 2017].
  - `MRAG` is an hybrid `CPU-GPU` wavelet adaptive solver based on high-order finite volume scheme and `OpenCL`. It has been implemented to solve multiphase compressible flows [Rossinelli et al. 2011].

- Using an hybrid OpenMP-MPI NUMA-aware solver, [Hejazialhosseini et al. 2012] performed a three-dimensional simulation of shock-bubble interaction discretized on 200 billion computational elements ($9728 \times 6656 \times 3072$). Achieved performance was 30% of the total peak computing power of a cluster containing 47.000 cores. Using one MPI process per NUMA node and replacing OpenMP by Intel TBB improved the runtime [Willhalm et al. 2008].
- COOLFluid is an open computational platform for multi-physics simulation and research. It features a collection of parallel numerical solvers for unstructured meshes such as finite-differences and finite-volume based solvers for computational fluid dynamics problems [Lani et al. 2013]. Some parts of the solver are GPU-enabled [Lani et al. 2014] and the implicit multi-fluid finite volume solver has been successfully run on up to 60.000 cores.

- **Lagrangian particle methods:**
  - Many GPU enabled SPH implementation have been proposed [Amada et al. 2004][Hérault et al. 2010][Cercos-Pita 2015]. Distributed open source SPH solvers based on CUDA and OpenCL include:
    1. GPUSPH [Rustico et al. 2012] (MPI+CUDA),
    2. DualSPHysics [Valdez-Balderas et al. 2013] (MPI+CUDA)
    3. PySPH [Ramachandran 2016] (MPI+OpenCL)
  - GADGET-2 is a code for collisionless cosmological simulations using SPH. A parallel version has been designed to run with MPI and reached total particle numbers of more than 250 million [Springel 2005]. A third version of the solver is being written to prepare for the exascale era [Goz et al. 2017].
  - A new family of eulerian solvers based on lagrangian particles and finite-volume approximation (Lagrange-flux) is being developed as an alternative to Lagrange-remap schemes with better scalability in mind [De Vuyst 2016a][De Vuyst 2016b].
  - Partial GPU acceleration of eulerian-lagrangian particle-laden turbulent flow simulations has been performed by [Sweet et al. 2018].

- **Remeshed particle methods:**
  - PPM has been as the first highly efficient MPI parallel solver based on remeshed-particles [Sbalzarini et al. 2006]. It provides a general-purpose, physics-independent infrastructure for simulating systems using particle methods. Using PPM along with the vortex method, simulations of vortex rings with up to 6×10^9 particles were performed on 16.000 cores [Chatelain et al. 2008b]. The same method was then applied to numerical simulations of aircraft wakes [Chatelain et al. 2008a]
  - In the mean time, two-dimensional remeshed particle methods were ported to GPU by using CUDA and extended with vortex penalization techniques [Rossinelli et al. 2008] [Rossinelli et al. 2010]. Corresponding spectral fluid solver was implemented by using the CUDA FFT library (cuFFT).

- `SCALES`: This library proposes a `MPI`-based `Fortran` implementation of remeshed-particles methods. It has been successfully run up to 8192 cores with 89% scalability for grids of size 2048^3 [Lagaert et al. 2014].
- `HySoP`: In parallel an hybrid `MPI-OpenCL` code has been developed for the transport of a scalar in a turbulent flow [Etancelin et al. 2014]. The fluid solver was also FFT-based (`FFTW`) and solved on `CPUs` while the scalar, defined on a finer grid, was transported on `GPUs`.
- `MRAG-I2D`: This is an open source software framework for multiresolution of two-dimensional incompressible viscous flows on multicore architectures (`OpenMP`). The solver uses remeshed particles and high order interpolating wavelets [Rossinelli et al. 2015].

- **Spectral and pseudo-spectral methods:**
  - `Tarang` is a general-purpose pseudo-spectral parallel code developed for turbulence and instability studies. It has been validated up to grid of size 4096^3 and offers good weak and strong scaling up to several thousand processors. It features periodic as well as homogeneous boundary conditions [Verma et al. 2013].
  - `SpectralDNS` is a periodic FFT solver for turbulent flows that has been implemented by using only `Python` modules: `pyFFTW` and `MPI4py`. This solver scales well up to 3000 cores by using the hybrid `MPI-OpenMP` capabilities offered by `FFTW` [Mortensen et al. 2016].
  - `Dedalus` translates symbolic user-supplied partial differential equations into an efficient `MPI` distributed spectral solver. It supports Fourier, sine and cosine bases as well as polynomial bases (Chebyshev, Hermite, Laguerre) and scales up to thousand of cores [Burns et al. 2016].
  - `CaNS` is a `FFTW` based code for massively parallel numerical simulations of turbulent flows. It aims at solving any fluid flow of an incompressible, Newtonian fluid that can benefit from a `FFT`-based solver [Costa 2018]. It proposes the same boundary conditions as `Tarang` supplemented with general boundary conditions on the last axis and offers good strong scaling performances up to 10.000 cores.
  - `FluidSim` is a framework for studying fluid dynamics on periodic domains by using `Python` [Mohanan et al. 2018]. It is based on `FluidFFT` is a common API for FFT HPC libraries. This `Python` module allows to perform two- and three-dimensional FFTs with `FFTW` and `cuFFT`.
  - `Shenfun` is a high performance computing platform for solving PDEs by the spectral Galerkin method on multi-dimensional tensor product grids [Mortensen 2018]. It proposes a `Python` user interface similar to `FEniCS` and can solve periodic problems (Fourier) and problems where the last direction is inhomogeneous by using a mix of Chebyshev and Legendre bases. It is parallelised by using `mpi4py-fft` package and scales up to thousand of cores for periodic boundary conditions [Mortensen et al. 2019].

Most of the libraries presented here are available as open-source projects and propose a large set of numerical methods using many different spatial and temporal discretization techniques. Those numerical routines are often distributed up to thousands of CPU cores by using hybrid MPI-OpenMP or MPI-OpenCL programming models. More and more solvers are starting to integrate at least partial GPU support. This partial GPU support often comes from the drop-in replacement of underlying linear solvers (CUDA libraries cuSparse, cuBLAS and OpenCL equivalents clSparse, clBLAS) or other accelerated libraries (cuRAND, cuFFT or clRNG, clFFT). Distributed GPU computing is becoming more and more popular especially for particle based methods. For many applications GPU adoption is penalized by additional CPU-GPU memory transfers. As an example multi-GPU distributed implementations of spectral Fourier based solvers do not scale well because of the additional cost implied by host-device transfers [Wu et al. 2013].

The tendency is clearly to simplify the user interfaces from compiled Fortran or C++ configuration files to accessible scripting languages such as Python while relying on compiled code under the hood for performance. An ever increasing number of high performance libraries are made available through simple Python modules (pyFFTW, pyBLAS, pyMUMPS, ...). Because those wrappers provide a pythonic access to equivalent performances, it is now possible to implement performant distributed numerical solvers using directly Python as the core language (FluidSim, PySPH, HySoP), enabling faster development cycles [Wagner et al. 2017]. Writing multi-threaded compiled extensions to Python has never been easier than today with the advent of tools like Cython [Behnel et al. 2011], Numba [Lam et al. 2015] and Pythran [Guelton et al. 2015]. The same goes for GPU-programming with the PyOpenCL and PyCUDA Python modules provided by [Klöckner et al. 2012]. Finally the mpi4py module allows fast prototyping and development of MPI-based distributed solvers [Smith 2016].

Because the problems to be solved are not known in advance, some libraries use code generation techniques from user specified inputs such as symbolic equations (FEniCS, Dedalus). Code generation is also a common technique used for GPU-programming [Klöckner et al. 2012]. In this work we will be particularly interested in the development of numerical routines using finite-differences, remeshed-particles and spectral methods in an hybrid MPI-OpenCL context. Similar code generation techniques will be used to generate finite-difference based numerical routines from user-supplied symbolic expressions. A particular attention will be paid to OpenCL performance portability between different compute devices. The corresponding numerical methods are introduced in section 2 and are implemented within the HySoP library in section 3.

Conclusion

We first described the derivation of the Navier-Stokes equations along with their simplification to the incompressible case. The velocity-vorticity formulation has been derived from the associated velocity-pressure formulation. Remeshed particle methods are CFL free numerical schemes used to discretize the transport that fit well with the velocity-vorticity formulation of the incompressible Navier-Stokes equations. Those methods are particularly adapted for the transport of fields that diffuse slower than the vorticity due to viscous momentum diffusion. Those flows are more commonly known as high Schmidt number flows. In such a flow, the slowly diffusive agent will develop smaller physical scales, up to a ratio depending on the square root of the Schmidt number, when compared to the smallest velocity eddies.

Sediment dynamics is still poorly understood and multiple physical models with increasing complexity exist. Those models mostly depend on the mass loading and the physical properties of the said sediments in the carrier fluid. Monodisperse distribution of particles with a mass loading of less than 1% can be reduced to a dual-way coupling between the particle concentration and the fluid, in an eulerian-eulerian framework. When the fluid density is only slightly affected by the presence of these additional particles, the equations can be simplified by using the Boussinesq approximation. As small particles may diffuse slower than other fields such as temperature or salinity, a configuration where two such species are present can lead to double-diffusive instabilities. In the nature, this situation can happen near sediment-laden riverine outflows flowing into the ocean. Although double-diffusivity is known to enhance vertical fluxes, effective sedimentation velocity cannot be numerically investigated because of the high Schmidt numbers in play for fine particles.

From a single mathematical problem, different numerical methods can be considered. The choice of the numerical method strongly depend on the target problem and available computing architectures. Here the choice of a remeshed particle method is natural but not sufficient. The use of high performance computing is required to be able to achieve sufficient resolution and solve all the physical scales of the sediment concentration up to the Batchelor scale. In this work, we will extend the `HySoP` library, a `Python`-based solver based on hybrid `MPI-OpenCL` programming that targets heterogeneous compute platforms. We will be able to solve custom user specified numerical problems such as incompressible Navier-Stokes fully on accelerator. The resulting solver will be modular enough to be able to run high performance `GPU` accelerated numerical simulation of particle-laden sediment flows.

Chapter 2

Numerical method

Contents

2.1 Outline of the method **63**
2.1.1 Operator splitting 63
2.1.2 Temporal resolution 65
2.1.3 Directional splitting 67
2.2 Spatial discretization **70**
2.2.1 Cartesian grids 70
2.2.2 Field discretization, interpolation and restriction 72
2.3 Hybrid particle-grid method **73**
2.3.1 Semi-lagrangian methods 74
2.3.2 Explicit Runge-Kutta methods 80
2.3.3 Construction of the remeshing formulas 82
2.4 Finite difference methods **85**
2.4.1 Spatial derivatives 86
2.4.2 Finite difference schemes 88
2.4.3 Performance considerations 90
2.5 Directional splitting **91**
2.5.1 Advection 92
2.5.2 Diffusion 93
2.5.3 Stretching 94
2.5.4 External forces and immersed boundaries 97
2.6 Fourier spectral methods **98**
2.6.1 Discrete Fourier transform 98
2.6.2 Spectral diffusion 105
2.6.3 Vorticity correction and computation of velocity 107
2.6.4 Real to real transforms and homogeneous boundary conditions 110
2.7 Extension to general non-periodic boundary conditions **118**
2.7.1 General spectral methods 118
2.7.2 Chebyshev collocation method 120
2.7.3 Chebyshev-Tau method 123
2.7.4 Chebyshev-Galerkin method 125
2.7.5 Comparison of the different methods 128

Introduction

In this chapter we derive a computational framework that will allow the resolution of the transport of a passive scalar in a turbulent flow, possibly at high Schmidt number. As seen in section 1.1.10, the modeling of the transport of a passive scalar in such a flow can be carried out using a system of continuous equations consisting of Navier-Stokes equations coupled with a scalar advection-diffusion equation (1.36). Both the momentum equation and the scalar equation can be viewed, at least partially, as advection-diffusion equations, one for the vorticity and the other for the scalar. Those two equations can be split into transport and diffusion terms, by relying on so-called operator splitting methods. The idea behind the proposed numerical method is to split the equations such that each subproblem can be solved by using a dedicated solver based on the most appropriate numerical scheme and by employing a space discretization that is regular enough to be handled by accelerators. Semi-lagrangian (remeshed) particle methods makes it possible to solve convection problems without imposing a Courant-Friedrichs-Lewy stability constraint, but a less restrictive stability condition allowing the use of larger time steps [Cottet et al. 2000][Van Rees et al. 2011][Mimeau et al. 2016]. On the one hand this is method particularly adapted for the transport of scalars in high Schmidt number flows, as introduced in chapter 1. On the other hand, through the presence of an underlying grid, this method allows to the use of eulerian solvers. In particular we will use Cartesian grids that are compatible with a wide variety of numerical methods such as finite difference methods and spectral methods.

The resulting solver should be able to run on multiple compute nodes, each exhibiting one or more accelerators. Thus, the numerical scheme is designed by employing numerical methods that are known to perform reasonably well in this context. In particular, each numerical scheme should exhibit adequate theoretical strong scalability with respect to the number of compute nodes used for the simulation. Choosing a numerical scheme that is not known to scale well would constitute a real performance bottleneck [Mathew et al. 2011]. Maximizing the memory bandwidth is crucial in this context because most of the aforementioned numerical methods are generally memory bound. The use of regular grids, enabled by the remeshed particle method, is particularly adapted to heavily vectorized hardware such as `GPUs`. Indeed, Cartesian grids will allow us to saturate memory controller of the target devices in memory-bound situations.

The chapter is organized as the following: we first give an overview of the method by splitting the problem into several subproblems, each of them accounting for a specific part of the underlying physics. Each of the resulting subproblems can be solved by using a grid compatible numerical method. Those additional numerical methods are introduced in the following subsections. We then show that some of those operators can further be split directionally reducing the overall cost and complexity of the method. The two last subsections focus on spectral methods from periodic problems to problems with any general set of boundary conditions. Spectral methods are mainly used to compute the velocity from the vorticity but also provide an alternative method to solve timestep limiting diffusion problems. The resulting hybrid particle-spectral-finite differences method will be particularly adapted for the problem of interest in this work.

2.1 Outline of the method

In this section we describe how we can split the problem into simpler subproblems to solve the incompressible Navier-Stokes equations on a n-dimensional spatial domain Ω coupled with a passive scalar. We will then decompose the problem by applying operator splitting strategies, consisting in solving successively different operators within the same time iteration.

Let n be the dimension of the spatial domain $\Omega \subset \mathbb{R}^n$, $\mathcal{T} = [t_{start}, t_{end}]$ be the time domain, $\boldsymbol{u}(\boldsymbol{x}, t) : \Omega \times \mathcal{T} \to \mathbb{R}^n$ be the velocity field, $\boldsymbol{\omega}(\boldsymbol{x}, t) : \Omega \times \mathcal{T} \to \mathbb{R}^p$ be the vorticity field, and $\theta(\boldsymbol{x}, t) : \Omega \times \mathcal{T} \to \mathbb{R}$ be the scalar field passively transported in the flow. The numerical method is obtained by starting with the incompressible Navier-Stokes equations in their velocity-vorticity formulation coupled with a passive scalar, recalled bellow.

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2.1a}$$

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{2.1b}$$

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{\omega} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} + \nu \Delta \boldsymbol{\omega} + \nabla \times \boldsymbol{f_{ext}} \tag{2.1c}$$

$$\frac{\partial \theta}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \theta = \kappa \Delta \theta \tag{2.1d}$$

In those equations, $\boldsymbol{f_{ext}}$ represents an external forcing term that depends on the problem being solved, ν is the constant kinematic viscosity of the fluid, and κ is the constant diffusivity of the scalar. Additionally, equation (2.1b) imposes that $p = \frac{n(n-1)}{2}$, such that the vorticity field has three components in 3D and only one in 2D. In two dimensions, the vorticity is orthogonal to the flow plane, so that stretching term $(\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u}$ vanishes and the vorticity equation (2.1c) reduces to a convection-diffusion equation that may include an additional external forcing term. Operator splitting will be applied for both the momentum equation (2.1c) and the scalar equation (2.1d).

2.1.1 Operator splitting

The approach followed here is summarized here for one time iteration. First, the scalar θ and the vorticity $\boldsymbol{\omega} = \nabla \times \boldsymbol{u}$ are evolved at fixed velocity $\boldsymbol{u}$ using an operator splitting on their respective equations. More precisely, we first split the scalar equation (2.1d) into independent convection and diffusion equations as the following:

1. **Scalar convection:** $$\frac{\partial \theta}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \theta = 0 \tag{2.2}$$

2. **Scalar diffusion:** $$\frac{\partial \theta}{\partial t} = \kappa \Delta \theta \tag{2.3}$$

We also split the vorticity transport equation (2.1c) in which the successive terms respectively model the convection, the stretching, the diffusion and the external forcing applied to the vorticity field:

1. **Convection:** $$\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,\boldsymbol{\omega} = 0 \quad (2.4)$$

2. **Diffusion:** $$\frac{\partial \boldsymbol{\omega}}{\partial t} = \nu \Delta \boldsymbol{\omega} \quad (2.5)$$

3. **Stretching:** $$\frac{\partial \boldsymbol{\omega}}{\partial t} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} \quad (2.6)$$ *(only in 3D)*

4. **External forces:** $$\frac{\partial \boldsymbol{\omega}}{\partial t} = \nabla \times \boldsymbol{f_{ext}} \quad (2.7)$$ *(depends on the problem being solved)*

As we will see later, the order of resolution of those equations is not important, but this is the one we will use throughout the manuscript. Viscous splitting within the framework of vortex methods was originally proposed in the early seventies by [Chorin 1973]. Its convergence has been proven by [Beale et al. 1982] and reformulated by [Cottet et al. 2000] to suit to the present vortex formulation.

The vorticity transport equation (VTE) has to be coupled to the system giving the velocity in terms of the vorticity. Hence, once the scalar and the vorticity have been evolved, the new divergence free velocity is computed from the vorticity by using a Poisson solver. The velocity is directly linked to the vorticity through the following relation obtained by taking the curl of (2.1b) and the incompressibility condition (2.1a):

$$\Delta \boldsymbol{u} = -\nabla \times \boldsymbol{\omega} \quad (2.8)$$

In some situations it is more convenient to use the Helmholtz decomposition of the velocity:

$$\boldsymbol{u} = \nabla \times \boldsymbol{\psi} + \nabla \phi \quad (2.9)$$

The stream function $\boldsymbol{\psi}$ and potential function ϕ then satisfy the following independent systems that have to be complemented with appropriate boundary conditions:

$$\begin{aligned} \Delta \boldsymbol{\psi} &= -\boldsymbol{\omega} && (2.10a) \\ \nabla \cdot \boldsymbol{\psi} &= 0 && (2.10b) \end{aligned} \qquad \text{and} \qquad \Delta \phi = 0 \quad (2.11)$$

The pressure can be easily recovered from the velocity field and the forcing term in the case of constant density ρ and dynamic viscosity μ by taking the divergence of equation (1.27b) together with the incompressibility condition (2.1a) to obtain the following Poisson equation:

$$\Delta P = \rho \nabla \cdot [\boldsymbol{f_{ext}} - (\boldsymbol{u} \cdot \nabla)\boldsymbol{u}] = \nabla \cdot [\boldsymbol{F_{ext}} - \rho(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}] \quad (2.12)$$

2.1.2 Temporal resolution

The time is discretized on time domain $\mathcal{T}$ from t_{start} to t_{end} using a variable timestep $dt^k > 0$ that will be determined later for each iteration k such that $t_0 = t_{start}$ and $t^{k+1} = t^k + dt^k \leq t_{end}$. In addition we define $T = t_{end} - t_{start} > 0$ as the total duration of the simulation. For a given field $F \in \{\boldsymbol{u}, \boldsymbol{\omega}, \theta\}$ and a given k with associated time $t^k \in \mathcal{T}$, we define F^k as $F(t^k)$.

Given an initial state $(\boldsymbol{u}^k, \boldsymbol{\omega}^k)$ at a given time t^k, we can compute $(\boldsymbol{u}^{k+1}, \boldsymbol{\omega}^{k+1})$ at time $t^{k+1} = t^k + dt^k$ using the following algorithm:

1. (a) Update the scalar and obtain θ^{k+1} from θ^k and $\boldsymbol{u}^k$ using equation (2.1d) by solving successively each split operator (2.2) and (2.3).

   (b) Update the vorticity and obtain an intermediate vorticity field $\boldsymbol{\omega}^{k,*}$ from $\boldsymbol{\omega}^k$ and $\boldsymbol{u}^k$ using equation (2.1c) at fixed velocity $\boldsymbol{u}^k$ by solving successively each split operator (2.4), (2.5), (2.6) and (2.7).

2. Correct the newly obtained vorticity $\boldsymbol{\omega}^{k,*}$ to obtain $\boldsymbol{\omega}^{k+1}$ such that the associated velocity $\boldsymbol{u}^{k+1}$ becomes divergence free by using the Helmholtz decomposition of the velocity (2.9). Compute the new velocity $\boldsymbol{u}^{k+1}$ from $\boldsymbol{\omega}^{k+1}$ by solving the Poisson problem (2.8) or by solving independently (2.10) and (2.11).

3. Compute the new timestep dt^{k+1} from the stability criteria of the numerical schemes depending on $\boldsymbol{u}^{k+1}$, $\boldsymbol{\omega}^{k+1}$, θ^{k+1} and the spatial discretization parameters. Advance in time by setting $t^{k+1} = t^k + dt^k$ and $k = k + 1$.

Steps (1a) and (1b) are independent and can be computed concurrently. Moreover the pressure term P^{k+1} can be recovered after step 2 by solving the Poisson problem (2.12).

The timestep only depends on the numerical methods used for each of the subproblems obtained in the splitting steps. We will denote Δt_{adv}, Δt_{diff}, $\Delta t_{stretch}$ and Δt_{fext} the timesteps computed from the stability criteria of each of the subproblems for the vorticity $\boldsymbol{\omega}$. The scalar transport also constraints the timestep to be less than $\Delta t_{adv,\theta}$ and $\Delta t_{diff,\theta}$. A constant maximum timestep Δt_{max} is also introduced to cover the case of dual-way couplings where the stability constraints would be too loose to ensure the convergence of the method. As we will never be interested in the pressure term, this results in algorithm 1.

The transported fields may exhibit a multiscale property. This means that the scalar θ may be discretized on a finer mesh compared to the velocity or vorticity mesh. We will see that for performance reasons we will only use Cartesian grid discretizations which are parametrized by a constant space step $\boldsymbol{dx} = (dx_1, \cdots, dx_n)$. Specific non-uniform grid discretizations may

be required for Chebyshev spectral solvers introduced in the last section.

$k \leftarrow 0$
$t^0 \leftarrow t_{start}$
$\boldsymbol{u}^0 \leftarrow \boldsymbol{u}(t = t_{start})$
$\boldsymbol{\omega}^0 \leftarrow \boldsymbol{\omega}(t = t_{start})$
$\theta^0 \leftarrow \theta(t = t_{start})$
$dt^0 \leftarrow \texttt{min}(\Delta t_{max}, \Delta t^0_{adv}, \Delta t^0_{diff}, \Delta t^0_{stretch}, \Delta t^0_{fext}, \Delta t^0_{adv,\theta}, \Delta t^0_{diff,\theta}, t_{end} - t^0)$
while $t^k < t_{end}$ **do**
  $\theta^{k,1} \leftarrow \texttt{convect}(\theta^k, dt^k, \boldsymbol{u}^k)$ — *Transport θ^k at velocity $\boldsymbol{u}^k$ during period dt^k*
  $\theta^{k+1} \leftarrow \texttt{diffuse}(\theta^{k,1}, dt^k, \kappa)$ — *Diffuse $\theta^{k,1}$ with coefficient κ during period dt^k*
  $\boldsymbol{\omega}^{k,1} \leftarrow \texttt{convect}(\boldsymbol{\omega}^k, dt^k, \boldsymbol{u}^k)$ — *Transport $\boldsymbol{\omega}^k$ at velocity $\boldsymbol{u}^k$ during period dt^k*
  $\boldsymbol{\omega}^{k,2} \leftarrow \texttt{diffuse}(\boldsymbol{\omega}^{k,1}, dt^k, \nu)$ — *Diffuse $\boldsymbol{\omega}^{k,1}$ with coefficient ν during period dt^k*
  $\boldsymbol{\omega}^{k,3} \leftarrow \texttt{stretch}(\boldsymbol{\omega}^{k,2}, dt^k, \boldsymbol{u}^k)$ — *Stretch $\boldsymbol{\omega}^{k,2}$ at velocity $\boldsymbol{u}^k$ during period dt^k*
  $\boldsymbol{\omega}^{k,4} \leftarrow \texttt{forcing}(\boldsymbol{\omega}^{k,3}, dt^k)$ — *Apply external forces to $\boldsymbol{\omega}^{k,3}$ during period dt^k*
  $\boldsymbol{\omega}^{k+1} \leftarrow \texttt{correct}(\boldsymbol{\omega}^{k,4})$ — *Correct the vorticity $\boldsymbol{\omega}^{k,4}$ to ensure $\nabla \cdot \boldsymbol{u}^{k+1} = 0$*
  $\boldsymbol{u}^{k+1} \leftarrow \texttt{poisson}(\boldsymbol{\omega}^{k+1})$ — *Compute $\boldsymbol{u}^{k+1}$ from $\boldsymbol{\omega}^{k+1}$ using a Poisson solver*
  $t^{k+1} \leftarrow t^k + dt^k$
  $dt^{k+1} \leftarrow \texttt{min}(\Delta t_{max}, \Delta t^{k+1}_{adv}, \Delta t^{k+1}_{diff}, \Delta t^{k+1}_{stretch}, \Delta t^{k+1}_{fext}, \Delta t^{k+1}_{adv,\theta}, \Delta t^{k+1}_{diff,\theta}, t_{end} - t^{k+1})$
  $k \leftarrow k + 1$
end

Algorithm 1: Algorithm to solve the one-way coupling of a scalar with the incompressible Navier-Stokes equations (2.1) by using operator splitting and variable timestep dt.

Classical eulerian solvers on such grids are submitted to a CFL condition [Courant et al. 1967] that caps the timestep Δt_{adv} depending on velocity $\boldsymbol{u}$ and space step $\boldsymbol{dx}$:

$$C = \Delta t_{adv} \left(\sum_{i=1}^{n} \frac{\|\boldsymbol{u}_i\|_\infty}{dx_i} \right) < C_{max} \tag{2.13}$$

When $\boldsymbol{\omega}$ and θ live on the same grid, it entails that $\boldsymbol{dx}_{\boldsymbol{\omega}} = \boldsymbol{dx}_\theta$ and thus $\Delta t_{adv} = \Delta t_{\theta,adv}$. Under the assumption of high Schmidt number, $Sc = \frac{\nu}{\kappa} \gg 1$, it is of interest to choose a finer grid to solve for all the physical scales of the scalar. In this case we may choose $\boldsymbol{dx} = \sqrt{Sc}\, \boldsymbol{dx}_\theta$, which implies $\Delta t_{adv,\theta} = \Delta t_{adv}/\sqrt{Sc}$. Within this framework, passively coupling a single scalar with a high Schmidt number can potentially slow down the whole simulation by a factor $\sqrt{Sc}$.

Using a lagrangian scheme instead of an eulerian scheme for advection allows to get rid of this factor at the cost of loosing the underlying uniform grid discretization of eulerian methods. The present method use semi-lagrangian schemes, that are able to combine the best of both worlds. With particle methods, the stability of the method just depends on the gradient of the velocity:

$$LCFL = \Delta t \, \|\nabla \boldsymbol{u}\|_\infty \leq C_{max} \tag{2.14}$$

The scalar diffusion term may also restrict the timestep because of stability considerations. The use of an explicit finite difference scheme on a diffusion equation with diffusion coefficient α is stable under:

$$C = \Delta t_{diff} \max_{i \in [\![1,n]\!]} \left(\frac{\alpha}{dx_i^2} \right) < C_{max} \tag{2.15}$$

When $\boldsymbol{\omega}$ and θ live on the same grid, we have $\boldsymbol{dx_\omega} = \boldsymbol{dx_\theta}$ such that $\Delta t_{diff} = \Delta t_{diff,\theta}/Sc$ and thus $\Delta t_{diff} \ll \Delta t_{diff,\theta}$, the scalar has no impact on the diffusion time step for high Schmidt numbers. It is however possible to use an unconditionally stable solver such as spectral diffusion to get rid of Δt_{diff}. If we choose $\boldsymbol{dx} = \sqrt{Sc}\ \boldsymbol{dx_\theta}$, we obtain $\Delta t_{diff,\theta} = \Delta t_{diff}$. Hence the additional diffusion of scalar at high Schmidt number should never impact the number of iterations of the solver given the same explicit diffusion schemes are used. When the diffusion limits the timestep is is possible to use an implicit spectral formulation instead.

2.1.3 Directional splitting

The operator splitting introduced in section 2.1.1 and illustrated in algorithm 1 is further decomposed by using a directional Strang splitting on advection, diffusion, stretching and external forcing operators. Such a Strang splitting can be used to speed up calculation for problems involving multidimensional partial differential equations by reducing them to a sum of one-dimensional problems [Strang 1968]. This method makes it possible to decouple the spatial directions by solving one-dimensional advection problems by alternating the directions.

Let L be a linear differential operator on y such that

$$\frac{\partial \tilde{y}}{\partial t} = L(\tilde{y}) \quad t \in [t^k, t^k + dt[\tag{2.16a}$$

$$\tilde{y}^k = \tilde{y}(t^k) = y^k \tag{2.16b}$$

We want to solve problem 2.16 to obtain $y^{k+1} = \tilde{y}(t^k + dt)$.

A Strang splitting allows to solve problem (2.16) when L can be decomposed as a sum of p differential operators by solving successively partial differential equations arrising from the splitting. It is particularly interesting when $L = L_1 + L_2 + \cdots + L_p$ is relatively difficult or costly to compute directly while there are readily available methods to compute each of the splitted differential operators L_i separately. In such a case, the method can then be used to speed up calculations at first and second order in time.

First order in time Strang splitting is achieved by solving successfully the p operators during a full timestep dt as the following:

$$\underset{y^k = y_0^k}{\rightarrow} L_1\,(dt) \underset{y_1^k}{\rightarrow} L_2\,(dt) \underset{y_2^k}{\rightarrow} \cdots \underset{y_{p-1}^k}{\rightarrow} L_p\,(dt) \underset{y_p^k = y^{k+1}}{\rightarrow}$$

input : $dt, y^k = y(t^k)$
output: $y^{k+1} = y(t^k + dt)$, solution of problem 2.16
$y_0^k \leftarrow y^k$
for $i \in [\![1, p]\!]$ **do**
  Solve $\left| \begin{array}{l} \dfrac{\partial \tilde{y}}{\partial t} = L_i(\tilde{y}) \quad t \in [t^k, t^k + dt[\\ \tilde{y}^k = \tilde{y}(t^k) = y_{i-1}^k \end{array} \right.$
  $y_i^k \leftarrow \tilde{y}(t^k + dt)$
end
$y^{k+1} \leftarrow y_p^k$

Algorithm 2: First order Strang splitting applied to operator $L = L_1 + \cdots + L_p$

Second order (in time) Strang splitting can be achieved by solving successfully p operators back and forth with half a timestep:

$$\underset{y^k = y_0^k}{\rightarrow} L_1\left(\frac{dt}{2}\right) \underset{y_1^k}{\rightarrow} L_2\left(\frac{dt}{2}\right) \underset{y_2^k}{\rightarrow} \cdots \underset{y_{p-1}^k}{\rightarrow} \underbrace{L_p\left(\frac{dt}{2}\right) \underset{y_p^k = y_0^{k+\frac{1}{2}}}{\rightarrow} L_p\left(\frac{dt}{2}\right)}_{\text{or equivalently } L_p(dt)} \underset{y_1^{k+\frac{1}{2}}}{\rightarrow} \cdots \underset{y_{p-2}^{k+\frac{1}{2}}}{\rightarrow} L_2\left(\frac{dt}{2}\right) \underset{y_{p-1}^{k+\frac{1}{2}}}{\rightarrow} L_1\left(\frac{dt}{2}\right) \underset{y_p^{k+\frac{1}{2}} = y^{k+1}}{\rightarrow}$$

input : $y^k = y(t^k)$
output: $y^{k+1} = y(t^k + dt)$, solution of problem 2.16
$y_0^k \leftarrow y^k$
for $i \in [\![1, p]\!]$ **do**
  Solve $\left| \begin{array}{l} \dfrac{\partial \tilde{y}}{\partial t} = L_i(\tilde{y}) \quad t \in [t^k, t^k + \frac{1}{2}dt[\\ \tilde{y}^k = \tilde{y}(t^k) = y_{i-1}^k \end{array} \right.$
  $y_i^k \leftarrow \tilde{y}(t^k + \frac{1}{2}dt)$
end
$y_0^{k+\frac{1}{2}} \leftarrow y_p^k$
for $i \in [\![1, p]\!]$ **do**
  Solve $\left| \begin{array}{l} \dfrac{\partial \tilde{y}}{\partial t} = L_{p-i+1}(\tilde{y}) \quad t \in [t^k + \frac{1}{2}dt, t^k + dt[\\ \tilde{y}^{k+\frac{1}{2}} = \tilde{y}(t^k + \frac{1}{2}dt) = y_{i-1}^{k+\frac{1}{2}} \end{array} \right.$
  $y_i^{k+\frac{1}{2}} \leftarrow \tilde{y}(t^k + dt)$
end
$y^{k+1} \leftarrow y_p^{k+\frac{1}{2}}$

Algorithm 3: Second order Strang splitting applied to operator $L = L_1 + \cdots + L_p$

In order to obtain only one-dimensional problems, the idea is to split directionally all partial differential equations relating to advection (2.2, 2.4), diffusion (2.3, 2.5), stretching (2.6) and external forcing (2.7) by breaking them down into n directional terms of the form:

$$\begin{aligned}\frac{\partial \boldsymbol{f}}{\partial t} &= \boldsymbol{L}\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}}{\partial x_1}, \cdots, \frac{\partial \boldsymbol{f}}{\partial x_n}, \frac{\partial \boldsymbol{f}^2}{\partial x_1^2}, \cdots, \frac{\partial \boldsymbol{f}^2}{\partial x_n^2}\right) \\ &= \boldsymbol{L}_1\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}}{\partial x_1}, \frac{\partial \boldsymbol{f}^2}{\partial x_1^2}\right) + \cdots + \boldsymbol{L}_n\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}}{\partial x_n}, \frac{\partial \boldsymbol{f}^2}{\partial x_n^2}\right)\end{aligned} \tag{2.17}$$

where $\boldsymbol{f} = [\boldsymbol{\omega}_1, \cdots, \boldsymbol{\omega}_p, \boldsymbol{u}_1, ..., \boldsymbol{u}_n, \theta]^T$. This kind of directional splitting offers many advantages as only one-dimensional problems have to be solved for directionally split operators leading to a dimension agnostic implementation. It also reduces the arithmetic intensity while being well suited for vectorized hardware architectures such as `CPU`s and `GPU`s [Magni et al. 2012][Etancelin 2014]. Once the decomposition $L_1, \cdots, L_n$ is known for all operators we can apply first or second order Strang splitting by grouping each direction to solve the respective operators as exposed in algorithm 4.

$(k, t^0) \leftarrow (0, t_{start})$
$(\boldsymbol{u}^0, \boldsymbol{\omega}^0, \theta^0) \leftarrow \boldsymbol{u}(t = t_{start}), \boldsymbol{\omega}(t = t_{start}), \theta(t = t_{start})$
$dt^0 \leftarrow \texttt{min}(\Delta t_{max}, \Delta t^0_{adv}, \Delta t^0_{diff}, \Delta t^0_{stretch}, \Delta t^0_{fext}, \Delta t^0_{adv,\theta}, \Delta t^0_{diff,\theta}, t_{end} - t^0)$
while $t^k < t_{end}$ **do**
  // Advection and diffusion of the scalar θ at constant velocity $\boldsymbol{u}$
  $\theta^{k,0} \leftarrow \theta^k$
  for $i \leftarrow 0$ *to* $(n-1)$ **do**
    $\theta^{k,2i+1} \leftarrow \texttt{directional_convect}_{\texttt{i+1}}(\theta^{k,2i+0}, dt^k, u^k_{i+1})$
    $\theta^{k,2i+2} \leftarrow \texttt{directional_diffuse}_{\texttt{i+1}}(\theta^{k,2i+1}, dt^k, \nu)$
  end
  // Advection, diffusion, stretching and ext. forcing of vorticity $\boldsymbol{\omega}$ at constant $\boldsymbol{u}$
  $\boldsymbol{\omega}^{k,0} \leftarrow \boldsymbol{\omega}^k$
  for $i \leftarrow 0$ *to* $(n-1)$ **do**
    $\boldsymbol{\omega}^{k,4i+1} \leftarrow \texttt{directional_convect}_{\texttt{i+1}}(\boldsymbol{\omega}^{k,4i+0}, dt^k, u^k_{i+1})$
    $\boldsymbol{\omega}^{k,4i+2} \leftarrow \texttt{directional_diffuse}_{\texttt{i+1}}(\boldsymbol{\omega}^{k,4i+1}, dt^k, \nu)$
    $\boldsymbol{\omega}^{k,4i+3} \leftarrow \texttt{directional_stretch}_{\texttt{i+1}}(\boldsymbol{\omega}^{k,4i+2}, dt^k, \boldsymbol{u}^k)$
    $\boldsymbol{\omega}^{k,4i+4} \leftarrow \texttt{directional_forcing}_{\texttt{i+1}}(\boldsymbol{\omega}^{k,4i+3}, dt^k)$
  end
  // Correction of the vorticity and computation of the new divergence-free velocity
  $\boldsymbol{\omega}^{k+1} \leftarrow \texttt{correct}(\boldsymbol{\omega}^{k,4n})$
  $\boldsymbol{u}^{k+1} \leftarrow \texttt{poisson}(\boldsymbol{\omega}^{k+1})$
  // Compute the new timestep and advance in time
  $t^{k+1} \leftarrow t^k + dt^k$
  $dt^{k+1} \leftarrow \texttt{min}(\Delta t_{max}, \Delta t^{k+1}_{adv}, \Delta t^{k+1}_{diff}, \Delta t^{k+1}_{stretch}, \Delta t^{k+1}_{fext}, \Delta t^{k+1}_{adv,\theta}, \Delta t^{k+1}_{diff,\theta}, t_{end} - t^{k+1})$
  $k \leftarrow k + 1$
end

Algorithm 4: Extension of algorithm 1 by using first order directional Strang splitting on both the scalar θ and the vorticity $\boldsymbol{\omega}$.

2.2 Spatial discretization

The physical domain Ω is first embedded into a n-dimensional bounding box $\mathcal{B}$ that has origin $\boldsymbol{x}_{min} = (x_1^{min}, \cdots, x_n^{min})$ and length $\boldsymbol{L} = (L_1, \cdots, L_n)$ such that $\boldsymbol{x}_{max} = (x_1^{min} + L_1, \cdots, x_n^{min} + L_n)$ and $\Omega \subset [x_1^{min}, x_1^{max}] \times \cdots \times [x_n^{min}, x_n^{max}]$. If Ω is not initially box-shaped, it is possible to inject additional penalization terms into the momentum equation (2.1c) and split it accordingly to impose the right boundary conditions on $\overline{\Omega}$. This includes homogeneous Dirichlet boundary conditions [Angot et al. 1999], homogeneous Neumann boundary conditions [Kadoch et al. 2012] and its non homogeneous counterpart [Sakurai et al. 2019] and can also be used to model the boundaries of complex obstacles moving inside the fluid domain. However, in this manuscript, the physical domain Ω will systematically represent a rectangular cuboid of dimension n and we will not make any distinctions between Ω and $\mathcal{B}$.

2.2.1 Cartesian grids

Such a box is then discretized by using rectilinear grids defined by a tessellation of hyper-rectangle cells that are all congruent to each other. We will denote $\boldsymbol{dx} = (dx_1, \cdots, dx_n)$ the size of each of those elements and $\boldsymbol{\mathcal{N}}^c = (\mathcal{N}_1^c, \cdots, \mathcal{N}_n^c)$ their number in each direction such that $\boldsymbol{L} = \boldsymbol{\mathcal{N}}^c \odot \boldsymbol{dx}$ where $\odot$ denotes elementwise multiplication ($L_i = \mathcal{N}_i^c dx_i \ \forall i \in [\![1, n]\!]$). When $dx_1 = \cdots = dx_n$, all the elements tesselating the domain are hypercubes and we obtain a Cartesian grid. All the components of velocity $\boldsymbol{u}$ and vorticity $\boldsymbol{\omega}$ are discretized on the same collocated grid, formed by the $\boldsymbol{\mathcal{N}}^v$ cell vertices defined by $\boldsymbol{x_i} = \boldsymbol{x}_{min} + \boldsymbol{i} \odot \boldsymbol{dx}$ for all $\boldsymbol{i} = (i_1, \cdots, i_n) \in [\![0, \mathcal{N}_1^v[\![\times \cdots \times [\![0, \mathcal{N}_n^v[\![$.

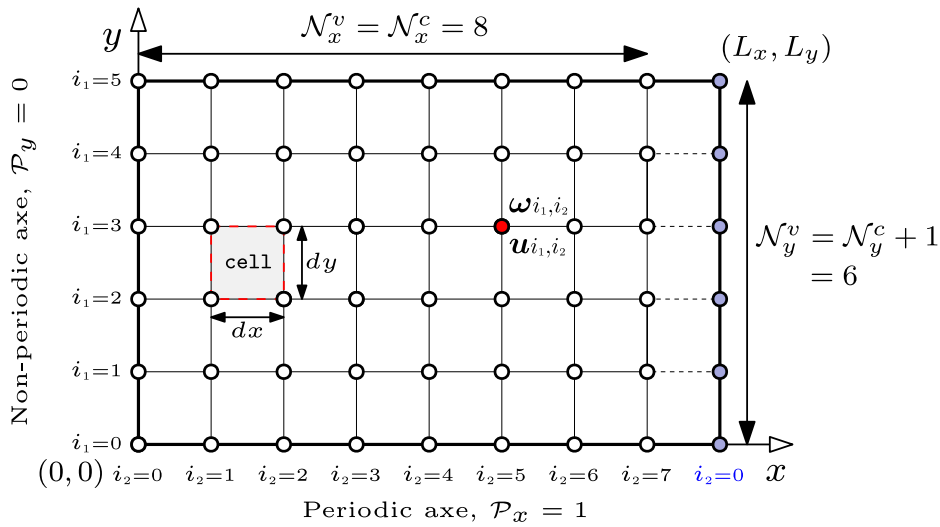
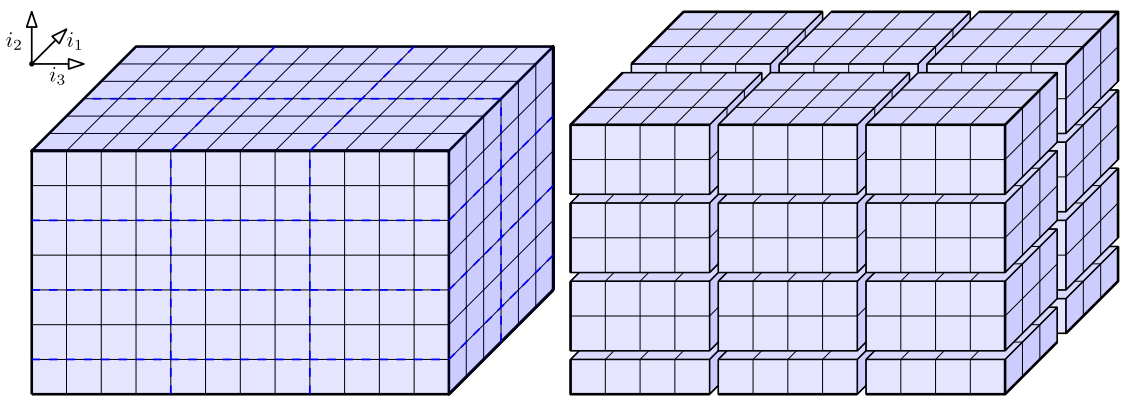


Figure 2.1 – Illustration of the discretization of rectangle-shaped domain $\mathcal{B}$ spanning from $\boldsymbol{x}_{min} = (0, 0)$ to $\boldsymbol{x}_{max} = (L_y, L_x)$. Here $\mathcal{P} = (0, 1)$, the domain is L_x-periodic on the x-axis. It is discretized by using $\mathcal{N}^c = (5, 8)$ rectangle cells of size $\boldsymbol{dx} = (dy, dx) = (L_y/\mathcal{N}_y^c, L_x/\mathcal{N}_x^c)$, yielding $\boldsymbol{\mathcal{N}}^v = (6, 8)$ vertices where the velocity $\boldsymbol{u}$ and vorticity $\boldsymbol{\omega}$ fields are defined. The blue vertices are not included in the discretization because they represent redundant vertices arising from the periodization of the x-axis, as $\boldsymbol{\omega}_{*,8} = \boldsymbol{\omega}_{*,0}$.

The number of vertices on axe i depends on whether the domain is L_i-periodic or not on the said axis. If we define $\mathcal{P} = (\mathcal{P}_1, \cdots, \mathcal{P}_n) \in \{0,1\}^n$ such that $\mathcal{P}_i = 1$ if the domain is periodic on axe i and $\mathcal{P}_i = 0$ if not, the number of vertices $\boldsymbol{\mathcal{N}}^v$ is defined by $\mathcal{N}_i^v = \mathcal{N}_i^c + 1 - \mathcal{P}_i \ \ \forall i \in [\![1, n]\!]$. The vertices are globally indexed and stored in memory by using row-major ordering, where consecutive elements of a row reside next to each other. This ordering is defined by the following relation: $\mathcal{I}_{\boldsymbol{i}} = ((((\cdots + i_{n-3})\mathcal{N}_{n-2}^v + i_{n-2})\mathcal{N}_{n-1}^v + i_{n-1})\mathcal{N}_n^v + i_n) = \boldsymbol{i} \cdot \boldsymbol{\mathcal{S}}$ with stride $\boldsymbol{\mathcal{S}} = (\mathcal{S}_1, \cdots, \mathcal{S}_n)$ where $\mathcal{S}_i = \Pi_{j=i+1}^n \mathcal{N}_j^v$ as illustrated on figure 3.29.

The convention in 2D is that the first axe corresponds to the y-axis and the second one corresponds to the x-axis such that a domain of size $\boldsymbol{L} = (L_1, L_2) = (L_y, L_x)$ is discretized using $\boldsymbol{\mathcal{N}}^c = (\mathcal{N}_y^c, \mathcal{N}_x^c)$ rectangle cells of size $\boldsymbol{dx} = (dy, dx)$ forming $\boldsymbol{\mathcal{N}}^v = (\mathcal{N}_y^v, \mathcal{N}_x^v)$ vertices globally indexed by $\mathcal{I}_{\boldsymbol{i}} = \mathcal{I}_{(i_1,i_2)} = i_1\mathcal{N}_x^v + i_2$ by taking into account the periodicity of the domain $\boldsymbol{\mathcal{P}} = (\mathcal{P}_y, \mathcal{P}_x)$. The 3D case follows the same idea, the first axis being the z-axis. In this case, the axes are ordered as (z, y, x) such that vertex $v_{\boldsymbol{i}} = v_{(i_1,i_2,i_3)}$ has a global index $\mathcal{I}_{\boldsymbol{i}} = ((i_1\mathcal{N}_y^v + i_2)\mathcal{N}_x^v + i_3 = \boldsymbol{i} \cdot \boldsymbol{\mathcal{S}}$ with $\boldsymbol{\mathcal{S}} = (\mathcal{N}_x^v\mathcal{N}_y^v, \mathcal{N}_x^v, 1)$ and position $\boldsymbol{x_i} = (z_{min} + i_1 dz, y_{min} + i_2 dy, x_{min} + i_3 dx)$. This convention is made such that the x-axis always represent the last axis where the vertex indices (and memory offsets) are contiguous.

The rationale behind this choice of discretization is that parallel computing is easy on rectilinear grids. First, when the grid is big enough with respect to the number of compute ressources, it is easy to perform domain decomposition on multiple processes as shown on figure 2.2. Secondly, Cartesian grids enable a wide variety of numerical methods such as finite differences [Courant et al. 1952], spectral methods [Canuto et al. 2012] and pseudo-spectral methods [Fornberg 1998]. It can be used in conjonction with remeshed particle methods [Cottet et al. 2000]. Last but not least, such a regular data structure offers uniform mapping from indexes to vertex coordinates, and can be allocated as a single large contiguous chunk of memory. Hence, this data structure is adapted for use with parallel computing platforms such as `CUDA` [Luebke 2008] and `OpenCL` [Stone et al. 2010]. Indeed, hardware peculiarities such as memory hierarchy or vector execution units inherent to the processors or coprocessors this kind of application programming interfaces can drive make them attractive [Owens et al. 2008][Cottet et al. 2013].



Overall Simulation Domain $\mathcal{B}$ Domain Decomposition

Figure 2.2 – Example of domain decomposition in 24 subdomains

2.2.2 Field discretization, interpolation and restriction

Each scalar field ω_i, u_i and θ are then discretized on such regular grids. In all this work the velocity and the vorticity components will be discretized on grids with the same size $\boldsymbol{\mathcal{N}}^v$. The scalar θ may be discretized on a finer grid such that this new grid contains $\boldsymbol{\mathcal{N}}^c_\theta = \boldsymbol{m} \odot \boldsymbol{\mathcal{N}}^c$ cells where $\boldsymbol{m} = \boldsymbol{\mathcal{N}}^c_\theta \oslash \boldsymbol{\mathcal{N}}^c \in \mathbb{N}^n$ is the grid ratio. Its associated number of vertices can be obtained as before by computing $\boldsymbol{\mathcal{N}}^v_\theta = \boldsymbol{\mathcal{N}}^c_\theta + 1 - \boldsymbol{\mathcal{P}}$. From now on we denote $N = \prod_{i=1}^n \mathcal{N}^v_i$ and $N_\theta = \prod_{i=1}^n \mathcal{N}^v_{\theta,i}$.

In practice those variables may interact due to the couplings and thus we need to be able to compute a restriction of θ on the coarse grid and interpolate the velocity and vorticity fields on the fine grid. There exist many different methods to compute an interpolation:

- Local polynomial interpolation (n-linear or n-cubic splines, see [Keys 1981])
- Local interpolation by using interpolating kernels defined in section 2.3.
- Global Fourier spectral interpolation for periodic and homogeneous boundary conditions. Can be computed in $\mathcal{O}\left(N \log N + N_\theta \log N_\theta\right)$ by using a n-dimensional forward and backward Fast Fourier transform with zero-padding (see section 2.6).
- Global Chebyshev spectral interpolation for general boundary conditions. Can be computed in $\mathcal{O}\left(N \log N + N_\theta \log N_\theta\right)$ at the Chebyshev-Gauss-Lobatto points by using Fast Chebyshev transforms and in $\mathcal{O}\left(N^2 + N_\theta^2\right)$ on a regular grid (see section 2.7).

For the local interpolation methods, the associated interpolation kernel has a compact support $\overline{\boldsymbol{s}} = \boldsymbol{s} \odot \boldsymbol{dx}$ with $\boldsymbol{s} \in \mathbb{N}^n$. Local interpolation algorithms usually treats one direction after another to reduce the computational complexity of the method, going from $\mathcal{O}\left(\left[\prod_{i=1}^n s_i\right] N_\theta\right)$ to $\mathcal{O}\left(\left[\sum_{i=1}^n s_i\right] N_\theta\right)$.

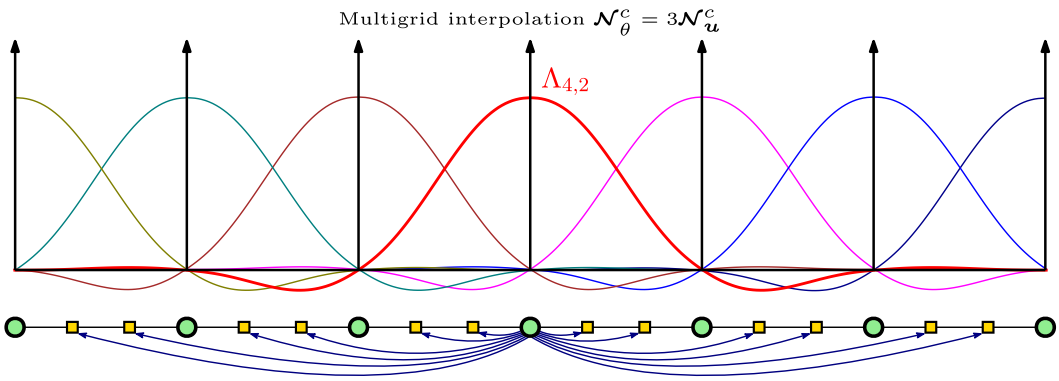
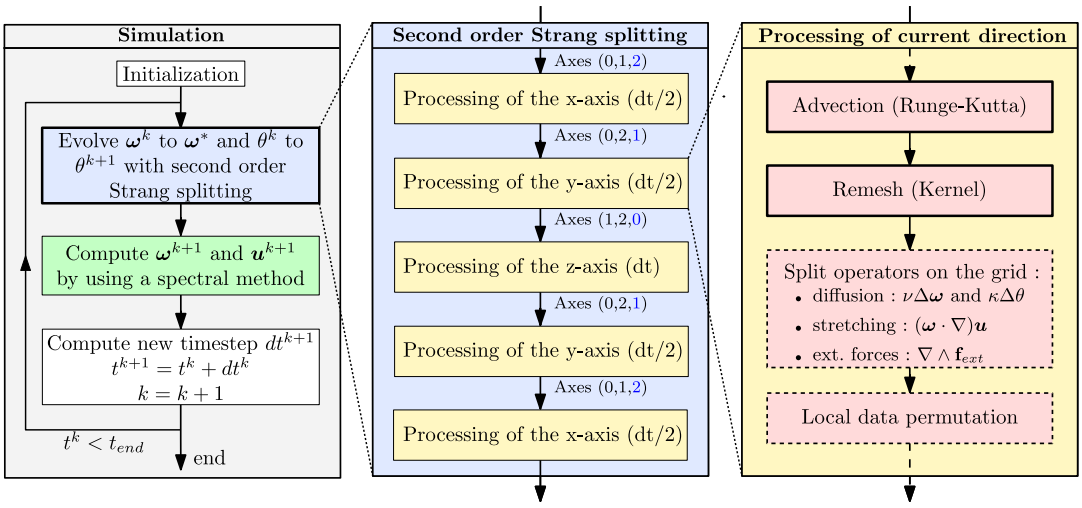


Figure 2.3 – Local interpolation by using an interpolating kernel (2.35)

Each of those methods have a corresponding restriction method such as low-pass filtering for global spectral methods and local restrictions for the local methods.

2.3 Hybrid particle-grid method

In this section we propose to introduce an hybrid method for the transport of scalar quantities using the particle method. This method will allow us to solve equations (2.2) and (2.4) and constitute the very first step of our numerical method as shown bellow:



A particle method is able to solve a transport equation of the form:

$$\frac{\partial\theta}{\partial t} + (\boldsymbol{u}\cdot\nabla)\,\theta = 0 \tag{2.18}$$

where the scalar field θ is transported at velocity $\boldsymbol{u}$.

With particle methods, the variables are discretized on a set of particles which correspond to an elementary volume of the domain. The general principle is that particles carry quantities corresponding to the variables of the problem. The particles are then transported in the velocity field and are free to evolve over the whole domain in a grid-less fashion.

In such a lagrangian framework, the scalar field $\theta(\boldsymbol{x})$ can be approximated by the discrete sum of P particles $\{p_i \mid i \in [\![1, P]\!]\}$ each carrying a quantity θ_i at position $\boldsymbol{x}_i$:

$$\theta(\boldsymbol{x}) \simeq \sum_{i=1}^{P} \theta_i W_\varepsilon(\boldsymbol{x} - \boldsymbol{x}_i) \tag{2.19}$$

The carried quantity θ_i is obtained by computing the mean value of θ inside the elementary discrete volume V_i that the particle represent:

$$\theta_i = \frac{1}{|V_i|}\int_{V_i} \theta(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \tag{2.20}$$

The function W_ε that appears in the equation (2.19) is a regularization function whose choice and properties are specific to each particle method. In all cases, it is constructed so that its limit when ε tends to 0 is the Dirac measure δ.

During the resolution, the particles are displaced relative to the velocity field and then interact to give a new distribution of the transported quantities. If the transported quantity is the vorticity $\boldsymbol{\omega}$, as in equation (2.4), this leads to a new velocity field $\boldsymbol{u}$. Particles are transported along their characteristics and evolve on a grid free domain. The motion of the particles is dictated by the following set of equations:

$$\frac{\mathrm{d}\boldsymbol{x}_i}{\mathrm{d}t} = \boldsymbol{u}(\boldsymbol{x}_i, t) \quad \forall i \in [\![1, P]\!] \tag{2.21}$$

As a grid-free method, it has the advantage of being free from any stability condition that would restrict the timestep depending on some grid discretization parameter or the distance between two particles. Indeed, with traditional eulerian methods, the timestep is constrained by a Courant-Friedrichs-Lewy (CFL) condition where C is often less than one.

$$CFL = \frac{\Delta t \, \|\boldsymbol{u}\|_\infty}{\Delta x} \leq C \tag{2.22}$$

With particle methods, the stability of the method, and thus the timestep, just depends on the gradient of the velocity:

$$LCFL = \Delta t \, \|\nabla \boldsymbol{u}\|_\infty \leq C \tag{2.23}$$

In practice, the constant C, called the LCFL (for lagrangian CFL), leads to a less restrictive condition than the CFL condition (2.22). This generally makes it possible to use larger time steps than in the case of an eulerian method. This feature is particularly interesting for high Schmidt number flows [Etancelin et al. 2014].

2.3.1 Semi-lagrangian methods

A well-known disadvantage of lagrangian methods is that the particles have trajectories that follow the velocity field and thus particle distribution may undergo distortion. This manifests itself by the clustering or spreading of the flow elements in high strain regions and leads to the deterioration of the representation of the fields by lack or excess of particles in some areas of the flow [Cottet et al. 2000]. This particle overlapping condition has been recognized as a major difficulty to perform accurate simulations, in particular for non-linear problems.

Semi-lagrangian methods were developed for the resolution of flows dominated by advection, mainly in the field of meteorology [Staniforth et al. 1991] and are able to combine the advantages of both eulerian (grid-based) and lagrangian (grid-free) methods. To maintain the regularity of the particle distribution, particles are remeshed on an underlying eulerian grid every few timesteps [Koumoutsakos et al. 1995]. When the particles are remeshed at every timestep, we obtain a systematic representation of the transported quantities on the grid,

hence the name semi-lagrangian method. Within this framework particle methods can be combined with grid-based techniques such as spectral methods and finite difference methods. Because the particles are systematically remeshed it is possible to create the particles only at the beginning of each advection step and to destroy them just after remeshing.

There exist two classes of semi-lagrangian methods:

- **Backward semi-lagrangian method:** The resolution consists to start from a grid point and integrate backward in time along the characteristics (advection step). The quantity interpolated at this new point is then transported to the starting point (interpolation step).
- **Forward semi-lagrangian method:** In this case we also start from a grid point, but the trajectory is integrated forward in time (advection step). The quantity transported by the particle is then distributed on the surrounding points (remeshing step).

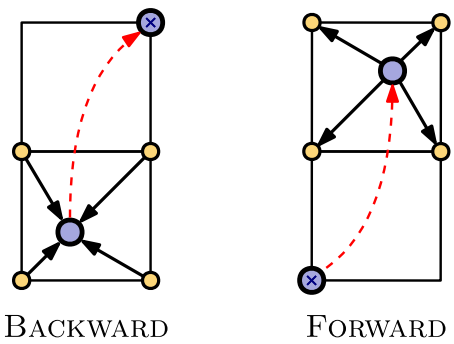


Figure 2.4 – Illustration of the two classes of semi-lagrangian methods

In this work we will use the high-order forward semi-lagrangian particle method proposed by [Cottet et al. 2014]. As a forward method, it features an advection and a remeshing step.

Advection step

At each time iteration of the method, particles are created on the underlying eulerian grid at positions $\{\boldsymbol{x_i} = \boldsymbol{x}_{min} + \boldsymbol{i} \odot \boldsymbol{dx} \mid \boldsymbol{i} \in [\![0, \mathcal{N}_1^v[\![\times \cdots \times [\![0, \mathcal{N}_n^v[\![\}$ and are transported along their characteristics while the associated transported quantities θ_i remain constant. The particle positions are driven by transport equations (2.21) and the new position of the particles are given by the following set ordinary differential equations:

$$\begin{aligned} \frac{\partial \boldsymbol{x_i}}{\partial t} &= \boldsymbol{u}(\boldsymbol{x_i}) \quad t \in [t^k, t^{k+1}[\\ \boldsymbol{x_i^k} &= \boldsymbol{x_i}(t^k) = \boldsymbol{x}_{min} + \boldsymbol{i} \odot \boldsymbol{dx} \end{aligned} \tag{2.24}$$

In this work we will numerically integrate particle positions in time by using explicit Runge-Kutta methods, which are described in section 2.3.2. The only difficuly that may arrise during this step is the interpolation of the velocity field, at all the intermediate positions of the particles (see figure 2.5). For a first order scheme, namely the Euler scheme, no interpolation is needed because the particles initially coincide with the eulerian grid where the velocity is known from the last timestep. In this case, the new position of the particles can be computed as the following:

$$\boldsymbol{x}_{\boldsymbol{i}}^{k+1} = \boldsymbol{x}_{\boldsymbol{i}}^{k} + dt\ \boldsymbol{u}_{\boldsymbol{i}}^{k} \tag{2.25}$$

Using a Runge-Kutta scheme of the second order leads to this second expression:

$$\begin{aligned} \boldsymbol{x}_{\boldsymbol{i},1}^{k} &= \boldsymbol{x}_{\boldsymbol{i}}^{k} + \frac{dt}{2}\boldsymbol{u}_{\boldsymbol{i}}^{k} \\ \boldsymbol{x}_{\boldsymbol{i}}^{k+1} &= \boldsymbol{x}_{\boldsymbol{i}}^{k} + dt\ \boldsymbol{u}\left(\boldsymbol{x}_{\boldsymbol{i},1}^{k}\right) \end{aligned} \tag{2.26}$$

Here it is clear that the intermediate position $\boldsymbol{x}_{\boldsymbol{i},1}^{k}$ will not always be aligned with the grid and the computation of the velocity at this point requires an interpolation. Figure 2.5 shows the difference between the two methods, by using bilinear interpolation.

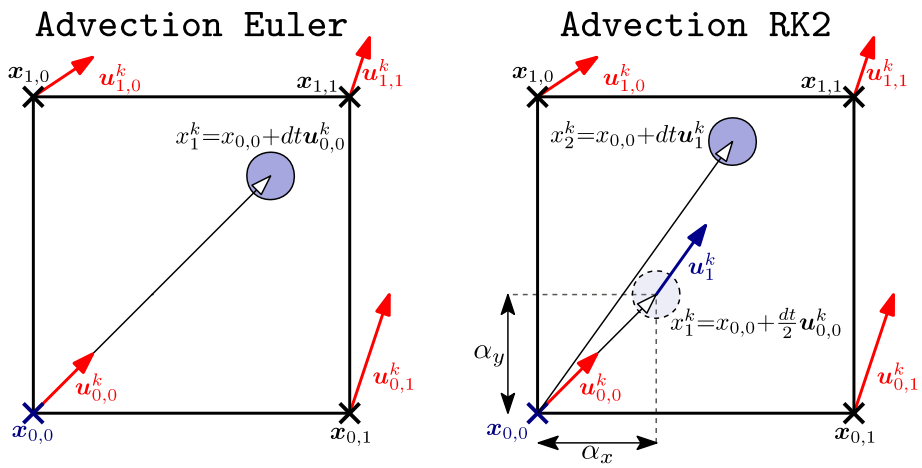


Figure 2.5 – Advection with first and second order explicit Runge-Kutta schemes

During the advection step, n-linear interpolation is used to evaluate velocity at positions that are not grid aligned. A value inside a cuboid cell is computed by the weighted average of the values present at the 2^n vertices forming the cell. As the scheme is not CFL constrained, a given particle can potentially travel over multiple cells at once and the cell in which the particle may arrive intersects the sphere centered at initial particle position $\boldsymbol{x_i} = \boldsymbol{x}_{min} + \boldsymbol{i} \odot \boldsymbol{dx}$ with radius $r = dt\,\|\boldsymbol{u}\|_\infty$. Let $\boldsymbol{x}$ be the position at which the velocity has to be interpolated and $\boldsymbol{y} = (\boldsymbol{x} - \boldsymbol{x}_{min}) \oslash \boldsymbol{dx}$ be the grid-normalized coordinate of $\boldsymbol{x}$. The multi-index $\boldsymbol{j}$ of the cell that contains the particle is obtained by taking the integer part of the grid-normalized coordinate: $\boldsymbol{j} = \lfloor \boldsymbol{y} \rfloor$. Similarly the weights $\boldsymbol{\alpha} \in [0, 1[^n$ for the n-linear interpolation are obtained by extracting the fractional part: $\boldsymbol{\alpha} = \boldsymbol{y} - \lfloor \boldsymbol{y} \rfloor = \boldsymbol{y} - \boldsymbol{j}$. The value of the velocity

at the new position $\boldsymbol{x}$ and time t^k is then obtained by the following formula:

$$\boldsymbol{u}^k(\boldsymbol{x}) = \sum_{\boldsymbol{d}\in\{0,1\}^n} \left(\prod_{i=1}^{n} \alpha_i^{d_i} (1-\alpha_i)^{1-d_i} \right) \boldsymbol{u}^k_{\boldsymbol{j}+\boldsymbol{d}} \tag{2.27}$$

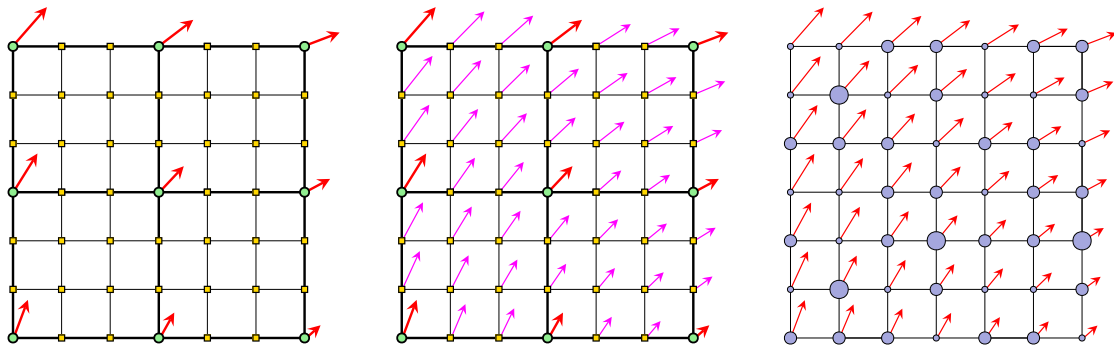
Figure 2.5 illustrates the two-dimensional interpolation of the velocity required for the second order Runge-Kutta method and corresponds to the case $\boldsymbol{j} = (0,0)$ and $\boldsymbol{\alpha} = (\alpha_y, \alpha_x) = (1/3, 1/3)$. The velocity at position $\boldsymbol{x} = \boldsymbol{j} + \boldsymbol{\alpha}$ is bilinearly interpolated using formula (2.27) that can be expanded to the following expression:

$$\boldsymbol{u}^k(\boldsymbol{x}) = (1-\alpha_y)(1-\alpha_x)\boldsymbol{u}^k_{0,0} + (1-\alpha_y)\alpha_x\boldsymbol{u}^k_{0,1} + \alpha_y(1-\alpha_x)\boldsymbol{u}^k_{1,0} + \alpha_y\alpha_x\boldsymbol{u}^k_{1,1}$$

In a multiscale approach, the velocity is known only on a coarser grid than that of the scalar and we have to use interpolation even in with the first order method. In this case we have $\mathcal{N}^c_\theta = \boldsymbol{m} \odot \mathcal{N}^c_{\boldsymbol{u}}$ where $\boldsymbol{m} \in \mathbb{N}^n$ denotes the grid ratio and we proceed in two steps:

1. First we interpolate the velocity from the coarse grid to the fine grid by using a given interpolation method. When the grid ratio between the velocity and the transported quantity is high, higher order interpolation schemes may be used (n-cubic splines or spectral interpolation, see section 2.2.2).

2. Now that the transported quantity θ and the velocity $\boldsymbol{u}$ are known on the same grid, particles are advected on the fine grid by using n-linear interpolation to get the velocity at intermediate particles positions.



Multigrid configuration $\mathcal{N}^c_\theta = 3\mathcal{N}^c_{\boldsymbol{u}}$ | Interpolation of velocity from coarse to fine grid | Advection on the fine grid

Figure 2.6 – Example of multiscale advection by using bilinear interpolation

Remeshing step

Once the particles have been advected, they have to be remeshed onto the grid by using a remeshing kernel W_ε, also called remeshing formula. The multidimensional case is generally

treated by tensor product of one-dimensional kernels $W_\varepsilon(\boldsymbol{x}) = \prod_{i=1}^{n} W_\varepsilon(x_i)$. Those formulas usually have a high arithmetic cost, especially for high order remeshing formula that exhibit large supports. Figure 2.7 illustrates the advection and remeshing procedure applied to a single particle marked with a blue cross. Initially particles are grid-aligned and their size represent the value they are carrying. After the displacement of the particles according to the velocity field represented by the thick red arrows, the particles are remeshed onto neighboring grid points:

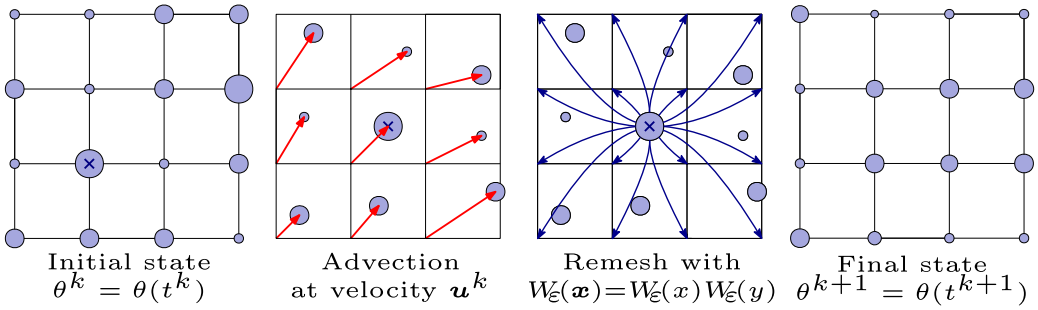


Figure 2.7 – Example of two-dimensional remeshing by tensor product

In this case, a single particle is remeshed onto $S = 16$ surroundings grid location (represented by the fine blue arrows) and the remeshing formula has a support $\boldsymbol{s}$ of $2\boldsymbol{dx}$ ($2\ dx_i$ in each direction i), or simply 2 in grid-normalized coordinates of the particle $\boldsymbol{y}_p = (\boldsymbol{x}_p - \boldsymbol{x}_{min}) \oslash \boldsymbol{dx}$. If we simplify equation (2.19) by using the grid normalized coordinates and grid-independent kernels $W(\boldsymbol{y}) = W_\varepsilon(\boldsymbol{y} \odot \boldsymbol{dx})$ we obtain the following expression:

$$\theta_{\boldsymbol{i}}^{k+1} = \sum_{p=1}^{P} \theta_p^k \, W\left((\boldsymbol{x}_p^{k+1} - \boldsymbol{x}_{\boldsymbol{i}}) \oslash \boldsymbol{dx}\right) = \sum_{p=1}^{P} \theta_p^k \, W\left(\boldsymbol{y}_p^{k+1} - \underbrace{\boldsymbol{y}_{\boldsymbol{i}}}_{=\boldsymbol{i}}\right) \tag{2.28}$$

This expression can be simplified by taking into account the support $\boldsymbol{s}$ of the remeshing kernel. Let us denote $\mathbb{P}_{\boldsymbol{i}}^{\boldsymbol{k}}$ the set of index of the particles p that where advected inside the rectangular cuboid centered around $\boldsymbol{i}$ with size $2\boldsymbol{s}$. Equation (2.28) can be rewritten as:

$$\theta_{\boldsymbol{i}}^{k+1} = \sum_{p \in \mathbb{P}_{\boldsymbol{i}}^{\boldsymbol{k}}} \theta_p^k \, W\left(\boldsymbol{y}_p^{k+1} - \boldsymbol{i}\right) \tag{2.29a}$$

$$\mathbb{P}_{\boldsymbol{i}}^{\boldsymbol{k}} = \left\{ p \in [\![1, P]\!] \mid \left(\boldsymbol{y}_p^{k+1} - \boldsymbol{i}\right) \in \,]-s_1, s_1[\times \cdots \times]-s_n, s_n[\right\} \tag{2.29b}$$

A given particle of global index p, associated to multi-index $\boldsymbol{j}$ such that $p = \mathcal{I}(\boldsymbol{j})$, has an initial grid-aligned grid-normalized coordinate $\boldsymbol{y}_p^k = \boldsymbol{y}_{\boldsymbol{j}}^k = \boldsymbol{j}$. In this configuration we can express the ball that contains the position of the particle after advection:

$$\boldsymbol{y}_{\boldsymbol{j}}^{k+1} \in \left\{ \boldsymbol{y} \;\middle|\; \|\boldsymbol{y} - \boldsymbol{j}\|_2 \leq dt \max_{\boldsymbol{x} \in \Omega} \left(\|\boldsymbol{u}(\boldsymbol{x}) \oslash \boldsymbol{dx}\|_2\right) \right\} \subset \left\{ \boldsymbol{y} \;\middle|\; |y_i - j_i| \leq \frac{dt \, \|\boldsymbol{u}_i\|_\infty}{dx_i} \;\forall i \in [\![1, n]\!] \right\}$$

Let $\boldsymbol{d}_{\boldsymbol{j}}^k = \boldsymbol{y}_{\boldsymbol{j}}^{k+1} - \boldsymbol{y}_{\boldsymbol{j}}^k$ be the grid-normalized displacement of a given particle $p_{\boldsymbol{j}}$ and $\boldsymbol{c} = (c_1, \cdots, c_n)$ with $c_i = s_i + \lfloor dt \, \|\boldsymbol{u}_i\|_\infty / dx_i \rfloor$. Equation (2.29) can be rewritten by using directly

the neighbor particle coordinates and displacements:

$$\begin{aligned}\theta_{\boldsymbol{i}}^{k+1} &= \sum_{j_1=-c_1}^{c_1} \cdots \sum_{j_n=-c_n}^{c_n} \theta_{\boldsymbol{i}+\boldsymbol{j}}^{k}\, W\left(\boldsymbol{y}_{\boldsymbol{i}+\boldsymbol{j}}^{k+1} - \boldsymbol{i}\right) \\ &= \sum_{j_1=-c_1}^{c_1} \cdots \sum_{j_n=-c_n}^{c_n} \theta_{\boldsymbol{i}+\boldsymbol{j}}^{k}\, W\left(\boldsymbol{d}_{\boldsymbol{i}+\boldsymbol{j}}^{k} + \boldsymbol{j}\right)\end{aligned} \tag{2.30}$$

In practice we only need to store the particle displacements $\boldsymbol{d}_{\boldsymbol{i}}^k$ and the remeshing procedure goes the other way arround: each advected particle is remeshed on the $S = \prod_{i=1}^n s_i$ surrounding grid locations as illustrated on figures 2.7, 2.8 and in algorithm 5.

Input : Input scalar field $\theta^{k+1} \in \mathbb{R}^{\mathcal{N}_1^v \times \cdots \times \mathcal{N}_n^v}$
Input : Grid-normalized particle displacement field $\boldsymbol{d} \in \left(\mathbb{R}^{\mathcal{N}_1^v \times \cdots \times \mathcal{N}_n^v}\right)^n$
Output: Output scalar field $\theta^k \in \mathbb{R}^{(\mathcal{N}_1^v + 2c_1) \times \cdots \times (\mathcal{N}_n^v + 2c_n)}$
for $\boldsymbol{i} \in [\![-c_1, \mathcal{N}_1^v + c_1[\![\times \cdots \times [\![-c_n, \mathcal{N}_n^v + c_n[\![$ **do**
  $\theta_{\boldsymbol{i}}^{k+1} \leftarrow 0$
end
for $\boldsymbol{i} \in [\![0, \mathcal{N}_1^v[\![\times \cdots \times [\![0, \mathcal{N}_n^v[\![$ **do**
  $\boldsymbol{j} \leftarrow \boldsymbol{i} + \lfloor \boldsymbol{d}_{\boldsymbol{i}} \rfloor$
  $\boldsymbol{\alpha} \leftarrow \boldsymbol{d}_{\boldsymbol{i}} - \lfloor \boldsymbol{d}_{\boldsymbol{i}} \rfloor$
  for $\boldsymbol{p} \in [\![-s_1 + 1, s_1]\!] \times \cdots \times [\![-s_n + 1, s_n]\!]$ **do**
    $\theta_{\boldsymbol{j}+\boldsymbol{p}}^{k+1} \leftarrow \theta_{\boldsymbol{j}+\boldsymbol{p}}^{k+1} + \theta_{\boldsymbol{i}}^{k} \underbrace{W(\boldsymbol{p} - \boldsymbol{\alpha})}_{= \prod_{l=1}^{n} W(p_l - \alpha_l)}$
  end
end

Algorithm 5: Remeshing procedure using a remeshing kernel W with support $\boldsymbol{s}$

The inner loop of algorithm 5 computes the contribution of particle $\boldsymbol{p}_{\boldsymbol{i}}$ carrying quantity $\theta_{\boldsymbol{i}}^k$ to the S grid locations present in the neighborhood of its new position $\boldsymbol{y}_{\boldsymbol{i}}^{k+1} = \boldsymbol{i} + \boldsymbol{d}_{\boldsymbol{i}}^k = \boldsymbol{j} + \boldsymbol{\alpha}$ as illustrated on figure 2.8 for the one-dimensional case. The values that are remeshed in the boundary cells, outside of the domain, are handled depending on the boundary conditions as a post-processing step.

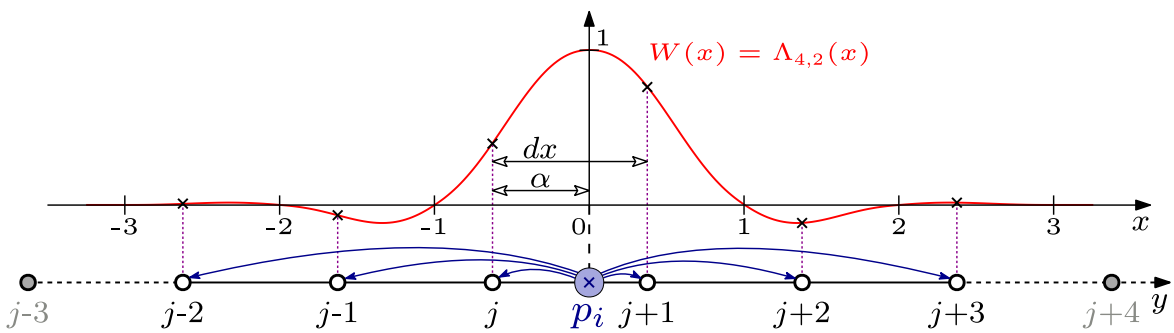


Figure 2.8 – One-dimensional remeshing kernel with support $s = [-3, 3]$

Multi-dimensional remeshing is generally achieved by tensor product of one-dimensional kernels, resulting in S calls to the one-dimensional remeshing formula per particle. We will see that the evaluation of W implies to evaluate high order polynomials, and thus the remeshing procedure through tensor product has high arithmetic intensity. It is possible to reduce the number of remeshing kernel calls by performing a directional splitting trough the use of a Strang splitting as proposed by [Magni et al. 2012]. This is presented in section 2.5.1.

2.3.2 Explicit Runge-Kutta methods

The accuracy in time can be improved in the advection step by using more ellaborate time stepping schemes such as the explicit Runge-Kutta methods [Kutta 1901]. The most widely known scheme of Runge-Kutta family is generally referred to as `RK4` or the Runge-Kutta method. Here for simplicity and without any loss in generality and consider that f is scalar valued. We consider the following initial value problem $\dfrac{\partial f}{\partial t} = F(t, f)$ with $f(t_k) = f^k$ and we seek to find $f^{k+1} = f(t_k + dt)$.

The fourth-order (in time) Runge-Kutta scheme is obtained as the following:

$$f^{k+1} = f^k + \frac{dt}{6}\left(F_1^k + 2F_2^k + 2F_3^k + F_4^k\right) \tag{2.31a}$$

$$F_1^k = F(t^k, f^k) \tag{2.31b}$$

$$F_2^k = F\left(t^k + \frac{dt}{2}, f^k + \frac{dt}{2}F_1^k\right) \tag{2.31c}$$

$$F_3^k = F\left(t^k + \frac{dt}{2}, f^k + \frac{dt}{2}F_2^k\right) \tag{2.31d}$$

$$F_4^k = F\left(t^k + dt, f^k + dt\ F_3^k\right) \tag{2.31e}$$

Here f^{k+1} is the `RK4` approximation of $f(t^{k+1})$ at order $p = 4$. It is determined by the current value plus the weighted average of four estimated slopes F_i multiplied by the timestep dt. The four slopes are estimated by evaluating F at different intermediate estimations (t_i, f_i) such that $t_i \in \{t^k, t^k + \frac{dt}{2}, t^k + dt\}$:

- F_1^k is the initial slope $F(t^k, f^k)$ as seen before with the Euler method.
- F_2^k is the slope at the midpoint $t^k + \frac{1}{2}dt$ based on initial slope F_1^k.
- F_3^k is the slope at the midpoint $t^k + \frac{1}{2}dt$ based on the second slope F_2^k.
- F_4^k is the slope at the endpoint $t^k + dt$ based on the third slope F_3^k.

The final slope is obtained by giving greater weights to the slopes obtained at the midpoints. The family of explicit Runge-Kutta methods is simply a generalization of the `RK4` method

presented above. An explicit scheme of s stages is given by the following expressions:

$$f^{k+1} = f^k + dt \sum_{i=1}^{s} b_i F_i^k \tag{2.32a}$$

$$F_1^k = F\left(t^k, f^k\right) \tag{2.32b}$$

$$F_2^k = F\left(t^k + c_1 dt, f^k + a_{1,1} F_1^k\right) \tag{2.32c}$$

$$F_3^k = F\left(t^k + c_2 dt, f^k + a_{2,1} F_2^k + a_{2,2} F_1^k\right) \tag{2.32d}$$

$$\vdots \tag{2.32e}$$

$$F_s^k = F\left(t^k + c_{s-1} dt, f^k + \sum_{i=1}^{s-1} a_{s-1,i} F_i^k\right) \tag{2.32f}$$

All the required coefficients can be summed up in a Butcher tableau [Butcher 1963]:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

Butcher tableau for RK4

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_1 & a_{11} & & & & \\ c_2 & a_{21} & a_{22} & & & \\ \vdots & \vdots & & \ddots & & \\ c_{s-1} & a_{s-1,1} & a_{s-1,2} & \cdots & a_{s-1,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array}$$

General Butcher tableau

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{3} & \frac{1}{3} & & & \\ \frac{2}{3} & -\frac{1}{3} & \frac{1}{2} & & \\ 1 & 1 & -1 & 1 & \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array}$$

Butcher tableau for RK4_38

The method is consistent under $\sum_{j=1}^{i} a_{i,j} = c_i$ for all $i \in [\![1, s-1]\!]$ and a method of order p has at least s stages. The only consistent explicit Runge-Kutta method with one stage is the Euler method. In practice we will use methods with up to four stages, denoted RK1 (Euler), RK2, RK3 and RK4 which are the classical Runge-Kutta methods complemented with RK4_38 which is a variation of the fourth-order scheme called the 3/8-rule [Butcher et al. 1987]. All those methods offer the same accuracy in time as their respective number of stages ($p = s$). This is not possible with methods using more than four stages. The primary advantage the RK4_38 method is that almost all of the error coefficients are smaller for the price of some extra FLOPs and memory.

As those methods are explicit, there is no need to solve any linear system and the total cost of the method is due to the s evaluations of F along with $2s + k - 1$ additions and $2s + k$ multiplications where $s - 1 \leq k \leq s(s+1)/2$ represent the number of non-zero coefficients a_{ij}. Those same Runge-Kutta schemes will also be used for diffusion and stretching in a finite differences framework. In this case f represent a space dependent field, and the cost of the method is hidden in the s evaluations of F where partial space derivatives of f can be obtained by applying centered finite difference stencils.

2.3.3 Construction of the remeshing formulas

This subsection is dedicated to the construction of one-dimensional remeshing kernels $W(x)$. Due to their nature, particle methods are suitable for solving conservation equations. It is therefore important that the remeshing step, performed in equation (2.19), enforce those conservation properties. In particular, remeshing formula should be designed to preserve the moments of the quantities transported. In dimension one, the discrete moment of order $m \in \mathbb{N}$ of the transported field θ is defined by $\sum_{i=0}^{\mathcal{N}_x^v-1} x_i^m \, \theta_i$.

Therefore, a remeshing kernel must satisfy the conservation equalities of the following discrete moments up to the p-th moment:

$$\sum_{i=0}^{\mathcal{N}_x^v-1} x_i^m \, \theta_i^k = \sum_{i=0}^{\mathcal{N}_x^v-1} x_i^m \, \theta_i^{k+1} \quad \forall m \in [\![0, p]\!] \tag{2.33}$$

where p is directly linked to the spatial order of the method (and the support s of the kernel).

By using equation (2.30) with $n = 1$, equation (2.33) is satisfied under:

$$\sum_{j \in \mathbb{Z}} j^m W(x-j) = x^m \quad \forall x \in \mathbb{R} \quad \forall m \in [\![0, p]\!] \tag{2.34}$$

It also has to be symmetric $W(x) = W(-x)$ so that there is no preferred remeshing direction. (this implies that the odd moments are all zero). As W approximates a Dirac, it is preferable that it satisfies the interpolation condition $W(j) = \delta_{j,0} \; \forall j \in \mathbb{Z}$ which is compatible with the 0-th moment equation that states that the global scalar quantity is preserved during the transport $\sum_{j \in \mathbb{Z}} W(x-j) = 1$. The interpolation property allows an exact conservation of remeshed quantities carried by particles that happen to be grid-aligned (particles whose position coincides with a grid point after advection). Thus, at zero-velocity, the scalar field θ remains the same.

A first family of kernels can be obtained by looking for functions with support $p+1$ built with $p+1$ piecewise polynomials of order p (with support 1). The coefficients of the polynomials are obtained enforcing the conservation of the p first moments with equations (2.34). Kernels arising from this family are not regular enough and some of them are not even continuous. This entails a loss of numerical precision [Cottet et al. 2009b]. A second family of kernels that are more regular can be obtained up by the use of regular B-splines [Schoenberg 1988]. This family does not enforce the interpolation condition and offers up to second order moment conservation for a given regularity r. The lack of high order moment preservation properties of this family have been later compensated by using Richardson extrapolation while preserving their regularity [Monaghan 1985]. This family has been used since the end of the 90's [Koumoutsakos 1997][Salihi 1998] in the context of semi-lagrangian methods. However, high order kernels obtained by this method also lack the interpolation property.

The present method will use the family high order kernels proposed by [Cottet et al. 2014] that allows to obtain a remeshing kernel of any order p and any regularity r while preserving

the interpolation property. The kernels are found as with the first family, by looking for piecewise polynomials with compact support. Those kernels satisfy the following properties:

- **P1**: Parity $W(x) = W(-x)$
- **P2**: Compact support $[-s, s]$ with $s \in \mathbb{N}$
- **P3**: Piecewise polynomial of degree q on the integer subsets.
- **P4**: Overall regularity r.
- **P5**: Preservation of the p first moments.
- **P6**: Interpolation property $W(j) = \delta_{j0}\ \forall j \in \mathbb{Z}$

The three first properties restrict the problem to find s polynomials of degree q yielding $s(q+1)$ unknowns. Because the odd moment are automatically preserved by the parity of the kernel, we only take into account an even number moments $p \in 2\mathbb{N}$. Choosing support $s = 1 + p/2$ and polynomials of degree $q = 2r + 1$ with $r \geq p/2$ give enough equations to solve the linear system satisfying all the properties. The kernels obtained by this method are named $\Lambda_{p,r}$ and some of them are listed in the following table:

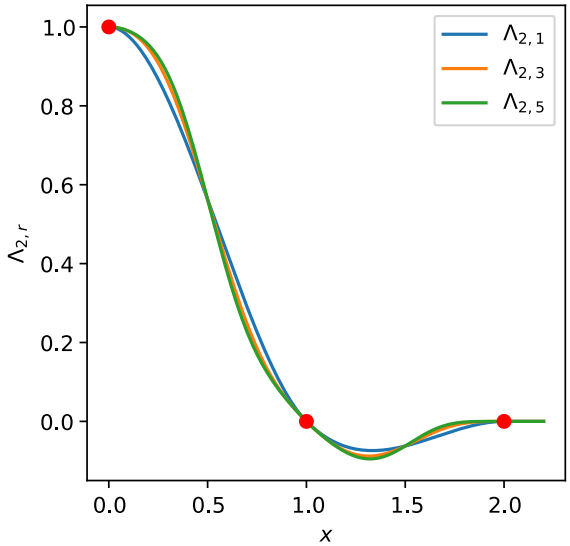
Kernel	**Max. preserved moment**	**Regularity**	**Support**	**Degree**
$\Lambda_{p,r}$	$p \in 2\mathbb{N}$	$C^r, r \geq p/2$	$[-s, s]$, $s = 1 + p/2$	$q = 2r + 1$
$\Lambda_{2,1}$	2	C^1	$[-2, 2]$	3
$\Lambda_{2,2}$	2	C^2	$[-2, 2]$	5
$\Lambda_{4,2}$	4	C^2	$[-3, 3]$	5
$\Lambda_{4,4}$	4	C^4	$[-3, 3]$	9
$\Lambda_{6,4}$	6	C^4	$[-4, 4]$	9
$\Lambda_{6,6}$	6	C^6	$[-4, 4]$	13
$\Lambda_{8,4}$	8	C^4	$[-5, 5]$	9
$\Lambda_{8,6}$	8	C^6	$[-5, 5]$	13

Table 2.1 – Comparative of some high order semi-lagrangian remeshing kernels

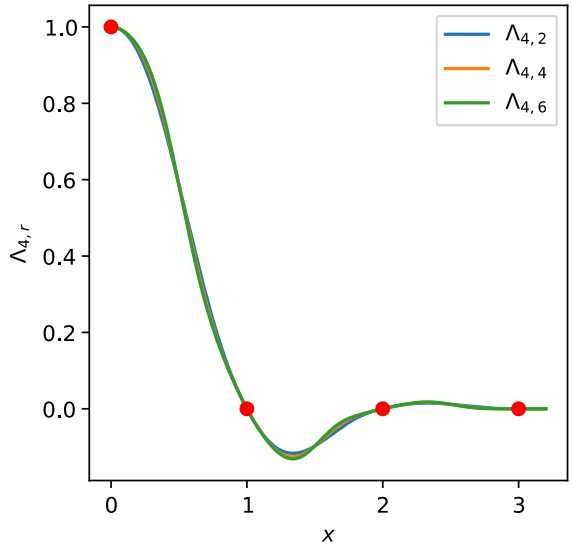
The default remeshing kernel that we will use in this work is $\Lambda_{4,2}$:

$$\Lambda_{4,2}(x) = \begin{cases} 1 - \frac{5}{4}|x|^2 - \frac{35}{12}|x|^3 + \frac{21}{4}|x|^4 - \frac{25}{12}|x|^5 & \text{if } 0 \leq |x| < 1 \\ -4 + \frac{75}{4}|x| - \frac{245}{8}|x|^2 + \frac{545}{24}|x|^3 - \frac{63}{8}|x|^4 + \frac{25}{24}|x|^5 & \text{if } 1 \leq |x| < 2 \\ 18 - \frac{153}{4}|x| + \frac{255}{8}|x|^2 - \frac{313}{24}|x|^3 + \frac{21}{8}|x|^4 - \frac{5}{24}|x|^5 & \text{if } 2 \leq |x| < 3 \\ 0 & \text{if } |x| \geq 3 \end{cases} \tag{2.35}$$

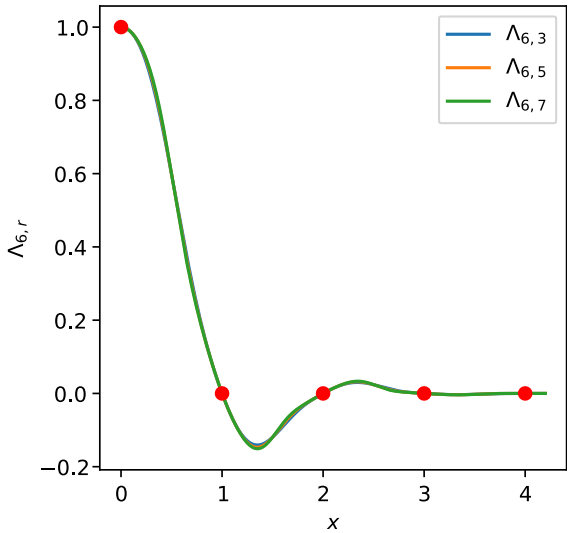
This specific formula is illustrated on figure 2.8. Other remeshing formulas can be found in appendix A of [Etancelin 2014] and some of them are illustrated on figure 2.9.



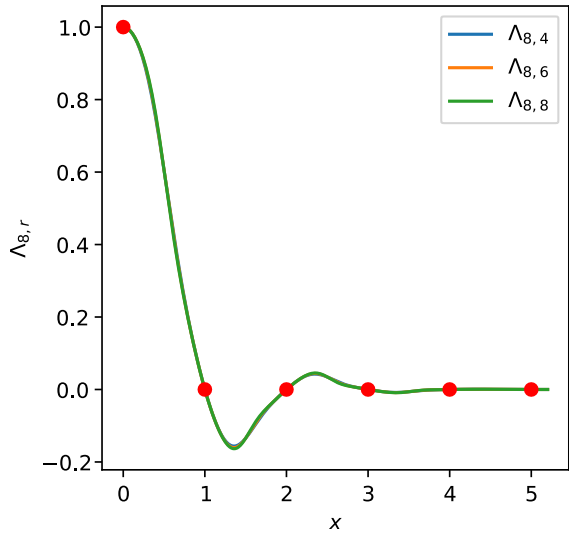
(a) $\Lambda_{2,r}$



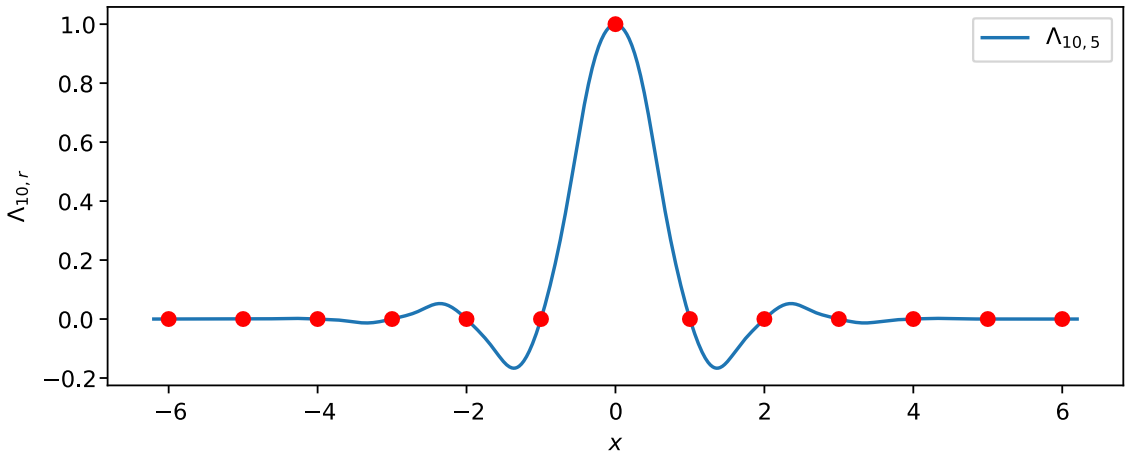
(b) $\Lambda_{4,r}$



(c) $\Lambda_{6,r}$



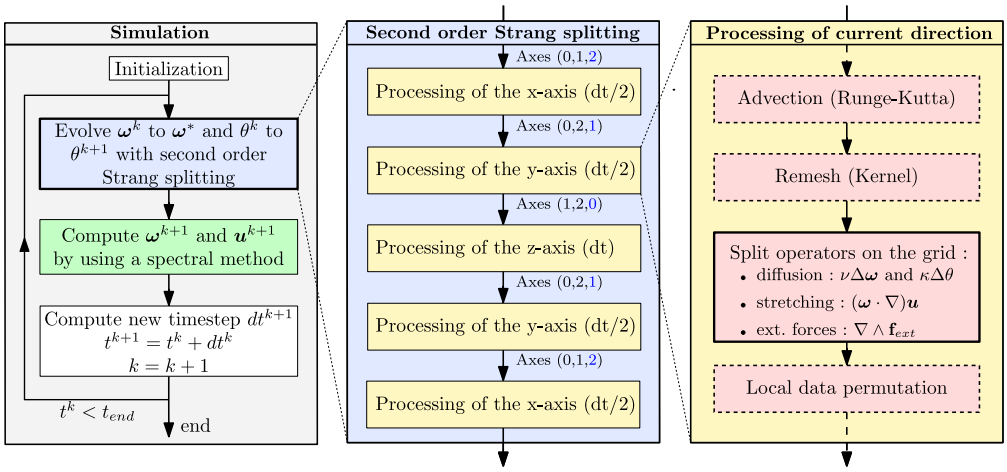
(d) $\Lambda_{8,r}$



(e) $\Lambda_{10,r}$

Figure 2.9 – Illustration of some semi-lagrangian remeshing kernels $\Lambda_{p,r}$

2.4 Finite difference methods



Finite-difference methods (FDM) are numerical methods for solving differential equations by approximating them with difference equations, in which finite differences approximate the derivatives [Smith et al. 1985]. Let $f : \mathbb{R} \to \mathbb{R}$ be the function whose derivatives are to be approximated, such that f is p times continuously differentiable, $f \in C^p(\mathbb{R}, \mathbb{R})$. Then by Taylor's theorem, we can write a Taylor series expansion around point $x \in \mathbb{R}$:

$$f(x+h) = \sum_{q=0}^{p} \frac{f^{(q)}(x)}{q!} h^q + R_p(h) \tag{2.36}$$

where $R_p(x) = \mathcal{O}\left(h^{p+1}\right)$ is a remainder term, resulting of the difference between the Taylor polynomial of degree p and the original function f. It is then possible to extract the target derivative $f^{(d)}(x)$ with $d \leq p$ in terms of elements in $f(x + \mathbb{Z}h)$ by building a linear system consisting of $s = i_{max} - i_{min} + 1$ Taylor series expanded around $x + ih$ at order p for $i \in [\![i_{min}, i_{max}]\!] \subset \mathbb{Z}$. If we denote $\boldsymbol{f}_x = (f(x+ih))_{i\in[\![i_{min},i_{max}]\!]}$ then the d-th derivative of f evaluated at x, can be approximated by $h^{-d}(\boldsymbol{S}\cdot\boldsymbol{f}_x)$ where $\boldsymbol{S} \in \mathbb{Q}^s$ is the associated finite difference stencil. The resulting order of the stencil depends on the order of the Taylor series p, the approximated derivative d and how errors terms cancel each other out. When $i_{min} = -i_{max}$ the finite difference scheme is said to be centered, $i_{max} = 0$ yields a backward finite difference scheme and $i_{min} = 0$ a forward finite differences scheme.

The centered finite difference stencils approximating $f^{(d)}$ with accuracy m contains $s = 2q + 1$ coefficients $\boldsymbol{S}$ with $q = \left\lfloor \frac{d+1}{2} \right\rfloor + \left\lfloor \frac{m+1}{2} \right\rfloor$. The s coefficients of the stencil $(S_{-q}, S_{-q+1}, \cdots, S_0, \cdots, S_{q-1}, S_q)$ can be computed by solving the following linear system:

$$\begin{pmatrix} (-q)^0 & (-q+1)^0 & \cdots & (q-1)^0 & q^0 \\ (-q)^1 & (-q+1)^1 & \cdots & (q-1)^1 & q^1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ (-q)^d & (-q+1)^d & \cdots & (q-1)^d & q^d \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ (-q)^{2q} & (-q+1)^{2q} & \cdots & (q-1)^{2q} & q^{2q} \end{pmatrix} \begin{pmatrix} S_{-q} \\ S_{-q+1} \\ \cdots \\ S_{-q+d} \\ \cdots \\ S_q \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \cdots \\ d! \\ \cdots \\ 0 \end{pmatrix} \tag{2.37}$$

The following table contains the coefficients of the central differences obtained for the first and second derivative with several orders of accuracy for a uniform grid spacing h:

`derivative`	`accuracy`	S_{-3}	S_{-2}	S_{-1}	S_0	S_1	S_2	S_3
1	2			$-1/2$	0	$1/2$		
1	4		$1/12$	$-2/3$	0	$2/3$	$-1/12$	
1	6	$-1/60$	$3/20$	$-3/4$	0	$3/4$	$-3/20$	$1/60$
2	2			1	-2	1		
2	4		$-1/12$	$4/3$	$-5/2$	$4/3$	$-1/12$	
2	6	$1/90$	$-3/20$	$3/2$	$-49/18$	$3/2$	$-3/20$	$1/90$

Table 2.2 – Finite differences coefficients for first and second derivatives

For example, table 2.2 states that the first and second derivatives of f at order 4 at point x can be approximated by:

$$f^{(1)}(x) = \frac{1}{h^2}\left[+\frac{1}{12}f(x-2h) - \frac{2}{3}f(x-h) + \frac{2}{3}f(x+h) - \frac{1}{12}f(x+2h)\right] + \mathcal{O}\left(h^4\right) \tag{2.38a}$$

$$f^{(2)}(x) = \frac{1}{h^2}\left[-\frac{1}{12}f(x-2h) + \frac{4}{3}f(x-h) - \frac{5}{2}f(x) + \frac{4}{3}f(x+h) - \frac{1}{12}f(x+2h)\right] + \mathcal{O}\left(h^4\right) \tag{2.38b}$$

For a fourth order accuracy stencil, the local truncation error is proportional to h^4.

2.4.1 Spatial derivatives

Finite difference coefficients can be extended to any function $f(\mathbb{R}^n, \mathbb{R}) \in C^p(\mathbb{R}^n, \mathbb{R})$ and can be computed on arbitrarily spaced grids [Fornberg 1988]. As part of this work we will remain in the uniform grid spacing case for discretization in space as presented in subsection 2.2. We will also approximate derivatives with at least fourth order accuracy to prevent odd-even decoupling which is a discretization error that can occur on collocated grids and which leads to checkerboard patterns in the solution [Harlow et al. 1965].

Let F be the discretization of a scalar field $f : \mathcal{B} \to \mathbb{R}$ on the cuboid domain $\mathcal{B} \in \mathbb{R}^n$ with global grid size $\boldsymbol{\mathcal{N}}^v$ containing a total of $N = \prod_{i=1}^n \mathcal{N}_i^v$ elements and let $S \in \mathbb{Q}^{s_1} \times \cdots \mathbb{Q}^{s_n}$ be a centered stencil of size $\boldsymbol{s} = 2\boldsymbol{q} + 1$ containing $k \leq \prod_{i=1}^n s_i$ non-zero coefficients that approximate some mixed spatial derivative of f on the grid at order m. The approximation of the derivative D over the whole domain is obtained by performing a convolution between the stencil S and the grid data $F = \{F_{\boldsymbol{i}} = f(\boldsymbol{x}_{min} + \boldsymbol{i} \odot \boldsymbol{dx}) \mid \boldsymbol{i} \in [\![1, \mathcal{N}_1^v[\![\times \cdots [\![0, \mathcal{N}_n^v[\![\}$ containing additional boundary values that we will later call ghost nodes (see algorithm 6). This operation costs $(k-1)N$ additions and kN multiplications and can thus be computed in

$\mathcal{O}(kN)$ operations.

Input : n-dimensional centered stencil $S \in \mathbb{R}^{s_1 \times \cdots \times s_n}$ with $\boldsymbol{s} = 2\boldsymbol{q} + 1$
Input : n-dimensional array $F \in \mathbb{R}^{(\mathcal{N}_1^v + 2q_1) \times \cdots \times (\mathcal{N}_n^v + 2q_n)}$
Output: n-dimensional array $D = F * S \in \mathbb{R}^{\mathcal{N}_1^v \times \cdots \times \mathcal{N}_n^v}$
for $\boldsymbol{i} \in [\![0, \mathcal{N}_1^v[\![\times \cdots \times [\![0, \mathcal{N}_n^v[\![$ **do**
  $D_{\boldsymbol{i}} \leftarrow 0$
  for $\boldsymbol{j} \in [\![-q_1, +q_1]\!] \times \cdots \times [\![-q_n, +q_n]\!]$ **do**
    if $S_{\boldsymbol{j}} \neq 0$ **then**
      $D_{\boldsymbol{i}} \leftarrow D_{\boldsymbol{i}} + S_{\boldsymbol{j}} F_{\boldsymbol{q}+\boldsymbol{i}+\boldsymbol{j}}$
    end
  end
end

Algorithm 6: Convolution of a discretized field F with a centered stencil S of size $\boldsymbol{s} = 2\boldsymbol{q} + 1$. F is discretized on a grid of size $\boldsymbol{\mathcal{N}}^v + 2\boldsymbol{q}$, $\boldsymbol{q}$ being the size of additional required boundary layers for input. The n-dimensional stencil S is indexed as in equation (2.37) whereas input and output arrays have indexes starting at zero. The additional inner loop conditional is here to highlight the fact that S may be sparse.

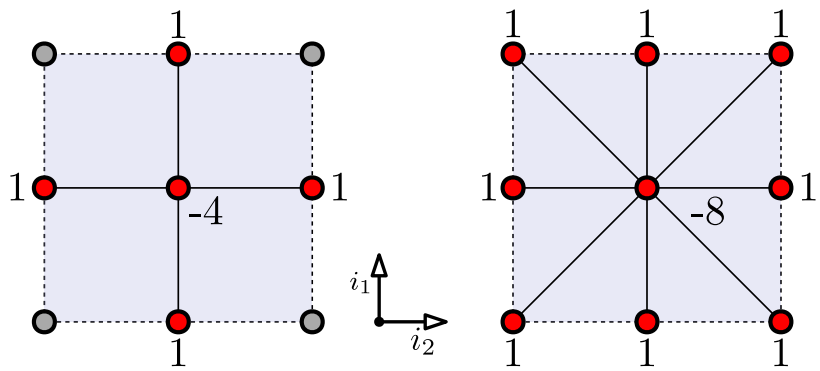


Figure 2.10 – Stencils approximating the 2D Laplacian: Examples of five-point and nine-point centered stencils that approximate the Laplacian operator $\Delta = \dfrac{\partial}{\partial x^2} + \dfrac{\partial}{\partial y^2}$ on a uniform grid with second order accuracy (leading error coefficients are however not the same).

Stencils can in general represent more than simple derivatives, it can for example approximate sums of derivatives as shown on figure 2.10. The mask of non-zero coefficients of a given stencil, represented in red, gives the location of values that have to be read to compute one value of the output, located at the center. The number of non-zero coefficients k of a finite difference stencil is typically small. Although dense centered stencils such as the second order nine-point Laplacian stencil have $(2q+1)^n$ non-zero coefficients, they are seldomly used for fourth-order accuracy because of performance reasons. In 3D, the fourth order accuracy dense Laplacian stencil requires $(2 * 2 + 1)^3 = 125$ input values, 125 multiplications and 124 additions to compute one single output value. Assuming the values are represented with double precision floating point numbers of 8 bytes each, and that the multiplications and additions are done by using fused multiply-add operations, this gives an arithmetic intensity of $125/8 \simeq 15.6$ `FLOP/B`.

Nowadays, most common architectures like CPUs and GPUs are optimized to target 5 FLOP/B for double precision computations [Rupp 2013]. A higher FLOP per byte ratio indicates that the given algorithms fully saturate the hardware compute units [Williams et al. 2009]. In this case, this means that this stencil would, in average, be compute-bound by a factor of 3 or put in other words that it would only use at most one third of the available memory bandwidth for an given device. Sparse stencils, such as the one obtained by sum or tensor product of 1D stencils, have the order of $1 + 2qn$ non-zero coefficients and thus are memory-bound. The same calculation for the sparse fourth-order accuracy 3D Laplacian stencil yields an arithmetic intensity of $8/8 = 1$ FLOP/B. Hence sparse stencils are likely to to be memory bound.

2.4.2 Finite difference schemes

A finite difference scheme can be used to approximate the solution of a partial differential equation such as (2.5), (2.6) and (2.7). It is obtained by approximating both space and time using finite differences. In this section we will focus on the two-dimensional diffusion equation as encountered in equation (2.5). The simplest temporal integration scheme is the explicit Euler scheme. It can be directly obtained by taking the Taylor expansion of $\boldsymbol{\omega}(\boldsymbol{x}, t)$ around $t^k + dt$ at order $p = 1$ with gives:

$$\frac{\partial \omega}{\partial t}(t = t^k) = \frac{\omega^{k+1} - \omega^k}{dt^k} + \mathcal{O}\left(dt\right) \tag{2.39}$$

By using equations (2.38b) and (2.39) applied to $\boldsymbol{\omega}(t = t^k)$ along with equation (2.5) we obtain:

$$\begin{aligned} \frac{\omega^{k+1}_{i_1,i_2} - \omega^k_{i_1,i_2}}{dt} + \mathcal{O}\left(dt\right) = \\ +\frac{\nu}{dx^2}\left[-\frac{1}{12}\boldsymbol{\omega}_{i_1,i_2-2} + \frac{4}{3}\boldsymbol{\omega}_{i_1,i_2-1} - \frac{5}{2}\boldsymbol{\omega}_{i_1,i_2} + \frac{4}{3}\boldsymbol{\omega}_{i_1,i_2+1} - \frac{1}{12}\boldsymbol{\omega}_{i_1,i_2+2}\right] + \mathcal{O}\left(dx^4\right) \\ +\frac{\nu}{dy^2}\left[-\frac{1}{12}\boldsymbol{\omega}_{i_1-2,i_2} + \frac{4}{3}\boldsymbol{\omega}_{i_1-1,i_2} - \frac{5}{2}\boldsymbol{\omega}_{i_1,i_2} + \frac{4}{3}\boldsymbol{\omega}_{i_1+1,i_2} - \frac{1}{12}\boldsymbol{\omega}_{i_1+2,i_2}\right] + \mathcal{O}\left(dy^4\right) \end{aligned} \tag{2.40}$$

from which we can easily express the value of the variable at the next timestep $\omega^{k+1}_{i_1,i_2}$:

$$\begin{aligned} \omega^{k+1}_{i_1,i_2} = \omega^k_{i_1,i_2} \\ +\frac{\nu dt}{dx^2}\left[-\frac{1}{12}\boldsymbol{\omega}_{i_1,i_2-2} + \frac{4}{3}\boldsymbol{\omega}_{i_1,i_2-1} - \frac{5}{2}\boldsymbol{\omega}_{i_1,i_2} + \frac{4}{3}\boldsymbol{\omega}_{i_1,i_2+1} - \frac{1}{12}\boldsymbol{\omega}_{i_1,i_2+2}\right] \\ +\frac{\nu dt}{dy^2}\left[-\frac{1}{12}\boldsymbol{\omega}_{i_1-2,i_2} + \frac{4}{3}\boldsymbol{\omega}_{i_1-1,i_2} - \frac{5}{2}\boldsymbol{\omega}_{i_1,i_2} + \frac{4}{3}\boldsymbol{\omega}_{i_1+1,i_2} - \frac{1}{12}\boldsymbol{\omega}_{i_1+2,i_2}\right] \\ +\mathcal{O}\left(dx^4\right) + \mathcal{O}\left(dy^4\right) + \mathcal{O}\left(dt\right) \end{aligned} \tag{2.41}$$

This constitutes a fourth order scheme in space and first order in time finite differences scheme. It is illustrated on figure 2.11. Higher order in time can be achieved trough the use of an higher order Runge-Kutta scheme.

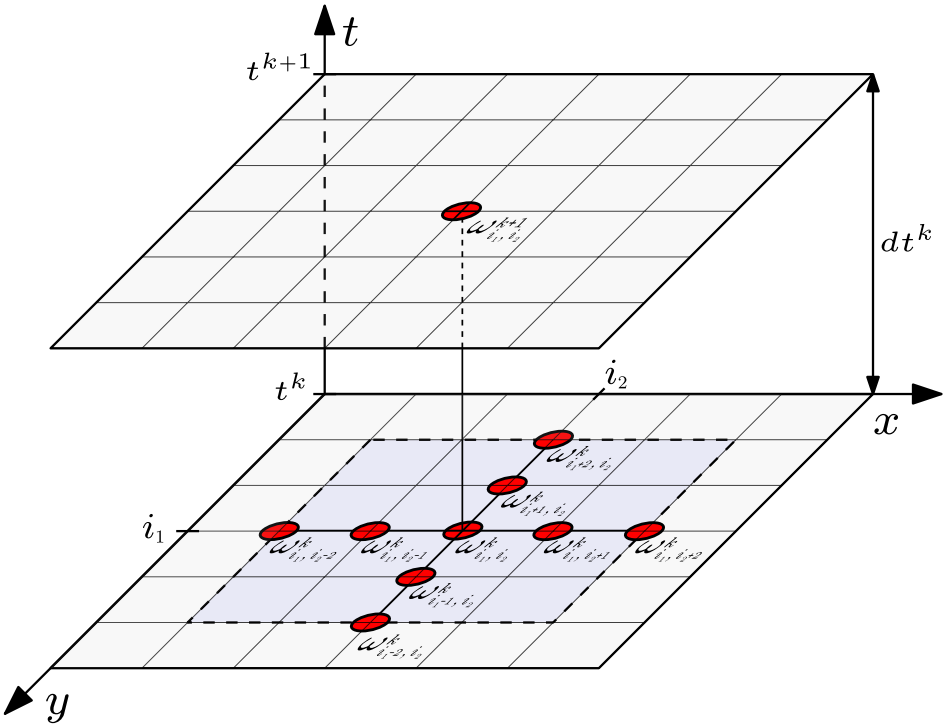


Figure 2.11 – Explicit 2D finite difference scheme used to solve the heat equation with first order accuracy in time and fourth order accuracy in space. This scheme requires one read and one write and a total of 9 multiplications and 8 additions, giving an operational intensity of about `0.56FLOP/B` in double precision by using fused multiply-add operations.

The major drawback of this explicit method is that the scheme in not unconditionally stable and for problems with large diffusivity the timestep restriction can be too severe. The von Neumann stability analysis [Crank et al. 1947] states that this scheme is stable under the following condition that restricts the timestep:

$$\max\left(\frac{\nu dt}{dx^2}, \frac{\nu dt}{dy^2}\right) \leq \frac{1}{2} \tag{2.42}$$

This explicit numerical scheme can be generalized in any dimension for all partial differential equations of the form:

$$\frac{\partial \boldsymbol{f}}{\partial t} = \boldsymbol{L}\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}^{\boldsymbol{k_1}}}{\partial \boldsymbol{x}^{\boldsymbol{k_1}}}, \cdots, \frac{\partial \boldsymbol{f}^{\boldsymbol{k_m}}}{\partial \boldsymbol{x}^{\boldsymbol{k_m}}}\right) \text{ with } m \in \mathbb{N} \text{ and } \boldsymbol{k}_i \in \mathbb{N}^n \text{ where } \frac{\partial \boldsymbol{f}^{\boldsymbol{k}}}{\partial \boldsymbol{x}^{\boldsymbol{k}}} = \frac{\partial \boldsymbol{f}^{k_1+\cdots+k_n}}{\partial x_1^{k_1} \cdots \partial x_n^{k_n}}$$

Equations (2.3), (2.5), (2.6) and (2.7) however do not contain any mixed spatial partial derivatives and fall into the following family of partial differential equations:

$$\frac{\partial \boldsymbol{f}}{\partial t} = \boldsymbol{L}\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}}{\partial x_1}, \cdots, \frac{\partial \boldsymbol{f}}{\partial x_n}, \frac{\partial \boldsymbol{f}^2}{\partial x_1^2}, \cdots, \frac{\partial \boldsymbol{f}^2}{\partial x_n^2}\right) \tag{2.43}$$

with $\boldsymbol{f} = [\boldsymbol{\omega}_1, \cdots, \boldsymbol{\omega}_p, \boldsymbol{u}_1, ..., \boldsymbol{u}_n, \theta]^T$.

2.4.3 Performance considerations

Achieving theoretical peak bandwidth or peak performance when dealing width the evaluation of three-dimensional (and higher dimensions) stencils is not an easy task [Datta et al. 2008]. This comes from the fact that when we estimate the number of operations per bytes we assume that the device has infinite cache such that each input value can be remembered after its first read. It is also assumed that data can be read and written at theoretical peak bandwidth, which may not be the case here because of the non-contiguous memory accesses. For dense stencils that require $\mathcal{O}(100)$ input values per output (fourth order Laplacian stencil in 3D), the data can simply not fit in cache requiring the implementation of stencil-dependent caching strategies. For sparse stencils the problem is by definition memory bound and memory accesses have to be optimized.

As stencil computation is a key part of many other high-performance computing applications, such as image processing and convolutional neural networks, numerous techniques have been developed to achieve optimal performance for multi-dimensional stencils:

- Rearrange the n-dimensional data to preserve the locality of the data points by using space-filling curves like Z-ordering instead of the linear row-major ordering [Nocentino et al. 2010]. Note that for `GPU`s, textures already provide space filling curves builtin for 2D and often 3D arrays [Li et al. 2003][Sugimoto et al. 2014]. This optimization technique is easiest to implement but do not provide enormous gains for general stencils.
- Manually tune the vectorization, memory accesses and how the data is cached with respect to the target architecture and the given stencil [Brandvik et al. 2010]. This has been the main approach for early `CUDA` and `OpenCL` implementations [Micikevicius 2009] [Su et al. 2013].
- As manual tuning requires a lot of work, the next idea was logically to develop frameworks, that for a given stencil, automatically generate optimized code for a target language and autotune its runtime parameters [Zhang et al. 2012] [Holewinski et al. 2012] [Cummins et al. 2015].
- Although the first solutions are all software based, there also exist hardware based solutions. It is for example possible to redesign the memory hardware to be optimized for stencil computations on `FPGA`s by using `FPGA`-compatible `OpenCL` platforms [Verma et al. 2016] [Wang et al. 2017][Zohouri et al. 2018].

The approach we will choose in this work is somehow orthogonal to what is usually done for finite differences based fluid solvers: we will not use any high dimensional stencils at all. In fact the absence of mixed spatial partial derivatives in equations (2.1c) and (2.1d) allows us to split all n-dimensional stencils that arrise from the approximation of the spatial derivatives to be split as efficient `1D` stencils. This offers numerous advantages:

- One-dimensional stencil optimizations like vectorization are easy to implement. As they only require contiguous memory accesses they maximize the cache hits.

- By implementing only 1D computations, the algorithm becomes dimension-agnostic. The support of n-dimensional problems is added by simply looping over the dimensions.
- This idea fits nicely into the directional splitting framework already introduced in section 2.1.3.

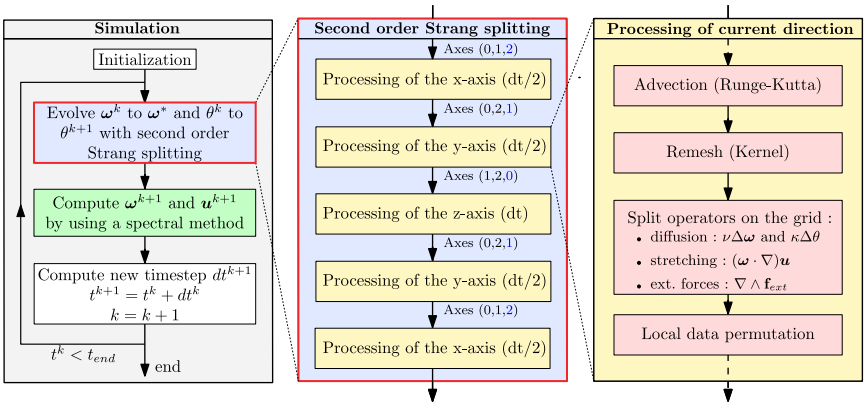
All the optimization aforementioned only work if the data is contiguous for all axes. This suppose that between the treatment of each direction, the data is rearranged in memory such that the next axis to be computed becomes contiguous in memory. This just shifts the performance bottleneck and the implementation effort on the permutations, that by definition do not compute anything, and are thus intrinsically memory-bound. However transpositions and permutations are also required for many other applications such as to compute fast Fourier transforms [Nukada et al. 2008] (that will be required for the spectral method) and many cache-aware and cache-oblivious algorithms have been developed for many different architectures [Ruetsch et al. 2009][Jodra et al. 2015][Lyakh 2015][Springer et al. 2016].

2.5 Directional splitting

In order to obtain only one-dimensional stencils and one dimensional remeshing kernels, the idea is to split directionally all partial differential equations by breaking them down into n directional terms:

$$\begin{aligned}\frac{\partial \boldsymbol{f}}{\partial t} &= \boldsymbol{L}\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}}{\partial x_1}, \cdots, \frac{\partial \boldsymbol{f}}{\partial x_n}, \frac{\partial \boldsymbol{f}^2}{\partial x_1^2}, \cdots, \frac{\partial \boldsymbol{f}^2}{\partial x_n^2}\right) \\ &= \boldsymbol{L}_1\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}}{\partial x_1}, \frac{\partial \boldsymbol{f}^2}{\partial x_1^2}\right) + \cdots + \boldsymbol{L}_n\left(\boldsymbol{x}, t, \boldsymbol{f}, \frac{\partial \boldsymbol{f}}{\partial x_n}, \frac{\partial \boldsymbol{f}^2}{\partial x_n^2}\right)\end{aligned} \tag{2.44}$$

Once $F_1, \cdots, F_n$ are known for diffusion, stretching and external forces, we can apply a first or second order Strang splitting to solve the respective operators (2.1.3).



This section is dedicated to express the directional splitting of all those operators. The analysis is done in the three-dimensional case where the velocity and the vorticity have three scalar components each but all the results can easily be extended to any dimension when applicable.

2.5.1 Advection

The three-dimensional transport equation (2.18) can be split into three distinct differential operators, each taking into account the effect of advection in a given direction:

$$\frac{\partial \theta}{\partial t} = -\underbrace{(\boldsymbol{u}\cdot\nabla)\theta}_{L(\boldsymbol{u},\partial_{\boldsymbol{x}}\theta)} = -\Bigg(\underbrace{u_x\frac{\partial\theta}{\partial x}}_{L_x(u_x,\partial_x\theta)} + \underbrace{u_y\frac{\partial\theta}{\partial y}}_{L_y(u_y,\partial_y\theta)} + \underbrace{u_z\frac{\partial\theta}{\partial z}}_{L_z(u_z,\partial_z\theta)}\Bigg) \tag{2.45}$$

As the advection is treated with a forward remeshed particle method the method is decomposed into an advection and a remeshing step. In the current direction i, the directionally split method just consists into a one-dimensional advection with velocity component u_i followed by a one-dimensional remeshing procedure as described in [Etancelin 2014]. This is illustrated on the following figure:

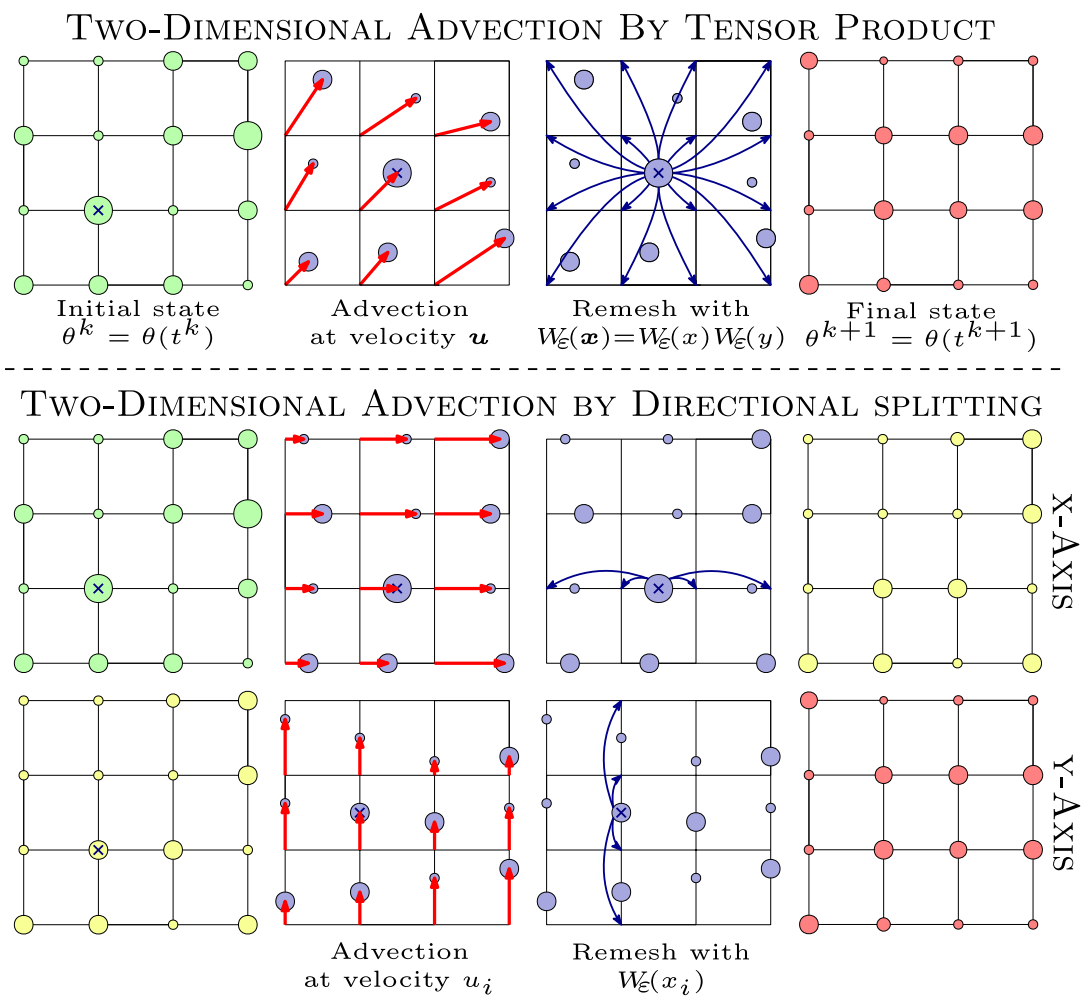


Figure 2.12 – Directional splitting of the advection-remeshing procedure

2.5.2 Diffusion

This operator appears in equation (2.1c) as a momentum diffusion due to shear stress and in equation (2.1d) as a scalar diffusion term due to molecular diffusivity:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,\boldsymbol{\omega} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} + \nu\Delta\boldsymbol{\omega} + \nabla \times \boldsymbol{f_{ext}}$$
$$\frac{\partial \theta}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,\theta = \kappa\Delta\theta$$

To illustrate the directional operator splitting (2.17), let us define $\boldsymbol{f} = [\omega_x, \omega_y, \omega_z, \theta]^T$ and $D = \mathrm{diag}(\nu, \nu, \nu, \kappa)$. Merging equations (2.3) and (2.5) leads to $\partial_t \boldsymbol{f} = \boldsymbol{L}\,(\partial_{\boldsymbol{xx}} \boldsymbol{f}) = D\Delta\boldsymbol{f}$ and we want to express $\boldsymbol{L}$ as a sum of directional operators $\boldsymbol{L}_x$, $\boldsymbol{L}_y$ and $\boldsymbol{L}_z$ where $\boldsymbol{L}_i$ contains only spatial derivatives with respect to axe i. The expression of $\boldsymbol{L_i}$ for each axis can be obtained by expanding the equation to four independent scalar partial differential equations:

$$\frac{\partial \boldsymbol{f}}{\partial t} = D\Delta\boldsymbol{f} \quad \Leftrightarrow \quad \begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \\ \frac{\partial \theta}{\partial t} \end{bmatrix} = \begin{bmatrix} \nu\Delta\omega_x \\ \nu\Delta\omega_y \\ \nu\Delta\omega_z \\ \kappa\Delta\theta \end{bmatrix} = \begin{bmatrix} \nu\left(\frac{\partial \omega_x^2}{\partial x^2} + \frac{\partial \omega_x^2}{\partial y^2} + \frac{\partial \omega_x^2}{\partial z^2}\right) \\ \nu\left(\frac{\partial \omega_y^2}{\partial x^2} + \frac{\partial \omega_y^2}{\partial y^2} + \frac{\partial \omega_y^2}{\partial z^2}\right) \\ \nu\left(\frac{\partial \omega_z^2}{\partial x^2} + \frac{\partial \omega_z^2}{\partial y^2} + \frac{\partial \omega_z^2}{\partial z^2}\right) \\ \kappa\left(\frac{\partial \theta^2}{\partial x^2} + \frac{\partial \theta^2}{\partial y^2} + \frac{\partial \theta^2}{\partial z^2}\right) \end{bmatrix}$$

We are looking to express $\boldsymbol{L}\left(\frac{\partial \boldsymbol{f}^2}{\partial x^2}, \frac{\partial \boldsymbol{f}^2}{\partial y^2}, \frac{\partial \boldsymbol{f}^2}{\partial z^2}\right)$ as $\boldsymbol{L_x}\left(\frac{\partial \boldsymbol{f}^2}{\partial x^2}\right) + \boldsymbol{L_y}\left(\frac{\partial \boldsymbol{f}^2}{\partial y^2}\right) + \boldsymbol{L_z}\left(\frac{\partial \boldsymbol{f}^2}{\partial z^2}\right)$.

As the right hand side of the equations does not contain any mixed derivative we can split it into three parts, each of them containing the partial derivatives specific to each axis. Here the red, green and blue terms represent the splittings for axe x, y and z.

1. x-axis splitting:

$$\frac{\partial \boldsymbol{f}}{\partial t} = \boldsymbol{L_x}\left(\frac{\partial \boldsymbol{f}^2}{\partial x^2}\right) = D\frac{\partial \boldsymbol{f}^2}{\partial x^2} \tag{2.47}$$

2. y-axis splitting:

$$\frac{\partial \boldsymbol{f}}{\partial t} = \boldsymbol{L_y}\left(\frac{\partial \boldsymbol{f}^2}{\partial y^2}\right) = D\frac{\partial \boldsymbol{f}^2}{\partial y^2} \tag{2.48}$$

3. z-axis splitting:

$$\frac{\partial \boldsymbol{f}}{\partial t} = \boldsymbol{L_z}\left(\frac{\partial \boldsymbol{f}^2}{\partial z^2}\right) = D\frac{\partial \boldsymbol{f}^2}{\partial z^2} \tag{2.49}$$

The same procedure can be repeated for any dimension n and for any operator that does not contain mixed spatial derivatives.

2.5.3 Stretching

Vortex stretching is the lengthening of vortices in three-dimensional fluid flow, associated with a corresponding increase of the component of vorticity in the stretching direction, due to the conservation of angular momentum. Here we want to evaluate explicitly the contribution due to the stretching term appearing in the conservation of momentum equation (2.1c).

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{\omega} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} + \nu \Delta \boldsymbol{\omega} + \nabla \times \boldsymbol{f_{ext}}$$

In the following, we will rather use the equivalent notation $(\nabla \boldsymbol{u})\boldsymbol{\omega} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u}$ where $\nabla \boldsymbol{u}$ is the (spatial) velocity gradient of $\boldsymbol{u}$ in dimension n:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = (\nabla \boldsymbol{u})\boldsymbol{\omega} \tag{2.50}$$

Equation (2.50) can be rewritten equivalently in either its conservative form or non-conservative form:

1. **Conservative formulation:** The stretching term can be rewritten in a conservative fashion by using the incompressibility condition (2.1a) and the definition of the vorticity (2.1b):

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{\omega}) \tag{2.51}$$

   This expression can be obtained by expanding the right hand side as the following $\nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{\omega}) = \boldsymbol{u}\, (\nabla \cdot \boldsymbol{\omega}) + (\nabla \boldsymbol{u})\boldsymbol{\omega}$ and by taking into account the fact that the vorticity is also divergence free: $\nabla \cdot \boldsymbol{\omega} = \nabla \cdot (\nabla \times \boldsymbol{u}) = \nabla \times (\nabla \cdot \boldsymbol{u}) = 0$. From a numerical point of view, the discretized version of the solver will not lead to the same property. Put in other words, this means that numerically we may have $\nabla \cdot \boldsymbol{\omega} \neq 0$ during the different stages of the solver.

2. **Non-conservative formulations:** We can split the Jacobian matrix of the velocity field into its symmetric and antisymmetric parts:

$$\nabla \boldsymbol{u} = \frac{\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T}{2} + \frac{\nabla \boldsymbol{u} - \nabla \boldsymbol{u}^T}{2} = \frac{1}{2}\left(\dot{\gamma} + \dot{\Omega}\right) \tag{2.52}$$

   and then express the coefficients of $\dot{\Omega}$ using vorticity components $\boldsymbol{\omega} = \nabla \times \boldsymbol{u}$:

$$\dot{\Omega} = \frac{\nabla \boldsymbol{u} - \nabla \boldsymbol{u}^T}{2} = \frac{1}{2}\begin{bmatrix} 0 & \frac{\partial u_x}{\partial y} - \frac{\partial u_y}{\partial x} & \frac{\partial u_x}{\partial z} - \frac{\partial u_z}{\partial x} \\ \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} & 0 & \frac{\partial u_y}{\partial z} - \frac{\partial u_z}{\partial y} \\ \frac{\partial u_z}{\partial x} - \frac{\partial u_x}{\partial z} & \frac{\partial u_z}{\partial y} - \frac{\partial u_y}{\partial z} & 0 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0 & -\omega_z & +\omega_y \\ +\omega_z & 0 & -\omega_x \\ -\omega_y & +\omega_x & 0 \end{bmatrix} \tag{2.53}$$

From the above relation we can deduce that $\dot{\Omega}\boldsymbol{\omega} = \mathbf{0}$ from which we obtain directly that $(\nabla \boldsymbol{u})\boldsymbol{\omega} = (\nabla \boldsymbol{u}^T)\boldsymbol{\omega}$ which gives a new non-conservative stretching formulation:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = (\nabla \boldsymbol{u})^T \boldsymbol{\omega} \tag{2.54}$$

More generally we can choose a linear combination of those two terms:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \left[\alpha \nabla \boldsymbol{u} + (1-\alpha)(\nabla \boldsymbol{u})^T\right] \boldsymbol{\omega} \quad \forall \alpha \in \mathbb{R} \tag{2.55}$$

With $\alpha = \frac{1}{2}$, we obtain the following expression for the stretching term:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \frac{\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T}{2} \boldsymbol{\omega} = \dot{\gamma} \boldsymbol{\omega} \tag{2.56}$$

Equations (2.51) and (2.55) with $\alpha \in \{0, 0.5, 1\}$ give the classical stretching formulations used in the literature [Mimeau 2015]. Those formulations are directionally split as the following:

1. **Conservative formulation:**

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{\omega}) \quad \Leftrightarrow \quad \begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} \frac{\partial u_x \omega_x}{\partial x} + \frac{\partial u_x \omega_y}{\partial y} + \frac{\partial u_x \omega_z}{\partial z} \\ \frac{\partial u_y \omega_x}{\partial x} + \frac{\partial u_y \omega_y}{\partial y} + \frac{\partial u_y \omega_z}{\partial z} \\ \frac{\partial u_z \omega_x}{\partial x} + \frac{\partial u_z \omega_y}{\partial y} + \frac{\partial u_z \omega_z}{\partial z} \end{bmatrix}$$

The splitting of the conservative formulation is straightforward once the scalar equations have been expressed:

(a) x-axis splitting:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \frac{\partial}{\partial x} [\omega_x \boldsymbol{u}] \tag{2.57}$$

(b) y-axis splitting:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \frac{\partial}{\partial y} [\omega_y \boldsymbol{u}] \tag{2.58}$$

(c) z-axis splitting:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \frac{\partial}{\partial z} [\omega_z \boldsymbol{u}] \tag{2.59}$$

2. **Non-conservative formulations:**

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \left[\alpha \nabla \boldsymbol{u} + (1-\alpha)\nabla \boldsymbol{u}^T\right]\boldsymbol{\omega}$$

$$\Leftrightarrow \begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \alpha \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} & \frac{\partial u_x}{\partial z} \\ \frac{\partial u_y}{\partial x} & \frac{\partial u_y}{\partial y} & \frac{\partial u_y}{\partial z} \\ \frac{\partial u_z}{\partial x} & \frac{\partial u_z}{\partial y} & \frac{\partial u_z}{\partial z} \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} + (1-\alpha) \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_y}{\partial x} & \frac{\partial u_z}{\partial x} \\ \frac{\partial u_x}{\partial y} & \frac{\partial u_y}{\partial y} & \frac{\partial u_z}{\partial y} \\ \frac{\partial u_x}{\partial z} & \frac{\partial u_y}{\partial z} & \frac{\partial u_z}{\partial z} \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

The directional splitting of the non-conservative formulations is less elegant but is easily obtained by extracting the directional space derivatives:

(a) x-axis splitting:

$$\begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} \frac{\partial u_x}{\partial x}\omega_x + (1-\alpha)\left[\frac{\partial u_y}{\partial x}\omega_y + \frac{\partial u_z}{\partial x}\omega_z\right] \\ \alpha \frac{\partial u_y}{\partial x}\omega_x \\ \alpha \frac{\partial u_z}{\partial x}\omega_x \end{bmatrix} \tag{2.60}$$

(b) y-axis splitting:

$$\begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} \alpha \frac{\partial u_x}{\partial y}\omega_y \\ \frac{\partial u_y}{\partial y}\omega_y + (1-\alpha)\left[\frac{\partial u_x}{\partial y}\omega_x + \frac{\partial u_z}{\partial y}\omega_z\right] \\ \alpha \frac{\partial u_z}{\partial y}\omega_y \end{bmatrix} \tag{2.61}$$

(c) z-axis splitting:

$$\begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} \alpha \frac{\partial u_x}{\partial z}\omega_z \\ \alpha \frac{\partial u_y}{\partial z}\omega_z \\ \frac{\partial u_z}{\partial z}\omega_z + (1-\alpha)\left[\frac{\partial u_x}{\partial z}\omega_x + \frac{\partial u_y}{\partial z}\omega_y\right] \end{bmatrix} \tag{2.62}$$

2.5.4 External forces and immersed boundaries

External forces (2.7) are handled by an operator of the form:

$$\frac{\partial \omega}{\partial t} = \nabla \times \boldsymbol{A} \quad \Leftrightarrow \quad \begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} \frac{\partial A_z}{\partial y} - \frac{\partial A_y}{\partial z} \\ \frac{\partial A_x}{\partial z} - \frac{\partial A_z}{\partial x} \\ \frac{\partial A_y}{\partial x} - \frac{\partial A_x}{\partial y} \end{bmatrix}$$

They can be split as the following:

1. x-axis splitting:

$$\begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{\partial A_z}{\partial x} \\ \frac{\partial A_y}{\partial x} \end{bmatrix} \tag{2.63}$$

2. y-axis splitting:

$$\begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} \frac{\partial A_z}{\partial y} \\ 0 \\ -\frac{\partial A_x}{\partial y} \end{bmatrix} \tag{2.64}$$

3. z-axis splitting:

$$\begin{bmatrix} \frac{\partial \omega_x}{\partial t} \\ \frac{\partial \omega_y}{\partial t} \\ \frac{\partial \omega_z}{\partial t} \end{bmatrix} = \begin{bmatrix} -\frac{\partial A_y}{\partial z} \\ \frac{\partial A_x}{\partial y} \\ 0 \end{bmatrix} \tag{2.65}$$

2.6 Fourier spectral methods

Spectral methods are a class of techniques to numerically solve certain differential equations in which the solution of the differential equation is decomposed in a function space whose basis are non-zero over the whole domain [Orszag 1969]. Unlike finite differences for which only local information are used to find a solution, spectral methods use basis functions that spans over the whole domain. Because spectral methods constitute global approaches, they offer excellent convergence properties given that the solution is smooth [Gottlieb et al. 1977].

In this section we will first focus on spectral methods arising the decomposition of periodic functions in Fourier series to numerically solve diffusion and Poisson problems on Cartesian grids. Spectral diffusion is unconditionally stable. It is an interesting alternative to explicit finite-differences based solvers when the diffusivity is high. Poisson solvers allow to correct the vorticity and compute the velocity from the corrected vorticity. This results into fast numerical solvers by using the fast Fourier transform [Cooley et al. 1965]. The method is then extended to handle Dirichlet and Neumann homogeneous boundary conditions by changing the basis of functions while remaining compatible with the use of the fast Fourier transform (FFT). The difficulties concerning domain decomposition within this spectral framework are discussed in the next chapter.

2.6.1 Discrete Fourier transform

In spectral methods for differential equations, considering one dimension here for simplicity, one has a periodic function $y : [0, L[\to \mathbb{C}$ with period L that expands as a Fourier series:

$$y(x) = \sum_{p=-\infty}^{\infty} \widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{px}{L}} \tag{2.66a}$$

$$\widehat{y_p} = \frac{1}{L}\int_0^L y(x)\ \mathrm{e}^{-2\boldsymbol{i}\pi\frac{px}{L}}\,\mathrm{d}x \tag{2.66b}$$

where $\boldsymbol{i}$ denotes the imaginary unit.

One then wishes to apply a differential operator like $\dfrac{\mathrm{d}^q}{\mathrm{d}x^q}$ for some $q \in \mathbb{N}^*$. Differentiation is performed term-by-term in the Fourier domain:

$$\frac{\mathrm{d}y^q}{\mathrm{d}x^q}(x) = \sum_{p=-\infty}^{\infty} \left(2\boldsymbol{i}\pi\frac{px}{L}\right)^q \widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{px}{L}} \tag{2.67}$$

which is just a pointwise multiplication of each $\widehat{y_p}$ by a complex factor depending on p.

To implement this on a computer, one approximates the Fourier series by a discrete Fourier transform (DFT) and we replace the function $y(x)$ by N discrete samples $\boldsymbol{y} = (y_j)_{j\in[\![0,N[\![}$

such that $y_j = y(jdx)$ with $dx = \frac{L}{N}$. Within this discrete framework $\widehat{\boldsymbol{y}} = (\widehat{y}_p)_{p\in[\![0,N[\![}$ is approximated by the discrete Fourier transform (DFT):

$$\widehat{y_p} = \frac{1}{N}\sum_{j=0}^{N-1} y_j \ \mathrm{e}^{-2\boldsymbol{i}\pi\frac{jp}{N}} \tag{2.68}$$

and the discrete samples y_j are recovered from the discrete Fourier coefficients $\widehat{y_p}$ by the associated inverse transform (IDFT):

$$y_j = \sum_{p=0}^{N-1} \widehat{y_p} \ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{jp}{N}} \tag{2.69}$$

An alternative and more compact notation is $\widehat{\boldsymbol{y}} = \mathcal{F}(\boldsymbol{y})$ and $\boldsymbol{y} = \mathcal{F}^{-1}(\widehat{\boldsymbol{y}})$. With these notations, it follows that $\mathcal{F}^{-1}(\mathcal{F}(\boldsymbol{y})) = \boldsymbol{y}$.

Fast Fourier Transform

The nice thing about the discrete Fourier transform expressions is that $\widehat{\boldsymbol{y}}$ can be computed from $\boldsymbol{y}$ in only $\mathcal{O}(NlogN)$ operations by a fast Fourier transform (FFT) algorithm instead of $\mathcal{O}(N^2)$ operations obtained for a naïve implementation based on its definition. By far the most commonly used FFT is the Cooley-Tukey algorithm that is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1N_2$ into many smaller DFTs of sizes N_1 and N_2 [Cooley et al. 1965]. The best known use of this algorithm is to divide the transform into two pieces of size $N/2$ at each step (radix-2 decimation), and is therefore limited to power-of-two sizes $N = 2^p$, but any factorization can be used in general. This algorithm has led to multiple variants called split-radix FFT algorithm [Duhamel et al. 1984]. The Cooley-Tukey has the disadvantage that it also requires extra multiplications by roots of unity called twiddle factors, in addition to the smaller transforms. The prime-factor algorithm (PFA) or Good-Thomas algorithm is a variant of Cooley-Tukey when $N1$ and $N2$ are coprimes that do not require those extra multiplications [Good 1958]. Transforms of arbitrary sizes, including prime sizes can be computed by the use of Rader [Rader 1968] or Bluestein [Bluestein 1970] algorithms that work by rewriting the DFT as a convolution. In the presence of round-off error, many FFT algorithms are much more accurate than evaluating the DFT definition directly. Considering that ε is the machine floating-point relative precision, the upper bound on the relative error for the Cooley-Tukey algorithm is $\mathcal{O}(\varepsilon \log N)$ versus $\mathcal{O}\left(\varepsilon N\sqrt{N}\right)$ for the naïve DFT formula [Gentleman et al. 1966]. The root mean square (RMS) errors are much better than these upper bounds, being only $\mathcal{O}(\varepsilon \log N)$ for Cooley-Tukey and $\mathcal{O}\left(\varepsilon\sqrt{N}\right)$ for the naïve DFT [Schatzman 1996]. Those algorithms makes working with Fourier series practical: we can quickly and accurately transform back and forth between space domain (where multiplying by functions is easy) and Fourier domain (where operations like derivatives are easy) by using single or double precision floating point numbers.

Trigonometric interpolation and derivatives

FFT implementations like `FFTW`, `FFTPACK`, `MKL-FFT` or `clFFT` compute DFTs and IDFTs in forms similar to equations (2.68) and (2.69), with the coefficients arranged in order (from $p = 0$ to $N-1$ for the forward transform and from $j = 0$ to $N-1$ for the backward transform). This ordering turns out to make the correct implementation of FFT-based differentiation more obscure. Let $\boldsymbol{y}'$ denote the sampling of the q-th derivative of $y(x)$ and $\widehat{\boldsymbol{y}'}$ its corresponding discrete Fourier transform. From equation (2.67) one should expect that $\widehat{\boldsymbol{y}'}$ can be expressed from $\widehat{\boldsymbol{y}}$ by the following relation: $\widehat{y'_p} = \left(2\boldsymbol{i}\pi\frac{px}{L}\right)^q \widehat{y_p}$. However in practice, this is not the case.

In order to compute derivatives, we need to use the IDFT expression to define a continuous interpolation between the samples y_j and then differentiate this trigonometric interpolation. At first glance, interpolating seems very straightforward: one simply evaluates the IDFT expression (2.69) at non-integer $x \in [0, L[$:

$$y_N(x) = \sum_{p=0}^{N-1} \widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{px}{L}} \tag{2.70}$$

This indeed defines an interpolation but this is not the only one. The reason there is more than one interpolation is due to aliasing: Any term of the form $\widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{pj}{N}}$ in the IDFT can be replaced by $\widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{j(p+mN)}{N}}$ for any integer m and still give the same samples $\boldsymbol{y}$ because $\widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{pj}{N}} = \widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{pj}{N}} \underbrace{\mathrm{e}^{+2\boldsymbol{i}\pi jm}}_{=1\ \forall m\in\mathbb{Z}} = \widehat{y_p}\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{j(p+mN)}{N}}$.

Essentially, adding mN to p means that the interpolated function $y(x)$ oscillates m extra times in between the sample points y_j and y_{j+1}. This has no effect on $\boldsymbol{y}$ but has a huge effect on its derivatives such as $\boldsymbol{y}'$. The unique minimal-oscillation trigonometric interpolation of order N is obtained by the following formula [Johnson 2011]:

$$y_N(x) = \begin{cases} \widehat{y}_0 + \sum_{p=1}^{(N/2-1)} \left(\widehat{y}_p\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{px}{L}} + \widehat{y}_{N-p}\ \mathrm{e}^{-2\boldsymbol{i}\pi\frac{px}{L}}\right) + \widehat{y}_{N/2}\ cos\left(\pi\frac{Nx}{L}\right) & \text{if } N \text{ is even} \quad (2.71a) \\ \widehat{y}_0 + \sum_{p=1}^{(N-1)/2} \left(\widehat{y}_p\ \mathrm{e}^{+2\boldsymbol{i}\pi\frac{px}{L}} + \widehat{y}_{N-p}\ \mathrm{e}^{-2\boldsymbol{i}\pi\frac{px}{L}}\right) & \text{if } N \text{ is odd} \quad (2.71b) \end{cases}$$

In this formula, the $N/2$ (Nyquist) term is absent for odd N.

The treatment of the maximum frequency (Nyquist component) is especially tricky when computing derivatives on even sized grid that are typically fast to compute FFTs: an odd mode is left without a conjugate partner as seen in the interpolant (2.71a).

We define the N wavenumbers $\boldsymbol{k} = (k_p)_{p\in[\![0,N[\![}$ with $k_p = \begin{cases} \frac{2\boldsymbol{i}\pi}{L}p & \text{if } p \leq \left\lfloor\frac{N-1}{2}\right\rfloor \\ \frac{2\boldsymbol{i}\pi}{L}(p-N) & \text{if } p > \left\lfloor\frac{N-1}{2}\right\rfloor \end{cases}$.

Computing $\frac{\partial y^q}{\partial x^q}(x)$ from spectral interpolation (2.71b) and evaluating it $x_j \in \boldsymbol{x}$ for an odd

N gives directly $y'_p = (k_p)^q y_p$. For even N and even derivatives, the result stays the same because the Nyquist component remains a cosine that evaluates to $+1$ or -1 at all discrete samples x_j. For even N and odd derivatives however the derivation of (2.71a) contains a Nyquist component with a sine that evaluates to 0 at all x_j such that $y_{N/2}$ vanishes. In this case we have to modify the formula to $y'_p = (1-\delta_{p,N/2})(k_p)^q y_p$ to impose $y'_{N/2} = 0$. Hence performing a second derivative is not equivalent to performing the spectral first derivative twice unless N is odd (the discretized spectral differential operators do not inherit all of the properties of the continuum operators).

To express any derivative, we define vector $\boldsymbol{k}' = (k'_p)_{p\in[\![0,N[\![}$ with $k'_p = (1-\delta_{p,N/2})k_p$.

$$g_N(x_j) = \left(\frac{\mathrm{d}^{2q} f}{\mathrm{d}x^{2q}}\right)_N (x_j) = \sum_{p=0}^{N-1} \widehat{g_p}\ \mathrm{e}^{+2i\pi\frac{jp}{N}} \qquad \text{with} \quad \widehat{g_p} = \underbrace{(k_p)^{2q}}_{\in\mathbb{R}} \widehat{f_p} \tag{2.72a}$$

$$h_N(x_j) = \left(\frac{\mathrm{d}^{2q+1} f}{\mathrm{d}x^{2q+1}}\right)_N (x_j) = \sum_{p=0}^{N-1} \widehat{h_p}\ \mathrm{e}^{+2i\pi\frac{jp}{N}} \qquad \text{with} \quad \widehat{h_p} = \underbrace{k'_p (k_p)^{2q}}_{\in\mathbb{C}} \widehat{f_p} \tag{2.72b}$$

Smoothness and spectral accuracy

In order to apply the discrete Fourier transform to interpolate a signal, compute derivatives or solve partial differential equations, the spectral approximation $f_N(x)$ from equation (2.71) should converge rapidly to $f(x)$ as $N \to \infty$. Let $f \in L^2(\mathbb{R})$ be a L-periodic function. The convergence rate of the spectral approximation f_N is dictated by the regularity of the periodized function f:

1. If $f(x)$ is analytic, the Fourier series converges exponentially fast:

$$f \in L^2(\mathbb{R}) \cap C^\omega(\mathbb{R}) \ \Rightarrow\ \exists \alpha > 0\ \ \|f_N(x) - f(x)\|_\infty = \underset{N\to\infty}{\mathcal{O}}\left(\mathrm{e}^{-\alpha N}\right) \tag{2.73}$$

2. If $f(x)$ has p square-integrable continuous derivatives and a $(p+1)$-th derivative of bounded variation, the Fourier series converges at order p:

$$\begin{cases} f \in L^2(\mathbb{R}) \cap C^p(\mathbb{R}) \\ f^{(q)} \in L^2(\mathbb{R}) \quad \forall q \in [\![1,p]\!] \\ f^{(p+1)} \in BV(\mathbb{R}) \end{cases} \Rightarrow \|f_N(x) - f(x)\|_\infty = \underset{N\to\infty}{\mathcal{O}}\left(\frac{1}{N^{p+1}}\right) \tag{2.74}$$

3. If f is piecewise continuous (discontinuous at some points or simply non-periodic), with a first derivative of bounded variation, f_N is not a good approximation anymore:

$$\begin{cases} f \in L^2(\mathbb{R}) \cap C^0_I(\mathbb{R}) \setminus C^0(\mathbb{R}) \\ f^{(1)} \in BV(\mathbb{R}) \end{cases} \Rightarrow \begin{cases} \|f_N(x) - f(x)\|_\infty = \underset{N\to\infty}{\mathcal{O}}(1) \\ \|f_N(x) - f(x)\|_2 = \underset{N\to\infty}{\mathcal{O}}\left(\frac{1}{\sqrt{N}}\right) \end{cases} \tag{2.75}$$

The three different cases are illustrated on figure 2.13 and 2.14. The inability to recover point values of a non-periodic (or discontinuous), but otherwise perfectly smooth function from its Fourier coefficients is known as the Gibbs phenomenon. Away from the discontinuity the convergence is rather slow ($|f(x) - f_n(x)| = \mathcal{O}(1/N)$) and close to the discontinuous locations x_d there is an overshoot that does not diminish with increasing N ($|f(x_d) - f_N(x_d)| = \mathcal{O}(1)$).

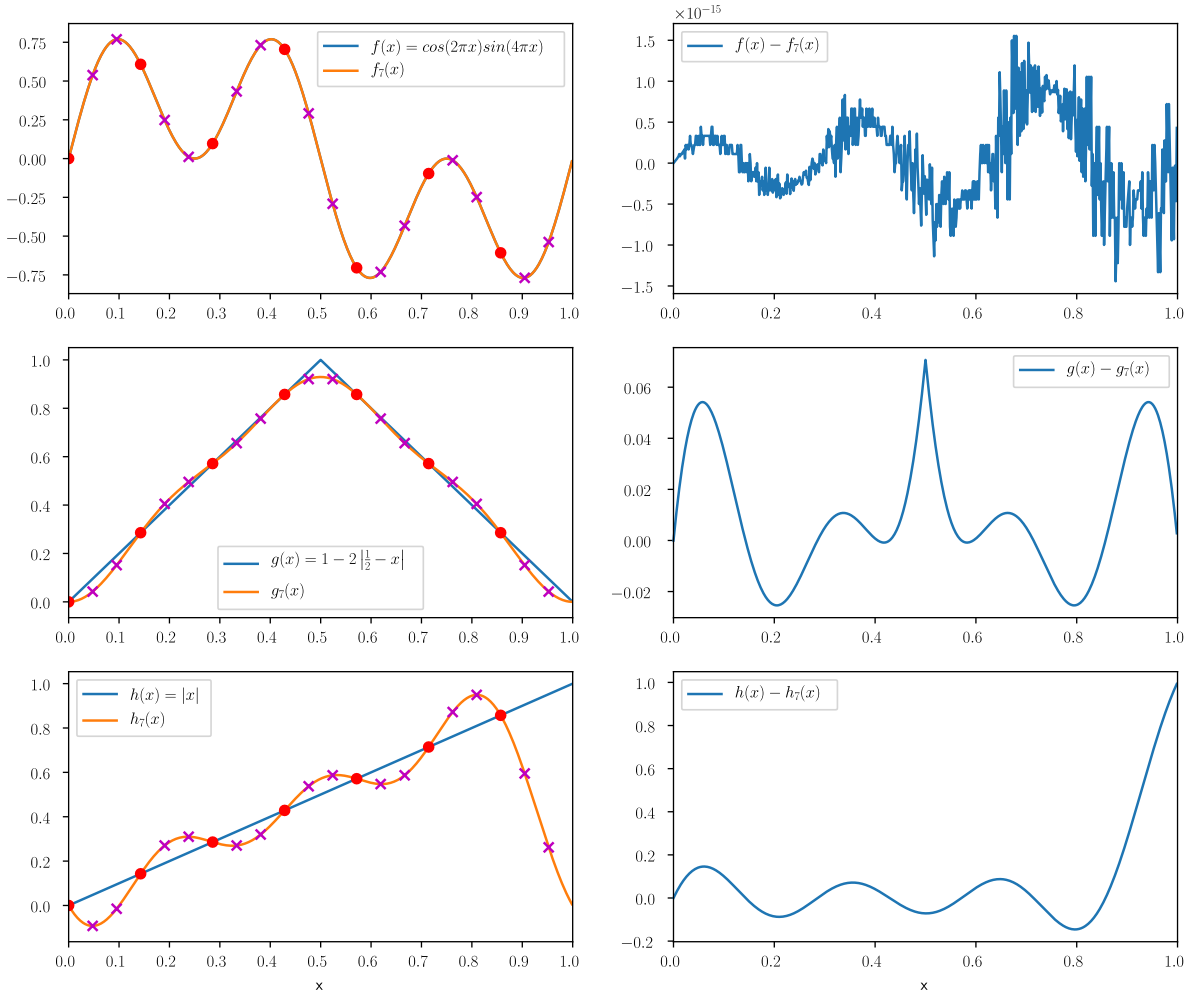


Figure 2.13 – Spectral interpolation of 1-periodic functions of varying regularity. The three functions considered here are all one-periodic square-integrable functions with at least a first derivative with bounded variations. For all $x \in [0, 1[$ we define f, g and h as $f(x) = cos(2\pi x)sin(4\pi x) \in C^\omega(\mathbb{R})$, $g(x) = 1 - 2|0.5 - x| \in C^0(\mathbb{R})$ and $h(x) = |x| \in C^0_I(\mathbb{R}) \setminus C^0(\mathbb{R})$. Those functions have decreasing regularity and match the three convergence cases (2.73), (2.74) and (2.75). Each leftmost plot represent the function $f(x)$ to be interpolated (blue) and its spectral DFT interpolant $f_7(x)$ obtained by evaluating (2.71) using $\mathcal{O}(N)$ operations per point (orange) with $N = 7$ samples. The samples $x_j = j/8 \in [0, 1[$ used to compute the DFT are shown as red points and discrete interpolation of size $3N$ is represented by magenta crosses. This interpolation is obtained by an IDFT of the N spectral coefficients $\widehat{f}$ padded with $2N$ zeros, computed in $\mathcal{O}(\log 3N)$ operations per point. Rightmost plots represent the pointwise errors between each function and their interpolant $f(x) - f_7(x)$.

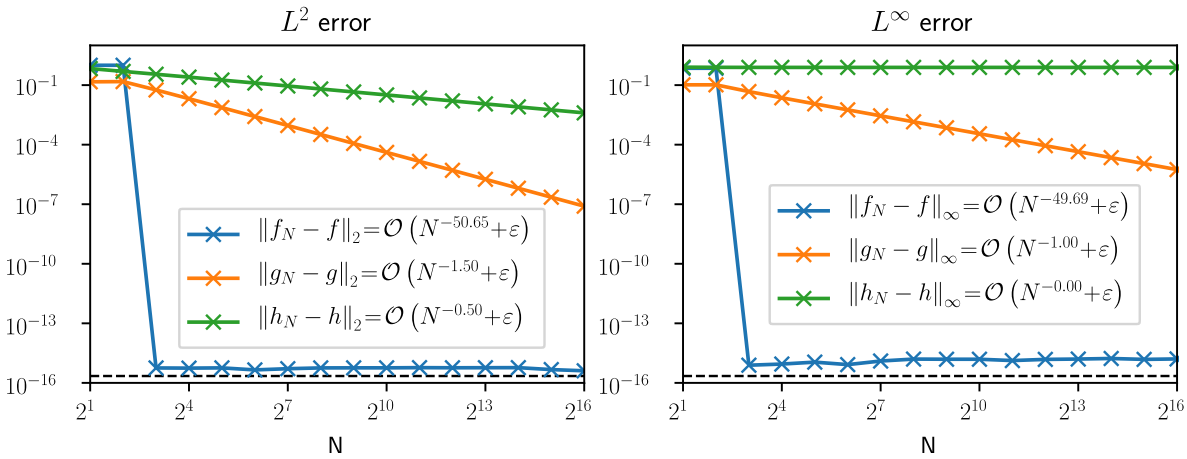


Figure 2.14 – Spectral convergence of 1-periodic functions of varying regularity. Here the black dotted line represents the machine error $y = \varepsilon \simeq 2.22 \times 10^{-16}$, see figure 2.13.

DFT of a real signal

When the DFT takes a purely real signal $y : [0, L[\to \mathbb{R}$ as input, it is possible to use a smaller index set due to the Hermitian symmetry of the output: $\widehat{y_p} = \left(\widehat{y_{N-p}}\right)^*$ where $\boldsymbol{c}^*$ denotes the complex conjugate of $\boldsymbol{c}$. It is possible to take advantage of these circumstances in order to achieve roughly a factor of two improvement in both speed and memory usage. This real-to-complex forward discrete Fourier transform that takes N reals as input and outputs $\left\lfloor \frac{N}{2} \right\rfloor + 1$ complex numbers will be denoted $\mathcal{H}$ as opposed to $\mathcal{F}$ that performs a complex-to-complex transform The corresponding forward $\mathcal{H}$ and backward $\mathcal{H}^{-1}$ transforms are defined as:

$$\widehat{y_p} = \frac{1}{N} \sum_{j=0}^{N-1} y_j \, \mathrm{e}^{-2\boldsymbol{i}\pi \frac{jp}{N}} \quad \forall p \in [\![0, \lfloor N/2 \rfloor + 1[\![\tag{2.76a}$$

$$y_j = \widehat{y}_0 + \sum_{p=1}^{\lfloor \frac{N-1}{2} \rfloor} \left(\widehat{y}_p \, \mathrm{e}^{+2\boldsymbol{i}\pi \frac{jp}{N}} + \widehat{y}_p^* \, \mathrm{e}^{-2\boldsymbol{i}\pi \frac{jp}{N}} \right) + \mathbb{1}_{2\mathbb{Z}}(N) \, \widehat{y}_{N/2} \underbrace{cos(j\pi)}_{=(-1)^j} \quad \forall j \in [\![0, N[\![\tag{2.76b}$$

where $\mathbb{1}_{2\mathbb{Z}}(N)$ is one when N is even and zero when N is odd.

Using the Hermitian symmetry of the output and equation (2.71) leads to:

$$\begin{aligned} y_N(x) = \widehat{y}_0 + 2 \sum_{p=1}^{\lfloor \frac{N-1}{2} \rfloor} \left[\mathrm{Re}\left(\widehat{y}_p\right) \cos\left(2\pi \frac{px}{L}\right) - \mathrm{Im}\left(\widehat{y}_p\right) \sin\left(2\pi \frac{px}{L}\right) \right] \\ + \mathbb{1}_{2\mathbb{Z}}(N) \, \widehat{y}_{N/2} \cos\left(\pi \frac{Nx}{L}\right) \end{aligned} \tag{2.77}$$

In this formula $\hat{y_0} \in \mathbb{R}$ represent the mean of the signal and when N is even, $\hat{y_{N/2}} \in \mathbb{R}$ is also a real coefficient associated to the highest frequency (Nyquist frequency). As a direct consequence, real-valued samples $y_j = y(x_j)$ will result in a purely real-valued interpolation $y(x)$ for all x. Equations (2.72) remain valid to compute even and odd derivatives with modified versions of $\boldsymbol{k}$ and $\boldsymbol{k}'$, mainly $k_p = \dfrac{2\boldsymbol{i}\pi}{L} p$ and $k'_p = (1 - \delta_{p,N/2}) k_p$ for $p \in [\![0, \lfloor N/2 \rfloor]\!]$.

Multi-dimensional discrete Fourier transform

The n-dimensional DFT is achieved by tensor product of one-dimensional DFTs. Let $\theta(\boldsymbol{x},t) : \Omega\times\mathcal{T}\to\mathbb{R}$ be a n-dimensional $\boldsymbol{L}$-periodic (in space) and time-dependent scalar field with $\boldsymbol{L} = (L_1,\cdots,L_n)$ and $\Omega = [0,L_1[\times\cdots\times[0,L_n[$. The field is discretized on a Cartesian grid of $\boldsymbol{\mathcal{N}} = \boldsymbol{\mathcal{N}}^v = \boldsymbol{\mathcal{N}}^c$ cells and vertices such that $\theta^k = \left\{\theta^k_{\boldsymbol{j}} = \theta(\boldsymbol{j}\odot\boldsymbol{dx},t^k) \mid \boldsymbol{j}\in[\![0,\mathcal{N}_1[\![\times\cdots\times[\![0,\mathcal{N}_n[\![\right\}$ with $\boldsymbol{dx} = \boldsymbol{L}\oslash\boldsymbol{\mathcal{N}}$ and $N = \prod_{i=1}^n \mathcal{N}_i$.

The forward n-dimensional discrete Fourier transform applied to θ^k is defined as:

$$\begin{aligned}\widehat{\theta^k_{\boldsymbol{p}}} &= \frac{1}{N}\sum_{j_1=0}^{\mathcal{N}_1-1}\left(\mathrm{e}^{-2\boldsymbol{i}\pi\frac{p_1j_1}{\mathcal{N}_1}}\sum_{j_2=0}^{\mathcal{N}_2-1}\left(\mathrm{e}^{-2\boldsymbol{i}\pi\frac{p_2j_2}{\mathcal{N}_2}}\cdots\sum_{j_n=0}^{\mathcal{N}_n-1}\mathrm{e}^{-2\boldsymbol{i}\pi\frac{p_nj_n}{\mathcal{N}_n}}\theta^k_{\boldsymbol{j}}\right)\right)\\ &= \frac{1}{N}\sum_{j_1=0}^{\mathcal{N}_1-1}\cdots\sum_{j_n=0}^{\mathcal{N}_n-1}\theta^k_{\boldsymbol{j}}\,\mathrm{e}^{-2\boldsymbol{i}\pi\sum_{q=1}^n\frac{p_qj_q}{\mathcal{N}_q}}\qquad\forall\boldsymbol{p}\in[\![0,\mathcal{N}_1[\![\times\cdots\times[\![0,\mathcal{N}_n[\![\end{aligned}\tag{2.78}$$

which can be simplified by using the following notation: $\widehat{\theta^k} = \mathcal{F}\left(\theta^k\right) = \mathcal{F}_1\left(\mathcal{F}_2\left(\cdots\mathcal{F}_n\left(\theta^k\right)\right)\right)$ where $\mathcal{F}_i$ denotes the forward partial discrete Fourier transform along axis i. Similarly we define the backward n-dimensional DFT as $\theta^k = \mathcal{F}^{-1}\left(\widehat{\theta^k}\right) = \mathcal{F}_n^{-1}\left(\mathcal{F}_{n-1}^{-1}\left(\cdots\mathcal{F}_1^{-1}\left(\widehat{\theta^k}\right)\right)\right)$. Those two transforms can be computed in $\mathcal{O}\left(N\log N\right)$ operations by using successive calls to any compatible one-dimensional FFT algorithm to compute each transforms $\mathcal{F}_i$ or $\mathcal{F}_i^{-1}$.

The $\boldsymbol{q}$-th spatial derivative of θ is obtained as in the one-dimensional case and for all $\boldsymbol{j}\in[\![0,\mathcal{N}_1[\![\times\cdots\times[\![0,\mathcal{N}_n[\![$ we have:

$$\begin{aligned}\zeta_N(\boldsymbol{x_j}) &= \left(\frac{\partial^{\boldsymbol{q}}\theta}{\partial x^{\boldsymbol{q}}}\right)_N(\boldsymbol{x_j}) = \frac{\partial^{q_1+\cdots+q_n}\theta}{\partial x_1^{q_1}\cdots\partial x_n^{q_n}}(\boldsymbol{j}\odot\boldsymbol{dx})\\ &= \sum_{p_1=0}^{\mathcal{N}_1-1}\cdots\sum_{p_n=0}^{\mathcal{N}_n-1}\left[\prod_{i=1}^n\exp\left(2\boldsymbol{i}\pi\frac{p_ij_i}{\mathcal{N}_i}\right)\right]\widehat{\zeta}_{p_1,\cdots,p_n}\\ \text{with}\quad\widehat{\zeta}_{p_1,\cdots,p_n} &= \left(\prod_{i=1}^n\left[1-\mathbb{1}_{2\mathbb{Z}+1}(q_i)\,\delta_{p_i,\mathcal{N}_i/2}\right](k_{p_i})^{q_i}\right)\widehat{\theta}_{p_1,\cdots,p_n}\end{aligned}\tag{2.79}$$

If the partial derivative is not performed on every axis, it is possible to compute only a partial m-dimensional transform with $m<n$.

As we will only deal with real scalar fields as inputs, the n-dimensional real-to-complex forward discrete Fourier transform is defined as the following:

$$\begin{aligned}\mathcal{H}:\mathbb{R}^{\mathcal{N}_1\times\cdots\times\mathcal{N}_{n-1}\times\mathcal{N}_n} &\to \mathbb{C}^{\mathcal{N}_1\times\cdots\times\mathcal{N}_{n-1}\times\left\lfloor\frac{\mathcal{N}_n}{2}\right\rfloor+1}\\ \theta^k &\mapsto \widehat{\theta^k} = \mathcal{F}_1\left(\mathcal{F}_2\left(\cdots\mathcal{F}_{n-1}\left(\mathcal{H}_n\left(\theta^k\right)\right)\right)\right)\end{aligned}\tag{2.80}$$

Its corresponding complex-to-real backward transform is defined as $\mathcal{H}^{-1} = \mathcal{H}_n^{-1}\circ\mathcal{F}_{n-1}^{-1}\circ\cdots\circ\mathcal{F}_1^{-1}$. In this case we have $\mathcal{H}^{-1}\left(\mathcal{H}\left(\theta^k\right)\right) = \theta^k$ where where $\mathcal{H}_i$ and $\mathcal{H}_i^{-1}$ use the Hermitian symmetry of the DFT on the last axis (n is the first axis being forward transformed).

2.6.2 Spectral diffusion

Diffusion problems were already encountered in equation (2.3) and were already solved by using the finite differences approach in section 2.4.2. Let Ω be a n-dimensional periodic box of size $\boldsymbol{L}$ discretized on a grid of $\boldsymbol{\mathcal{N}}^c = \boldsymbol{\mathcal{N}}^v = \boldsymbol{\mathcal{N}}$ point with $N = \prod_{i=1}^{n} \mathcal{N}_i$. Let θ^k be the discretization of $\theta(x,t) : \Omega \times \mathcal{T} \to \mathbb{R}$ with $\theta^k_{\boldsymbol{j}} = \theta(\boldsymbol{j} \odot \boldsymbol{dx}, t^k) \;\; \forall \boldsymbol{j} \in \mathcal{J} = [\![0, \mathcal{N}_1[\![\times \cdots \times [\![0, \mathcal{N}_n[\![$ with $\boldsymbol{dx} = \boldsymbol{L} \oslash \boldsymbol{\mathcal{N}}$. Here we suppose that $\theta(t = t^k) \in L^2(\mathbb{R})$ and $\theta(t = t^{k+1}) \in L^2(\mathbb{R})$ are at least C^2 with first and second order derivatives that are square integrable and with third order derivatives of bounded variations.

We want to solve the following diffusion equation implicitely in time so that there will be no restriction on the timestep:

$$\frac{\partial \theta}{\partial t} = \kappa \, \Delta \theta \quad t \in [t^k, t^k + dt[\tag{2.81a}$$

$$\theta^k = \theta(t^k) \tag{2.81b}$$

where κ is a constant. Implicit discretization in time using an Euler scheme yields:

$$\theta(x, t^{k+1}) = \theta(x, t^k) + dt \, \kappa \, \Delta \theta(x, t^{k+1}) \quad \forall x \in \Omega \tag{2.82}$$

In order to solve this equation, we impose equality (2.82) for the variable discretized on the grid $\theta^k_{\boldsymbol{j}} = \theta(t^k, x_{\boldsymbol{j}})$:

$$\theta^{k+1}_{\boldsymbol{j}} = \theta^k_{\boldsymbol{j}} + dt \, \kappa \, \Delta \theta^{k+1}_{\boldsymbol{j}} \quad \forall \boldsymbol{j} \in \mathcal{J} \tag{2.83}$$

The derivatives can then be approximated at order p by a weighted sum of the discretized variable values:

$$\theta^{k+1}_{\boldsymbol{j}} = \theta^k_{\boldsymbol{j}} + dt \, \kappa \sum_{\boldsymbol{i} \in \mathcal{J}} l_{\boldsymbol{j},\boldsymbol{i}} \, \theta^{k+1}_{\boldsymbol{i}} + \mathcal{O}\left(N^{-p}\right) \quad \forall \boldsymbol{j} \in \mathcal{J} \tag{2.84}$$

$$\Leftrightarrow \theta^{k+1} = \theta^k + dt \, \kappa \, L \, \theta^{k+1} + \mathcal{O}\left(N^{-p}\right) \text{ for some } L \in M_N(\mathbb{R}) \tag{2.85}$$

This equation can be rewritten

$$A \, \theta^{k+1} = \theta^k + \mathcal{O}\left(N^{-p}\right) \text{ with } A = I_N - dt \, \kappa \, B \in M_N(\mathbb{R}) \tag{2.86}$$

Using centered finite differences to approximate the derivatives leads to a linear system that is not easy to solve efficiently for a general $n > 1$. In this case A is a sparse Toeplitz matrix [Recktenwald 2004] and it is known that general Toeplitz systems can be solved by the Levinson algorithm in $\mathcal{O}\left(N^2\right)$ operations [Trench 1964] and can be reduced to $\mathcal{O}\left(N^{1.5}\right)$ by using iterative algorithms such as successive over-relaxation [Liu 2002]. The order in space of this method in space will depend on the order of the stencil choosed to approximate the second derivatives and the sparsity of A decreases with increasing order. When $n = 1$, the complexity drops to $\mathcal{O}(N)$ for slightly modified tridiagonal Toeplitz systems by using the Scherman-Morison formula to recover a trigiagonal Toeplitz system along with the Thomas algorithm [Stone 1973].

We can also rewrite equation (2.82) by using the spectral interpolation of $\theta(x)$, denoted $\theta_N(x)$ at time t^k and $t^{k+1} = t^k + dt$ in order to obtain a solution in only $\mathcal{O}(N \log N)$ operations by using two FFTs along with $\mathcal{O}(N)$ posprocessing:

$$\theta_N(x, t^{k+1}) = \theta_N(x, t^k) + dt\,\kappa\,\Delta\theta_N^{k+1}(x) \quad \forall x \in \Omega \tag{2.87}$$

As before, in order to solve this equation, we impose equation (2.87) on the N spatial discretization points $x_{\boldsymbol{j}} = \boldsymbol{i} \odot \boldsymbol{dx}$:

$$\theta_N^{k+1}(\boldsymbol{x_j}) = \theta_N^k(\boldsymbol{x_j}) + dt\,\kappa\,\Delta\theta_N^{k+1}(\boldsymbol{x_j}) \quad \forall \boldsymbol{j} \in \mathcal{J} \tag{2.88}$$

Let $\widehat{\theta^k}$ represent the n-dimensional complex-to-complex DFT of θ^k and $\widehat{\theta^{k+1}}$ be the one of θ^{k+1}. We can rewrite equation 2.88 by identifying the coefficients in the spectral basis:

$$\widehat{\theta_{\boldsymbol{p}}^{k+1}} = \widehat{\theta^k}_{\boldsymbol{p}} + dt\,\kappa\,\Delta\widehat{\theta_{\boldsymbol{p}}^{k+1}} \quad \forall \boldsymbol{p} \in \mathcal{J} \tag{2.89}$$

$$= \widehat{\theta^k}_{\boldsymbol{p}} + dt\,\kappa\left(\sum_{i=1}^{n} k_{p_i}^2\right)\widehat{\theta_{\boldsymbol{p}}^{k+1}} \quad \forall \boldsymbol{p} \in \mathcal{J} \tag{2.90}$$

which can be rewritten

$$\left[1 - dt\,\kappa\left(\sum_{i=1}^{n} k_{p_i}^2\right)\right]\widehat{\theta_{\boldsymbol{p}}^{k+1}} = \widehat{\theta^k}_{\boldsymbol{p}} \quad \forall \boldsymbol{p} \in \mathcal{J} \tag{2.91}$$

$$\Leftrightarrow A\,\widehat{\theta^{k+1}} = \widehat{\theta^k} \text{ with } A = I_N - dt\,\kappa\,\mathrm{diag}\left(k_{\boldsymbol{p}} \cdot k_{\boldsymbol{p}}\right)_{\boldsymbol{p}\in\mathcal{J}} \tag{2.92}$$

$$\text{and } k_{\boldsymbol{p}} = [k_{p_1}, \cdots, k_{p_n}]^T$$

Hence in the frequency space the matrix $A \in M_N(\mathbb{R})$ becomes diagonal due to the properties of the Fourier transform with respect to the derivatives (2.72). This system can be solved in $\mathcal{O}(N)$ operations. Here $\widehat{\theta^k}$ is obtained by DFT in $\mathcal{O}(N \log N)$ operations and θ^{k+1} can be recovered by IDFT in $\mathcal{O}(N \log N)$ operations. The order of convergence of the method is $\mathcal{O}(N^{-p})$ with $p \geq 2$ depending n the smoothness of $\theta^{k+1}(x)$.

Spectral convergence is achieved only for smooth θ:

- Polynomial convergence is achieved for $\theta \in C^\infty$ given its derivatives are all square-integrable. In this case the error is of order $\mathcal{O}(N^{-p})$ for all $p \in \mathbb{N}$.
- Exponential convergence is achieved for $\theta \in C^\omega$, the additional condition being that there exists some constant $C \in \mathbb{R}$ such that the all its spatial derivatives $\theta^{(\boldsymbol{k})}$ are bounded by $C^{k_1+k_2+\cdots+k_n}\,k_1!k_2!\cdots k_n!$ (see [Komatsu 1960]).

In practice those two cases are indistinguishable at the discrete level. Although the diffusive process may help to enhance the local regularity away from the boundaries, this method does not work well for non-periodic boundary conditions because there is no hope of uniform convergence for discountinuous θ. The specific handling of general non-periodic boundaries is described in the last section of this chapter.

The algorithm to compute θ^{k+1} from θ^k is straightforward:

input : $dt, \theta^k = \theta(t^k)$
output: $\theta^{k+1} = \theta(t^k + dt)$, solution of problem (2.81)
$\widehat{\theta^k} \leftarrow \mathcal{F}\left(\theta^k\right)$
for $\boldsymbol{p} \in \mathcal{J}$ **do**
  $\widehat{\theta_{\boldsymbol{p}}^{k+1}} \leftarrow \dfrac{\widehat{\theta_{\boldsymbol{p}}^k}}{1 - dt\,\kappa\,(k_{\boldsymbol{p}} \cdot k_{\boldsymbol{p}})}$
end
$\theta^{k+1} \leftarrow= \mathcal{F}^{-1}\left(\widehat{\theta^{k+1}}\right)$

Algorithm 7: Spectral resolution of a periodic Poisson problem

2.6.3 Vorticity correction and computation of velocity

Let $\boldsymbol{\omega}^*$ be the state of the discretized vorticity after convection, diffusion, stretching and external forcing corresponding to $\boldsymbol{\omega}^{k,4}$ in algorithm 1 or $\boldsymbol{\omega}^{k+\frac{1}{2},4n}$ in algorithm 4. The correction step consists into correcting $\boldsymbol{\omega}^*$ to $\boldsymbol{\omega}^{k+1}$ such that $\boldsymbol{\omega}^{k+1} = \nabla \times \boldsymbol{u}^{k+1}$ where $\boldsymbol{u}^{k+1}$ is divergence free. Once the vorticity has been projected, the divergence-free velocity can be recomputed from vorticity by solving Poisson equation (2.8) spectrally as in the last subsection. To derive required equations, it is convenient to work with the Helmholtz decomposition of the velocity $\boldsymbol{u}^{k+1}$ and the vorticity $\boldsymbol{\omega}^*$:

$$\boldsymbol{u}^{k+1} = \nabla \times \boldsymbol{\psi} + \nabla\phi \tag{2.93}$$

$$\boldsymbol{\omega}^{k+1} = \boldsymbol{\omega}^* + \nabla e \tag{2.94}$$

In addition we are free to impose $\nabla \cdot \boldsymbol{\psi} = 0$ and we want the velocity to stay divergence free at time t^{k+1}:

$$\nabla \cdot \boldsymbol{u}^{k+1} = 0 \tag{2.95}$$

$$\boldsymbol{\omega}^{k+1} = \nabla \times \boldsymbol{u}^{k+1} \tag{2.96}$$

Equations (2.93), (2.94), (2.95) and (2.96) lead to:

$$\nabla \cdot \boldsymbol{\omega}^{k+1} = \nabla \cdot (\nabla \times \boldsymbol{u}^{k+1}) = \nabla \times (\underbrace{\nabla \cdot \boldsymbol{u}^{k+1}}_{=0}) = 0 \tag{2.97}$$

$$\begin{aligned} \nabla \cdot \boldsymbol{u}^{k+1} = \nabla \cdot (\nabla \times \boldsymbol{\psi} + \nabla\phi) = 0 \\ \Rightarrow \Delta\phi = 0 \end{aligned} \tag{2.98}$$

$$\begin{aligned} \nabla \cdot \boldsymbol{\omega}^{k+1} = \nabla \cdot (\boldsymbol{\omega}^* + \nabla e) = 0 \\ \Rightarrow \Delta e = -\nabla \cdot \boldsymbol{\omega}^* \end{aligned} \tag{2.99}$$

The Poisson problem relating to the correction of the vorticity is obtained by applying the Laplace operator to equation (2.94) along with equation (2.99):

$$\Delta\boldsymbol{\omega}^{k+1} = \Delta\left(\boldsymbol{\omega}^* + \nabla e\right) = \Delta\boldsymbol{\omega}^* + \nabla\Delta e = \Delta\boldsymbol{\omega}^* - \nabla(\nabla \cdot \boldsymbol{\omega}^*) \tag{2.100}$$

Finally, the equation relating to $\boldsymbol{\psi}$ is obtained by taking the curl of equation (2.93) along with equation (2.96):

$$\boldsymbol{\omega}^{k+1} = \nabla \times \boldsymbol{u}^{k+1} = \nabla \times \nabla \times \boldsymbol{\psi} + \nabla\phi = \nabla \times \nabla \times \boldsymbol{\psi} = -\Delta\boldsymbol{\psi} + \nabla \underbrace{(\nabla \cdot \boldsymbol{\psi})}_{=0} = -\Delta\boldsymbol{\psi} \tag{2.101}$$

The general algorithm to compute $\boldsymbol{\omega}^{k+1}$ and $\boldsymbol{u}^{k+1}$ from $\boldsymbol{\omega}^*$ is thus the following:

1. Correct $\boldsymbol{\omega}^*$ such that $\boldsymbol{\omega}^{k+1}$ is divergence-free using a Poisson solver:

$$\boxed{\Delta\boldsymbol{\omega}^{k+1} = \Delta\boldsymbol{\omega}^* - \nabla(\nabla \cdot \boldsymbol{\omega}^*)} \tag{2.102}$$

2. Compute $\boldsymbol{u}_0^{k+1}$ using a Poisson solver:

$$\Delta\boldsymbol{\psi} = -\boldsymbol{\omega}^{k+1} \tag{2.103}$$
$$\nabla \cdot \boldsymbol{\psi} = 0 \tag{2.104}$$
$$\boldsymbol{u}_0^{k+1} = \nabla \times \boldsymbol{\psi} \tag{2.105}$$

3. Compute $\boldsymbol{u}_1^{k+1}$ using a Poisson solver:

$$\Delta\phi = 0 \tag{2.106}$$
$$\boldsymbol{u}_1^{k+1} = \nabla\phi \tag{2.107}$$

4. Compute $\boldsymbol{u}^{k+1}$ using the two computed velocities:

$$\boxed{\boldsymbol{u}^{k+1} = \boldsymbol{u}_0^{k+1} + \boldsymbol{u}_1^{k+1}} \tag{2.108}$$

On a fully periodic domain Ω, the only solution to $\boldsymbol{u}_1$ is 0 so step 3 and 4 can be skipped, and step 2 is simplified because we do not have to impose $\nabla \cdot \boldsymbol{\psi} = 0$ on the domain boundaries while solving for $\boldsymbol{\psi}$. Other boundary conditions are taken in account in next subsection. In 2D, $\nabla \cdot \boldsymbol{\omega}^*$ is 0 because $\boldsymbol{\omega}^* = [0, 0, \omega_z^*(x, y)]^T$, so there is no correction to do, and step 1 can also be skipped. Basically this leads to the same algorithm as for the diffusion in the previous subsection, followed by the computation of a curl as exposed in algorithm 8. This algorithm requires one DFT and two IDFTs such that the total cost of the method is proportional to $3N \log N$ with $N = \mathcal{N}_1\mathcal{N}_2$. The 3D case shown in algorithm 9 is a bit more computationally intensive because the vorticity has to be corrected, leading to 3 DFTs and 6 IDFTS for a total cost proportional to $9N \log N$ with $N = \mathcal{N}_1\mathcal{N}_2\mathcal{N}_3$.

input : $\omega^*, (\overline{u_x^{k+1}}, \overline{u_y^{k+1}})$, vorticity and spatial mean of the velocity components
output: $\omega^{k+1}, (u_x^{k+1}, u_y^{k+1})$, divergence free vorticity and velocity
$\omega^{k+1} = \omega^*$
$\widehat{\omega^{k+1}} \leftarrow \mathcal{F}\left(\omega^{k+1}\right)$
for $\boldsymbol{p} = (p_x, p_y) \in [\![0, \mathcal{N}_x[\![\times [\![0, \mathcal{N}_y[\![$ **do**
  if $\boldsymbol{p} = (0,0)$ **then**
    $\left(\widehat{u_{x,\mathbf{0}}^{k+1}}, \widehat{u_{y,\mathbf{0}}^{k+1}}\right) \leftarrow \left(\overline{u_x^{k+1}}, \overline{u_y^{k+1}}\right)$
  else
    $\widehat{\psi_{\boldsymbol{p}}} \leftarrow \dfrac{\widehat{\omega_{\boldsymbol{p}}^{k+1}}}{k_{p_x}^2 + k_{p_y}^2}$
    $\left(\widehat{u_{x,\boldsymbol{p}}^{k+1}}, \widehat{u_{y,\boldsymbol{p}}^{k+1}}\right) \leftarrow \left(-k_{p_y}\widehat{\psi_{\boldsymbol{p}}}, +k_{p_x}\widehat{\psi_{\boldsymbol{p}}}\right)$
  end
end
$\left(u_x^{k+1}, u_y^{k+1}\right) \leftarrow \left(\mathcal{F}^{-1}\left(\widehat{u_x^{k+1}}\right), \mathcal{F}^{-1}\left(\widehat{u_y^{k+1}}\right)\right)$

Algorithm 8: Algorithm to compute divergence-free vorticity and velocity on a 2D fully periodic domain discretized on $\boldsymbol{\mathcal{N}} = (\mathcal{N}_y, \mathcal{N}_x)$ points where $\mathcal{N}_x$ and $\mathcal{N}_y$ are odd.

input : $(\omega_x^*, \omega_y^*, \omega_z^*), (\overline{u_x^{k+1}}, \overline{u_y^{k+1}}, \overline{u_z^{k+1}})$, vorticity and spatial mean of the velocity
output: $(\omega_x^{k+1}, \omega_y^{k+1}, \omega_z^{k+1}), (u_x^{k+1}, u_y^{k+1}, u_z^{k+1})$, divergence free vorticity and velocity
$(\widehat{\omega_x^*}, \widehat{\omega_y^*}, \widehat{\omega_z^*}) \leftarrow \left(\mathcal{F}\left(\omega_x^*\right), \mathcal{F}\left(\omega_y^*\right), \mathcal{F}\left(\omega_z^*\right)\right)$
for $\boldsymbol{p} = (p_x, p_y, p_z) \in [\![0, \mathcal{N}_x[\![\times [\![0, \mathcal{N}_y[\![\times [\![0, \mathcal{N}_z[\![$ **do**
  if $\boldsymbol{p} = (0,0,0)$ **then**
    $\left(\widehat{\omega_{x,\mathbf{0}}^{k+1}}, \widehat{\omega_{y,\mathbf{0}}^{k+1}}, \widehat{\omega_{z,\mathbf{0}}^{k+1}}\right) \leftarrow \left(\widehat{\omega_{x,\mathbf{0}}^*}, \widehat{\omega_{y,\mathbf{0}}^*}, \widehat{\omega_{z,\mathbf{0}}^*}\right)$
    $\left(\widehat{u_{x,\mathbf{0}}^{k+1}}, \widehat{u_{y,\mathbf{0}}^{k+1}}, \widehat{u_{z,\mathbf{0}}^{k+1}}\right) \leftarrow \left(\overline{u_x^{k+1}}, \overline{u_y^{k+1}}, \overline{u_z^{k+1}}\right)$
  else
    $\left(\widehat{\omega_{x,\boldsymbol{p}}^{k+1}}, \widehat{\omega_{y,\boldsymbol{p}}^{k+1}}, \widehat{\omega_{y,\boldsymbol{p}}^{k+1}}\right) \leftarrow \left(\widehat{\omega_{x,\boldsymbol{p}}^*}, \widehat{\omega_{y,\boldsymbol{p}}^*}, \widehat{\omega_{z,\boldsymbol{p}}^*}\right) - \boldsymbol{k_p}\left[\dfrac{k_{p_x}\widehat{\omega_{x,\boldsymbol{p}}^*} + k_{p_y}\widehat{\omega_{y,\boldsymbol{p}}^*} + k_{p_z}\widehat{\omega_{z,\boldsymbol{p}}^*}}{k_{p_x}^2 + k_{p_y}^2 + k_{p_z}^2}\right]$
    $\left(\widehat{\psi_{x,\boldsymbol{p}}}, \widehat{\psi_{y,\boldsymbol{p}}}, \widehat{\psi_{z,\boldsymbol{p}}}\right) \leftarrow \left(\dfrac{\widehat{\omega_{x,\boldsymbol{p}}^{k+1}}}{k_{p_x}^2 + k_{p_y}^2 + k_{p_z}^2}, \dfrac{\widehat{\omega_{y,\boldsymbol{p}}^{k+1}}}{k_{p_x}^2 + k_{p_y}^2 + k_{p_z}^2}, \dfrac{\widehat{\omega_{z,\boldsymbol{p}}^{k+1}}}{k_{p_x}^2 + k_{p_y}^2 + k_{p_z}^2}\right)$
    $\left(\widehat{u_{x,\boldsymbol{p}}^{k+1}}, \widehat{u_{y,\boldsymbol{p}}^{k+1}}, \widehat{u_{z,\boldsymbol{p}}^{k+1}}\right) \leftarrow \left(k_{p_z}\widehat{\psi_{y,\boldsymbol{p}}} - k_{p_y}\widehat{\psi_{z,\boldsymbol{p}}},\ k_{p_x}\widehat{\psi_{z,\boldsymbol{p}}} - k_{p_z}\widehat{\psi_{x,\boldsymbol{p}}},\ k_{p_y}\widehat{\psi_{x,\boldsymbol{p}}} - k_{p_x}\widehat{\psi_{y,\boldsymbol{p}}}\right)$
  end
end
$\left(u_x^{k+1}, u_y^{k+1}, u_z^{k+1}\right) \leftarrow \left(\mathcal{F}^{-1}\left(\widehat{u_x^{k+1}}\right), \mathcal{F}^{-1}\left(\widehat{u_y^{k+1}}\right), \mathcal{F}^{-1}\left(\widehat{u_z^{k+1}}\right)\right)$
$\left(\omega_x^{k+1}, \omega_y^{k+1}, \omega_z^{k+1}\right) \leftarrow \left(\mathcal{F}^{-1}\left(\widehat{\omega_x^{k+1}}\right), \mathcal{F}^{-1}\left(\widehat{\omega_y^{k+1}}\right), \mathcal{F}^{-1}\left(\widehat{\omega_z^{k+1}}\right)\right)$

Algorithm 9: Algorithm to compute divergence-free vorticity and velocity on a 3D fully periodic domain discretized on $\boldsymbol{\mathcal{N}} = (\mathcal{N}_z, \mathcal{N}_y, \mathcal{N}_x)$ points with $\mathcal{N}_* \in 2\mathbb{N} + 1$.

2.6.4 Real to real transforms and homogeneous boundary conditions

It is possible to extend the validity of algorithms 7, 8 and 9 to homogeneous Dirichlet and homogeneous Neumann boundary conditions by periodizing the signal by introducing left and right even or odd extensions [Martucci 1994]. In order for this to work, the discrete Fourier transform (DFT) has to be replaced by either a discrete sine transform (DST) or a discrete cosine transform (DCT) corresponding to the required boundary condition. The distinction between a DST or a DCT and a DFT is that the former use only sine or cosine functions, while the latter uses both cosines and sines with the frequencies halved. Given N discrete equispaced samples $f_j = f(x_j)$ of a non-periodic function $f : [0, L] \to \mathbb{R}$, the signal can be periodized in various ways, each of them imposing a particular boundary condition. Each boundary can be either even or odd (2 choices per boundary) and can be symmetric about a data point or the point halfway between two data points (also 2 choices per boundary). This gives a total of 16 transforms, half of these possibilities where the left extension is even corresponding to the 8 types of DCTs and the other half where the left extension is odd to the 8 types of DSTs. In practice, boundary conditions are imposed on collocated or staggered grids so only 4 types of DCTs and DSTs are of interest in order to solve PDEs.

The four transforms for which the homogeneous boundary conditions can be prescribed at the grid points (collocated grid) are given bellow:

Left-right B.C.	**Extensions**	**Transform**	**Unnormalized discrete transform**
Dirichlet-Dirichlet	odd-odd	`DST-I`	$\widehat{f_p} = \sum_{j=0}^{N-1} 2 f_j \sin\left(\frac{\pi}{N+1}(p+1)(j+1)\right)$
Dirichlet-Neumann	odd-even	`DST-III`	$\widehat{f_p} = \sum_{j=0}^{N-1} (2-\delta_{j,N-1}) f_j \sin\left(\frac{\pi}{N}(p+\frac{1}{2})(j+1)\right)$
Neumann-Dirichlet	even-odd	`DCT-III`	$\widehat{f_p} = \sum_{j=0}^{N-1} (2-\delta_{j,0}) f_j \cos\left(\frac{\pi}{N}(p+\frac{1}{2})(j)\right)$
Neumann-Neumann	even-even	`DCT-I`	$\widehat{f_p} = \sum_{j=0}^{N-1} (2-\delta_{j,0}-\delta_{j,N-1}) f_j \cos\left(\frac{\pi}{N-1}(p)(j)\right)$

The four other transforms correspond to boundaries prescribed in between two grid points:

Left-right B.C.	**Extensions**	**Transform**	**Unnormalized discrete transform**
Dirichlet-Dirichlet	odd-odd	`DST-II`	$\widehat{f_p} = 2\sum_{j=0}^{N-1} f_j \sin\left(\frac{\pi}{N}(p+1)(j+\frac{1}{2})\right)$
Dirichlet-Neumann	odd-even	`DST-IV`	$\widehat{f_p} = 2\sum_{j=0}^{N-1} f_j \sin\left(\frac{\pi}{N}(p+\frac{1}{2})(j+\frac{1}{2})\right)$
Neumann-Dirichlet	even-odd	`DCT-IV`	$\widehat{f_p} = 2\sum_{j=0}^{N-1} f_j \cos\left(\frac{\pi}{N}(p+\frac{1}{2})(j+\frac{1}{2})\right)$
Neumann-Neumann	even-even	`DCT-II`	$\widehat{f_p} = 2\sum_{j=0}^{N-1} f_j \cos\left(\frac{\pi}{N}(p)(j+\frac{1}{2})\right)$

Choosing one of those transforms impose the left and right boundary conditions of the spectral interpolant f_N for a given function f. In other words, f is projected into a basis that verify the chosen boundary conditions. The eight different cases are shown on the following figure:

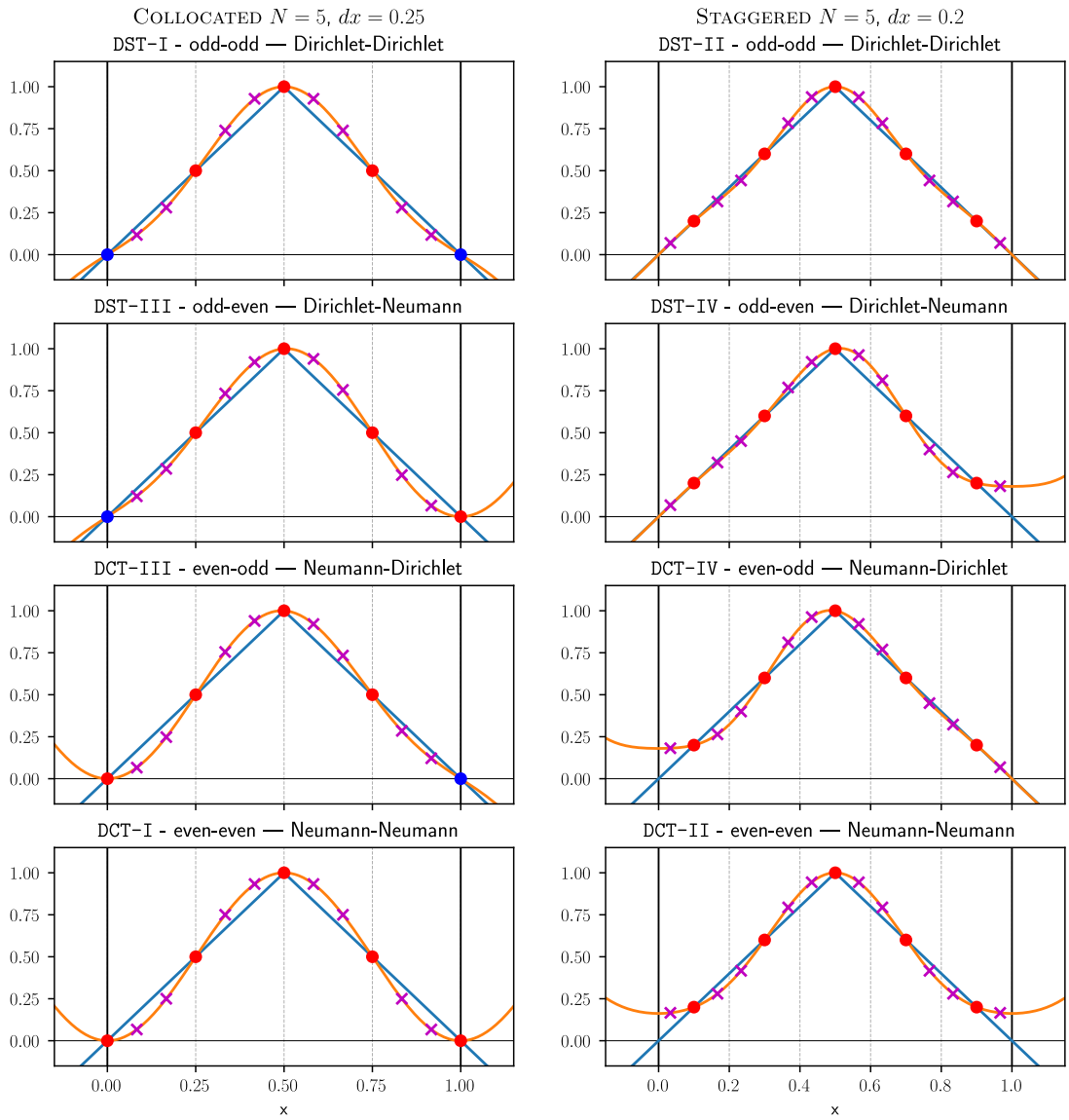


Figure 2.15 – Spectral interpolation by using the eight real-to-real transforms
Interpolation of $f(x) = 1 - 2|x - 0.5|$ on interval $[0, 1]$ by using eight different sine and cosine transforms each corresponding to specific homogeneous boundary conditions. Red dots represent the five samples $f_j = f(x_j)$ used to compute the forward transform and blue dots represent implicit zeros due to the homogeneous Dirichlet boundary conditions (those points are thus not used in the forward and backward transforms). All leftmost plots represent the four collocated boundary configurations while the rightmost plots represent the four staggered boundary configurations. The function f is represented in blue and its interpolant f_N obtained by evaluating (2.109) (orange). The magenta crosses are obtained efficiently with the corresponding backward transforms of roughly three times the size of the forward transform. The effective size of the forward and backward transforms is not the same in all cases (3,4,4,5) top-to-bottom for the collocated cases and (5,5,5,5) for the staggered cases.

Upper part of figure 2.15 shows the even and odd extensions for the transforms of interest where the boundary conditions are imposed on the grid. Other transforms are however required to compute the corresponding inverse transforms. Let $(\phi_p)_{p\in[\![0,N-1]\!]}$ be the basis of sines or cosines of the considered transform, the spectral interpolant of $f(x)$ is given by:

$$f(x) \simeq f_N(x) = \sum_{p=0}^{N-1} \widehat{f_p}\phi_p(x) \tag{2.109}$$

The reconstruction basis are summarized in the following table:

Transform	**Inverse**	**Odd inv.**	s	k_p	**Basis functions** $\phi_p(x)$
`DST-I`	`DST-I`	`DCT-I`	0	$\pi(p+1)/L$	$2\sin(k_p\,x)$
`DST-II`	`DST-III`	`DCT-III`	0	$\pi(p+1)/L$	$(2-\delta_{p,N-1})\sin(k_p\,x)$
`DST-III`	`DST-II`	`DCT-II`	0	$\pi\,(p+\frac{1}{2})/L$	$2\sin(k_p\,x)$
`DST-IV`	`DST-IV`	`DCT-IV`	0	$\pi\,(p+\frac{1}{2})/L$	$2\sin(k_p\,x)$
`DCT-I`	`DCT-I`	`DST-I`	1	$\pi\,p/L$	$(2-\delta_{p,0}-\delta_{p,N-1})\cos(k_p\,x)$
`DCT-II`	`DCT-III`	`DST-III`	1	$\pi\,p/L$	$(2-\delta_{p,0})\cos(k_p\,x)$
`DCT-III`	`DCT-II`	`DST-II`	1	$\pi\,(p+\frac{1}{2})/L$	$2\cos(k_p\,x)$
`DCT-IV`	`DCT-IV`	`DST-IV`	1	$\pi\,(p+\frac{1}{2})/L$	$2\cos(k_p\,x)$

This brings to six the total number of required transforms to handle all homogeneous boundary conditions on a collocated grid (`DCT-I`, `DCT-II`, `DCT-III`, `DST-I`, `DST-II` and `DST-III`). All those transforms will be referred to as real-to-real transforms as opposed to the DFT and the IDFT which are complex-to-complex, real-to-complex or complex-to-real transforms.

The computation of even derivatives is done similarly to the DFT, by multiplying the Fourier coefficients $\widehat{\boldsymbol{f}}$ by some constant wavenumbers $\boldsymbol{k}$ that depends on p and by applying the corresponding inverse transform. For odd derivatives, things get a little more complicated because of the change of basis: sines become cosines, and cosines become sines. Thus the inverse transform to compute odd derivatives is not the same as the inverse transform used to compute even derivatives (the standard backward transform). This specific odd derivative transform will be referred to as the odd inverse transform. All inverse and odd inverse transforms are summarized in the previous table. As for the DFT, all the inverse transforms have to be scaled according to some implementation dependent normalization factor that depends on the transform size N. The size of the transform may be less than the grid size $\mathcal{N}$ because of implicit zeros enforced by the homogeneous Dirichlet boundary conditions as it can be seen on figure 2.15.

$$\begin{aligned} f^{(q)}(x) \simeq f_N^{(q)}(x) &= \sum_{p=0}^{N-1} \widehat{f_p}\phi_p^{(q)}(x) \\ &= \sum_{p=0}^{N'-1} \widehat{f'_p}\varphi_p(x) \end{aligned} \tag{2.110}$$

For even derivatives $\phi_p = \varphi_p$ and $N = N'$ but for odd derivatives ϕ_p and N may be different than φ_p and N'. The odd collocated case can be especially tricky to handle depending on the considered transforms, because of the change of the basis and the number of coefficients.

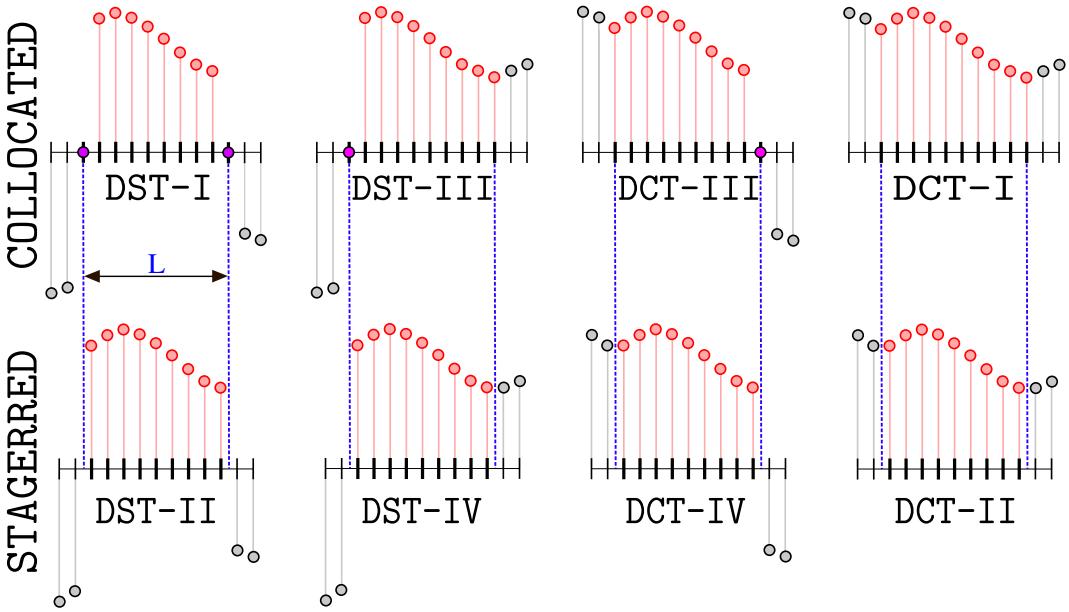


Figure 2.16 – Zoom on the symmetries exhibited by the different transforms. In each case, we look at a DCT or DST of $\mathcal{N}^v = 10$ input samples for the collocated case and $\mathcal{N}^c = 9$ for the staggered case (shown as red and purple dots). Purple dots corresponds to implicit zeros enforced by the sine or cosine basis ϕ_p. The size of the transform N is the number of samples minus the number of collocated Dirichlet boundary conditions. The data extension, shown as gray dots, is the result of applying left and right even or odd extensions enforcing the left and right homogeneous boundary conditions. Image adapted from DCT-symmetries and DST-symmetries from Steven G. Johnson, licensed under CC BY-SA 3.0.

Here because there is no periodicity, $\mathcal{P} = \mathbf{0}$ and the grid size corresponds to the number of grid vertices $\mathcal{N} = \mathcal{N}^v = \mathcal{N}^c + 1$, but the presence of homogeneous Dirichlet boundary conditions may alter the size N of the corresponding transform in the collocated case:

1. **Collocated grid:** boundaries are prescribed on the grid vertices. In this case we have $dx = L/(\mathcal{N} - 1)$ and $x_j = j\,dx$ with $j \in \mathcal{J} \subset [\![0, \mathcal{N} - 1]\!]$:
   (a) **Dirichlet-Dirichlet:** $j \in [\![1, \mathcal{N} - 2]\!]$, the forward `DST-I` transform is of size $N = \mathcal{N} - 2$ with $dx = L/(N + 1)$ because of implicit $x_0 = x_{\mathcal{N}-1} = 0$.
   (b) **Dirichlet-Neumann:** $j \in [\![1, \mathcal{N} - 1]\!]$, the forward `DST-III` transform is of size $N = \mathcal{N} - 1$ with $dx = L/N$ because of implicit $x_0 = 0$.
   (c) **Neumann-Dirichlet:** $j \in [\![0, \mathcal{N} - 2]\!]$, the forward `DCT-III` transform is of size $N = \mathcal{N} - 1$ with $dx = L/N$ because of implicit $x_{\mathcal{N}-1} = 0$.
   (d) **Neumann-Neumann:** $j \in [\![0, \mathcal{N} - 1]\!]$, the forward `DCT-I` transform is of size $N = \mathcal{N}$ with $dx = L/(N - 1)$.

2. **Staggered grid:** Here the boundary conditions are imposed in between two grid points so the transforms are all of size $N = \mathcal{N}^c = \mathcal{N}^v - 1 = \mathcal{N} - 1$. For all the cases we have $dx = L/N = L/(\mathcal{N} - 1)$ and $x_j = (j + \frac{1}{2})\, dx$ with $j \in [\![0, \mathcal{N} - 2]\!]$.

Computation of derivatives

Let $q \in 2\mathbb{N}$, the computation of the even q-th derivative in the frequency space is done as in the Fourier space by elementwise multiplication of $\widehat{\boldsymbol{f}}$ by the real wavenumbers $\boldsymbol{k} = (k_p)_{p \in [\![0, N-1]\!]}$ corresponding to the transform: $\boldsymbol{k}' = (-1)^{\lfloor (s+q)/2 \rfloor} \boldsymbol{k}^q$. Here $\boldsymbol{k}^q = \boldsymbol{k} \odot \boldsymbol{k} \odot \cdots \odot \boldsymbol{k}$ is defined as the elementwise q-th power of $\boldsymbol{k}$ and s and $\boldsymbol{k}$ are forward transform dependent quantities that are summarized in the table under equation (2.109). The values of the derivatives are obtained at the same points than for the forward transform.

For odd derivatives, $q \in 2\mathbb{N} + 1$, the left and right boundaries are swapped due to the change of basis and this may change the size of the transform in the collocated Dirichlet-Dirichlet and Neumann-Neumann cases. Here we describe the algorithm to compute odd derivatives for all collocated cases: Let d_l and d_r represent the setup of left and right boundaries ($d_l = 1$ when there is a left homogeneous boundary and 0 otherwise, and similarly for the right boundary condition). For a collocated boundary setup, the size of the forward transform is $N = \mathcal{N} - d_l - d_r$ and the size of the odd derivative backward transform is $N' = \mathcal{N} - (1 - d_l) - (1 - d_r) = \mathcal{N} - d'_l - d'_r$. The location x'_j of the N' resulting samples $f^{(q)}(x'_j)$ correspond to the ones associated to the odd derivative forward transform:

$$dx = L/(\mathcal{N} - 1) \tag{2.111a}$$

$$x_j = j\, dx \quad \forall j \in [\![d_l, \mathcal{N} - d_r[\![\tag{2.111b}$$

$$x'_j = j\, dx \quad \forall j \in [\![d'_l, \mathcal{N} - d'_r[\![\tag{2.111c}$$

$$k'_p = (-1)^{\lfloor (s+q)/2 \rfloor} k_p^q \quad \forall p \in [\![0, N[\![\tag{2.111d}$$

$$\widehat{f'_p} = \begin{cases} 0 & \text{if } p = d_l d_r - 1 \\ 0 & \text{if } p = N' - d_l d_r \\ k'_{p+d'_l d'_r} \widehat{f'}_{p+d'_l d'_r} & \text{else} \end{cases} \quad \forall p \in [\![0, N'[\![\tag{2.111e}$$

Convergence of the method

The real-to-real transforms are subject to the same convergence properties as the DFT when considering the discrete signal obtained by applying left and right extensions. For the DFT we saw that implicit periodicity of the transform meant that discontinuities usually occur at the boundaries because any random segment of a signal is unlikely to have the same value at both the left and right boundaries. A similar problem arises for all real-to-real transforms that happens to impose en odd symmetry (an homogeneous Dirichlet boundary condition). For those six transforms there is a discontinuity for any function that does not happen to be zero at Dirichlet boundaries. When there is such a discontinuity, there is no expected

uniform convergence of the interpolant ($\|f - f_N\|_\infty = \mathcal{O}(1)$) so we cannot use this method for interpolation. Because of the integrations we may however expect to uniformly converge with second order to the solution of a Poisson equation (algorithm 7) and with first order to the solution of a Poisson-curl problem (algorithm 8 and 9). In practice we will only use those last two algorithms when the vorticity is zero at the Dirichlet boundaries such that the order only depend on the smoothness of the vorticity. In contrast, the Neumann-Neumann cases where both extensions are even always yields a continuous extension at the boundaries and we recover exponential convergence with any smooth input. This is why `DCT-I` and `DCT-II` generally perform better for interpolation and signal compression than other real-to-real transforms.

Multi-dimensional transforms

Let $\mathscr{F} = \mathscr{F}_1 \circ \mathscr{F}_2 \circ \cdots \circ \mathscr{F}_n$ denote a n-dimensional real-to-real or real-to-complex forward transform with associated real-to-real or complex-to-real backward transform $\mathscr{F}^{-1} = \mathscr{F}_n^{-1} \circ \cdots \circ \mathscr{F}_1^{-1}$. Let $\mathcal{S}_X$ and $\mathcal{C}_X$ with $X \in \{\mathtt{I}, \mathtt{II}, \mathtt{III}\}$ denote the six required sine and cosine transforms and $\mathcal{S}_X^{-1}$ and $\mathcal{C}_X^{-1}$ their corresponding inverse transforms. We recall that the complex-to-complex DFT is denoted $\mathcal{F}$ and the real-to-complex DFT with Hermitian symmetry is denoted $\mathcal{H}$. With these notations we have $\mathcal{F}_i \in \{\mathcal{F}, \mathcal{H}, \mathcal{S}_I, \mathcal{S}_{II}, \mathcal{S}_{III}, \mathcal{C}_I, \mathcal{C}_{II}, \mathcal{C}_{III},\}$ and $\mathcal{F}_i^{-1} \in \left\{\mathcal{F}^{-1}, \mathcal{H}^{-1}, \mathcal{S}_I^{-1}, \mathcal{S}_{II}^{-1}, \mathcal{S}_{III}^{-1}, \mathcal{C}_I^{-1}, \mathcal{C}_{II}^{-1}, \mathcal{C}_{III}^{-1},\right\}$ providing reconstruction basis $(\phi_{i,p})_{p\in[\![0,\mathcal{M}_i[\![}$ where $\boldsymbol{\mathcal{M}}$ represent the size of the transform. In practice the axes of the problem are reordered for performance reasons and the permutation is such that all the real-to-real transforms come first (non-periodic axes), followed an Hermitian real-to-complex DFT for the first periodic axis, and complex-to-complex DFTs for all the subsequent periodic axes. The first transform being done on axis x (the last one, contiguous in memory), such a configuration yields a periodicity mask of the form $\boldsymbol{\mathcal{P}} = (1, \cdots, 1, 0, \cdots, 0)$. We recall that the number of grid vertices (the grid size) is defined by $\boldsymbol{\mathcal{N}}^v = \boldsymbol{\mathcal{N}}^c + 1 - \boldsymbol{\mathcal{P}}$ where $\boldsymbol{\mathcal{N}}^c$ represent the number of grid cells and that $\boldsymbol{dx} = \boldsymbol{L} \oslash \boldsymbol{\mathcal{N}}^c$. This defines nodes $\boldsymbol{x_j} = \boldsymbol{x}_{start} + \boldsymbol{j} \odot \boldsymbol{dx} \quad \forall \boldsymbol{j} \in \mathcal{J} = [\![0, \mathcal{N}_1^v[\![\times \cdots \times [\![0, \mathcal{N}_n^v[\![$.

Let $\boldsymbol{d}_l = (d_{i,l})_{i\in[\![1,n]\!]}$ and $\boldsymbol{d}_r = (d_{i,r})_{i\in[\![1,n]\!]}$ such that $d_{i,l}$ (respectively $d_{i,r}$) denote whether problem imposes a left (respectively a right) homogeneous boundary condition on the i-th axis. The size of the forward transform is $\boldsymbol{\mathcal{M}} = \boldsymbol{\mathcal{N}} - \boldsymbol{d}_l - \boldsymbol{d}_r$ because of the implicit hyperplanes containing only zero-valued nodes due the homogeneous Dirichlet boundary conditions. Let $\boldsymbol{f} = (f(x_{\boldsymbol{j}}))_{\boldsymbol{j}\in\mathcal{K}}$ with $\mathcal{K} = [\![d_{1,l}, \mathcal{N}_1 - d_{1,r}[\![\times \cdots \times [\![d_{n,l}, \mathcal{N}_n - d_{n,r}[\![$ be the $M = \prod_{i=1}^n \mathcal{M}_i$ samples such that $\widehat{\boldsymbol{f}} = \mathscr{F}(\boldsymbol{f}) = (\widehat{f}_{\boldsymbol{p}})_{\boldsymbol{p}\in\mathcal{K}}$. The spectral interpolant f_M is defined similarly to the multi-dimensional DFT as:

$$f(\boldsymbol{x}) \simeq f_M(\boldsymbol{x} - \boldsymbol{x}_{start}) = \sum_{p_1=0}^{\mathcal{M}_1-1} \cdots \sum_{p_n=0}^{\mathcal{M}_n-1} \left[\prod_{i=1}^{n} \phi_{i,p_i}(\boldsymbol{x})\right] \widehat{f}_{p_1,\cdots,p_n} \tag{2.112}$$

Each forward directional transform $\mathscr{F}_i$ has associated diagonal derivative matrices of any given order $q \in \mathbb{N}$ such that in general $D_i^{(q)}$ is not square and has shape $(\mathcal{M}_i', \mathcal{M}_i)$, $D_i^{(q)} \neq \left(D_i^{(1)}\right)^q$ defined by (2.72) and (2.111). For all the transforms, those matrices depend on the parity

of q, the simplest case being that for an even DFT derivative $D_i^{(q)} = \mathrm{diag}(k_0, \cdots, k_{\mathcal{N}_i-1})^q \in M_{\mathcal{N}_i}(\mathbb{C})$ where the k_i are the associated complex wavenumbers. Let $(\mathscr{F}')^{-1}$ denote the inverse transform required to compute the $\boldsymbol{q}$-th derivative. $(\mathscr{F}')^{-1}$ may be different than $\mathscr{F}^{-1}$ due to the change of basis entailed by odd derivatives on non-periodic axes. This also modifies the points x'_j where the derivative samples are recovered and we define $\boldsymbol{d}'_l = \mathbf{1} - \boldsymbol{d}_l$, $\boldsymbol{d}'_r = \mathbf{1} - \boldsymbol{d}_r$, $\boldsymbol{\mathcal{M}}' = \boldsymbol{\mathcal{N}} - \boldsymbol{d}'_l - \boldsymbol{d}'_r$, $M' = \prod_{i=1}^n \mathcal{M}_i$ and $\mathcal{K}' = [\![d'_{1,l}, \mathcal{N}_1 - d'_{1,r}[\![\times \cdots \times [\![d'_{n,l}, \mathcal{N}_n - d'_{n,r}[\![$.
With those notations we have:

$$\boldsymbol{f} = f(\boldsymbol{x_j}) \quad \boldsymbol{j} \in \mathcal{K} \tag{2.113a}$$

$$\boldsymbol{f}' \simeq \frac{\partial^{q_1+\cdots+q_n} f}{\partial x_1^{q_1} \cdots \partial x_n^{q_n}}(\boldsymbol{x_j}) \quad \boldsymbol{j} \in \mathcal{K}' \tag{2.113b}$$

$$\widehat{\boldsymbol{f}} = \mathscr{F}(\boldsymbol{f}) \tag{2.113c}$$

$$\widehat{\boldsymbol{f}'} = K\,\widehat{\boldsymbol{f}} \tag{2.113d}$$

$$\boldsymbol{f}' = (\mathscr{F}')^{-1}(\widehat{\boldsymbol{f}'}) \tag{2.113e}$$

$$K = D_1^{(q_1)} \otimes D_2^{(q_2)} \otimes \cdots \otimes D_n^{(q_n)} \in M_{M',M}(\mathbb{K}) \text{ with } \mathbb{K} \in \{\mathbb{R}, \mathbb{C}\} \tag{2.113f}$$

In practice the matrices $D_i^{(q_i)}$ and K are never built and we proceed by iterating on the spectral coefficients as in the pure periodic case. From now on for simplicity we consider only the case $M = M'$ where $D_i^{(q)} = \mathrm{diag}(d_{i,0}^{(q)}, \cdots, d_{i,\mathcal{M}_i-1}^{(q)})$ is always a square diagonal matrix.

Modification of the algorithms to handle homogeneous boundary conditions

Spectral diffusion algorithm 7 only use second derivatives and requires minimal modification to handle all real-to-real transforms:

input : $dt, \theta^k = \theta(t^k)$
output: $\theta^{k+1} = \theta(t^k + dt)$, solution of problem (2.81)
$\widehat{\theta^k} \leftarrow \mathscr{F}\left(\theta^k\right)$
for $\boldsymbol{p} \in \mathcal{K}$ **do**
$$\widehat{\theta_{\boldsymbol{p}}^{k+1}} \leftarrow \frac{\widehat{\theta_{\boldsymbol{p}}^k}}{1 - dt\,\kappa \sum_{i=1}^n d_{i,p_i}^{(2)}}$$
end
$\theta^{k+1} \leftarrow= \mathscr{F}^{-1}\left(\widehat{\theta^{k+1}}\right)$

Algorithm 10: Spectral diffusion with mixed periodic and homogeneous Dirichlet/Neumann boundaries

Algorithm 8 and 9 that solve the Poisson-curl problem have to be modified such that the mean velocity can only be imposed for fully periodic problems, fully homogeneous Neumann boundaries problems or a mix of the two (which is coherent with the fact those are the only transforms for which $d_0^{(q)} = 0$). Moreover, the presence of odd derivatives may introduce zero-padding and offsets as seen in (2.111) on every axes.

For clarity those extra offsets have not been included in the following modified algorithm:

input : $\omega^*, (\overline{u_x^{k+1}}, \overline{u_y^{k+1}})$, vorticity and *weighted* mean of the velocity components
output: $\omega^{k+1}, (u_x^{k+1}, u_y^{k+1})$, divergence free vorticity and velocity
$\omega^{k+1} = \omega^*$
$\widehat{\omega^{k+1}} \leftarrow \mathscr{F}\left(\omega^{k+1}\right)$
for $\boldsymbol{p} \in \mathcal{K}$ **do**
  if $p_1 = p_2 = 0$ ***and*** $d_{1,0}^{(2)} + d_{2,0}^{(2)} = 0$ **then**
    $\left(\widehat{u_{x,\mathbf{0}}^{k+1}}, \widehat{u_{y,\mathbf{0}}^{k+1}}\right) \leftarrow \left(\overline{u_x^{k+1}}, \overline{u_y^{k+1}}\right)$ *//Only happens for periodic/Neumann BCs*
  else
    $\widehat{\psi_{\boldsymbol{p}}} \leftarrow \dfrac{\widehat{\omega_{\boldsymbol{p}}^{k+1}}}{d_{1,p_1}^{(2)} + d_{2,p_2}^{(2)}}$
    $\left(\widehat{u_{x,\boldsymbol{p}}^{k+1}}, \widehat{u_{y,\boldsymbol{p}}^{k+1}}\right) \leftarrow \left(-d_{1,p_1}^{(1)}\widehat{\psi_{\boldsymbol{p}}}, +d_{2,p_2}^{(1)}\widehat{\psi_{\boldsymbol{p}}}\right)$
  end
end
$\left(u_x^{k+1}, u_y^{k+1}\right) \leftarrow \left((\mathscr{F}'_{\partial_y})^{-1}\left(\widehat{u_x^{k+1}}\right), (\mathscr{F}'_{\partial_x})^{-1}\left(\widehat{u_y^{k+1}}\right)\right)$

Algorithm 11: Algorithm to compute divergence-free vorticity and velocity on a 2D domain with a mix of periodic and homogeneous boundary conditions.

2.7 Extension to general non-periodic boundary conditions

In order to solve spectrally Poisson problems with general boundary conditions, the spectral basis has once again to be changed. We already saw that because of the Gibbs phenomenon, non-periodic signals cannot be reconstructed accurately by IDFT, the error being of the order of the jump discontinuity (2.75). The same goes for real-to-real transforms enforcing Dirichlet boundary conditions where the signal does not happen to be zero on the said boundaries. In those problematic cases, increasing the number of samples N does not improve accuracy because $\|f - f_N\|_\infty = \mathcal{O}(1)$ as shown on figure 2.14 for L^∞ convergence results.

In this subsection we propose to solve the following one-dimensional problem where $\mathscr{L}(\phi, \partial_x\phi, \partial_{x^2}\phi, \cdots)$ is a linear operator with left and right boundary conditions specified:

$$\mathscr{L}\,\phi = f \text{ on } [0, L] \tag{2.114a}$$

$$\alpha_l\,\phi(0) + \beta_l\,\frac{\partial\phi}{\partial x}(0) = \gamma_l \tag{2.114b}$$

$$\alpha_r\,\phi(L) + \beta_r\,\frac{\partial\phi}{\partial x}(L) = \gamma_r \tag{2.114c}$$

$$(\alpha_l, \alpha_r) \neq (0, 0) \tag{2.114d}$$

Boundary condition	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$
Homogeneous Dirichlet	1	0	0
Homogeneous Neumann	0	1	0
Dirichlet	1	0	γ
Neumann	0	1	γ
Robin	α	β	γ

2.7.1 General spectral methods

As already seen for the periodic case, the idea behind a spectral method is to approximate a solution $f(x)$ by a finite sum:

$$f_N(x) = \sum_{p=0}^{N} \widehat{f_p}\phi_p(x) \tag{2.115a}$$

$$\widehat{f_p} = \int_a^b h(x)f(x)\phi_p(x)\mathrm{d}x \tag{2.115b}$$

where $\{\phi_p\}_{p\in[\![0,N]\!]}$ is a set of orthogonal basis functions with respect to some weighting function $h(x)$ and where f represent a square integrable function with respect to h, noted as $f \in L^2_h$. Successful expansion basis meets the following requirements:

1. **Convergence:** The approximation $f_N(x)$ should converge uniformly to $f(x)$ as $N \to \infty$.
2. **Differentiation:** Given coefficients $\widehat{\boldsymbol{f}}$, it should be easy to determine the set of coefficients $\widehat{\boldsymbol{f'}}$ representing some derivative:

$$f_N^{(q)}(x) = \sum_{p=0}^{N} \widehat{f_p}\phi_p^{(q)}(x) \simeq \sum_{p=0}^{N} \widehat{f'_p}\phi_p(x) \tag{2.116}$$

3. **Transformation:** The computation of expansion coefficients $\widehat{f}$ from function values $u_j = u(x_j)$ for $j \in [\![0, N]\!]$ should be algorithmically efficient.

Spectral accuracy for non-periodic functions is generally achieved trough the use of Fourier series in orthogonal polynomials on some interval (a, b). All of the classical orthogonal polynomials [Gautschi 2004] can be reduced to one of the following four types:

Polynomial	Symbol	Weight $h(x)$	Interval [a,b]	Domain
Trigonometric	$P_p(x)$	1	$[0, 2\pi[$	periodic
Jacobi	$P_p^{\alpha,\beta}(x)$	$(1-x)^\alpha(1+x)^\beta$	$[-1, +1]$	bounded
Laguerre	$L_p^\alpha(x)$	$x^\alpha \mathrm{e}^{-x}$	$[0, +\infty[$	semi-bounded
Hermite	$H_p(x)$	e^{-x^2}	$]-\infty, +\infty[$	unbounded

As we only need to solve PDEs on bounded domains [0,L] the natural choice is to use Jacobi polynomials and their derivatives when the domain is not periodic.

Jacobi polynomials

Jacobi polynomials [Jacobi 1859] are useful to handle non-periodic functions on bounded domains $\Omega = [0, L]$ with a simple change of variable to $[-1, 1]$. The class of orthonormal Jacobi polynomials are defined for $\alpha > -1$ and $\beta > -1$ by the following formula:

$$P_p^{\alpha,\beta}(x) = \sqrt{\omega_p^{\alpha,\beta}} \frac{(-1)^p}{p! 2^p} (1-x)^{-\alpha}(1+x)^{-\beta} \frac{\mathrm{d}^p}{\mathrm{d}x^p} \left[(1-x)^\alpha (1+x)^\beta (1-x^2)^p\right] \tag{2.117a}$$

$$\omega_p^{\alpha,\beta} = \frac{p!\,(\alpha+\beta+2p+1)\,\Gamma(\alpha+\beta+p+1)}{2^{\alpha+\beta+1}\Gamma(\alpha+p+1)\,\Gamma(\beta+p+1)} \tag{2.117b}$$

Special cases of the Jacobi polynomials are the ultraspherical polynomials $C_p^\lambda(x)$ (or Gegenbauer polynomials) obtained with $\alpha = \beta = \lambda - \frac{1}{2}$. Special cases of ultraspherical polynomials include Legendre polynomials $L_p(x)$ with $\alpha = \beta = 0$, Chebyshev polynomials of the first kind $T_p(x)$ with $\alpha = \beta = -\frac{1}{2}$ and Chebyshev polynomials of the second kind $U_p(x)$ with $\alpha = \beta = +\frac{1}{2}$. Like for the periodic case, the Fourier-Jacobi series of a function f is uniformly convergent on $[-1, 1]$ under regularity constraints and the convergence rate that depends on the smoothness of f but also the distribution of samples used to approximates the series coefficients.

As trigonometric interpolation in equispaced points suffers from the Gibbs phenomenon where $\|f - f_N\|_\infty = \mathcal{O}(1)$, polynomial interpolation in equispaced points suffers from the Runge phenomenon where $\|f - f_N\|_\infty = \mathcal{O}\left(2^N\right)$, which is much worse [Grandclement 2006]. A particularly interesting non-uniform distribution of points x_j to limit the Runge phenomenon are the zeros of some Chebyshev and Legendre polynomials, also called Gauss-Lobatto-Chebyshev (GLC) and Gauss-Lobatto-Legendre points (GLL), respectively. Those nodes gives a polynomial interpolant that is not to far from the optimal polynomial of order N, denoted f_N^{opt}:

$$\left\|f - f_N^{opt}\right\|_\infty \leq (1 + \Lambda_N)\,\|f - f_N\|_\infty \tag{2.118}$$

where Λ_N is the Lebesgue constant [Ibrahimoglu 2016] for the considered interpolation nodes:

$$\Lambda_N^{GLC} = \underset{N\to\infty}{\mathcal{O}}(\log N) \qquad \Lambda_N^{GLL} = \underset{N\to\infty}{\mathcal{O}}\left(\sqrt{N}\right) \qquad \Lambda_N^{Uniform} = \underset{N\to\infty}{\mathcal{O}}\left(\frac{2^N}{N\log N}\right)$$

The uniform convergence rate $\|f - f_N\|_\infty$ also depends on the interpolation technique:

1. Expansions in Legendre polynomials are optimal approximants in the L_2-norm:

$$\left\{\begin{array}{l} f \in L^2([-1,1]) \cap C^p([-1,1]) \\ f^{(q)} \in L^2([-1,1]) \quad \forall q \in [\![1,p]\!] \end{array}\right. \Rightarrow \|f - f_N\|_2 = \underset{N\to\infty}{\mathcal{O}}\left(N^{-p-1-\frac{1}{2}}\right) \tag{2.119}$$

2. Expansions in Chebyshev polynomials are optimal approximants in the L_∞-norm:

$$\left\{\begin{array}{l} f \in L^2_h([-1,1]) \cap C^p([-1,1]) \\ f^{(q)} \in L^2_h([-1,1]) \quad \forall q \in [\![1,p]\!] \end{array}\right. \Rightarrow \left\{\begin{array}{l} \|f - f_N\|_{L^2_h} = \underset{N\to\infty}{\mathcal{O}}\left(N^{-p-1-\frac{1}{2}}\right) \\ \|f - f_N\|_\infty = \underset{N\to\infty}{\mathcal{O}}\left(N^{-p-1}\right) \end{array}\right. \tag{2.120}$$

with weights $h(x) = h_{Chebyshev}(x) = \dfrac{1}{\sqrt{1-x^2}}$.

2.7.2 Chebyshev collocation method

The Chebyshev polynomials of the first kind can alternatively be defined by following the recurrence relation:

$$T_0(x) = 1 \tag{2.121a}$$
$$T_1(x) = x \tag{2.121b}$$
$$T_{p+1}(x) = 2xT_p(x) - T_{p-1}(x) \tag{2.121c}$$

T_p is the only polynomial satisfying $T_p(cos(\theta)) = cos(p\theta)$.

The Chebyshev-Gauss-Lobatto points have a better asymptotic behaviour for the Lebesgue constant and have the advantage of being known analytically. The $N+1$ points have analytical formula $x_j = -cos\,(j\pi/N) \in [-1,1]$ for $j \in [\![0,N]\!]$. Moreover, with the knowledge of the $N+1$ samples $\boldsymbol{f} = (f(x_j))_{j\in[\![0,N]\!]}$ it is possible to compute the expansion coefficients $\widehat{\boldsymbol{f}}$ in the basis of orthogonal Chebyshev polynomials of order N efficiently by applying a type one cosine transform (`DCT-I`) with $\mathcal{O}\,(N \log N)$ complexity by using a specialized variant of `FFT` [Neagoe 1990]. In practice, it is better to reverse the indices of the Chebyshev-Gauss-Lobatto points, yielding $x_j = cos\,(j\pi/N)$ to have a direct correspondance with the `DCT-I`. In this case if we define weights $\boldsymbol{w} = (1 + \delta_{p0} + \delta_{pN})_{p\in[\![0,N]\!]}$ we have $\widehat{\boldsymbol{f}} = \mathcal{C}_{\mathtt{I}}(\boldsymbol{f}) \oslash \boldsymbol{w}$, the first and the last coefficients being scaled by a factor one-half. As the inverse of the `DCT-I` is the `DCT-I` itself, $\boldsymbol{f}$ can also be computed from $\widehat{\boldsymbol{f}}$ with the same complexity with $\boldsymbol{f} = \mathcal{C}_{\mathtt{I}}(\widehat{\boldsymbol{f}} \odot \boldsymbol{w})$. The result of these procedures has eventually to be scaled by some factor depending of N according to the chosen implementation. This fast algorithm only works when f has no sharp discontinuity on $[-1,1]$ because the `DCT-I` coefficients are subject to the same convergence results as the DFT (without the periodicity constraint) and thus are subject to the Gibbs phenomenon. The same error result in $\mathcal{O}\left(\varepsilon\sqrt{N}\right)$ holds when using a suitable algorithm (`FFTW` has a correct implementation unlike `FFTPACK`, see [Frigo et al. 2005] part B). Another way to compute the coefficients is to compute the $N+1$ scalar products $\langle f, \phi_j\rangle_h$ directly by using the discrete variant of formula (2.115b).

The convergence of the interpolation at the Chebyshev-Gauss-Lobatto points is illustrated on figure 2.17 for three different functions defined on $[-1, 1]$ with varying regularity. As for the Fourier transform in the periodic case, a limit case can be provided by a discontinuous function where convergence can not be guaranteed as formulated in (2.120).

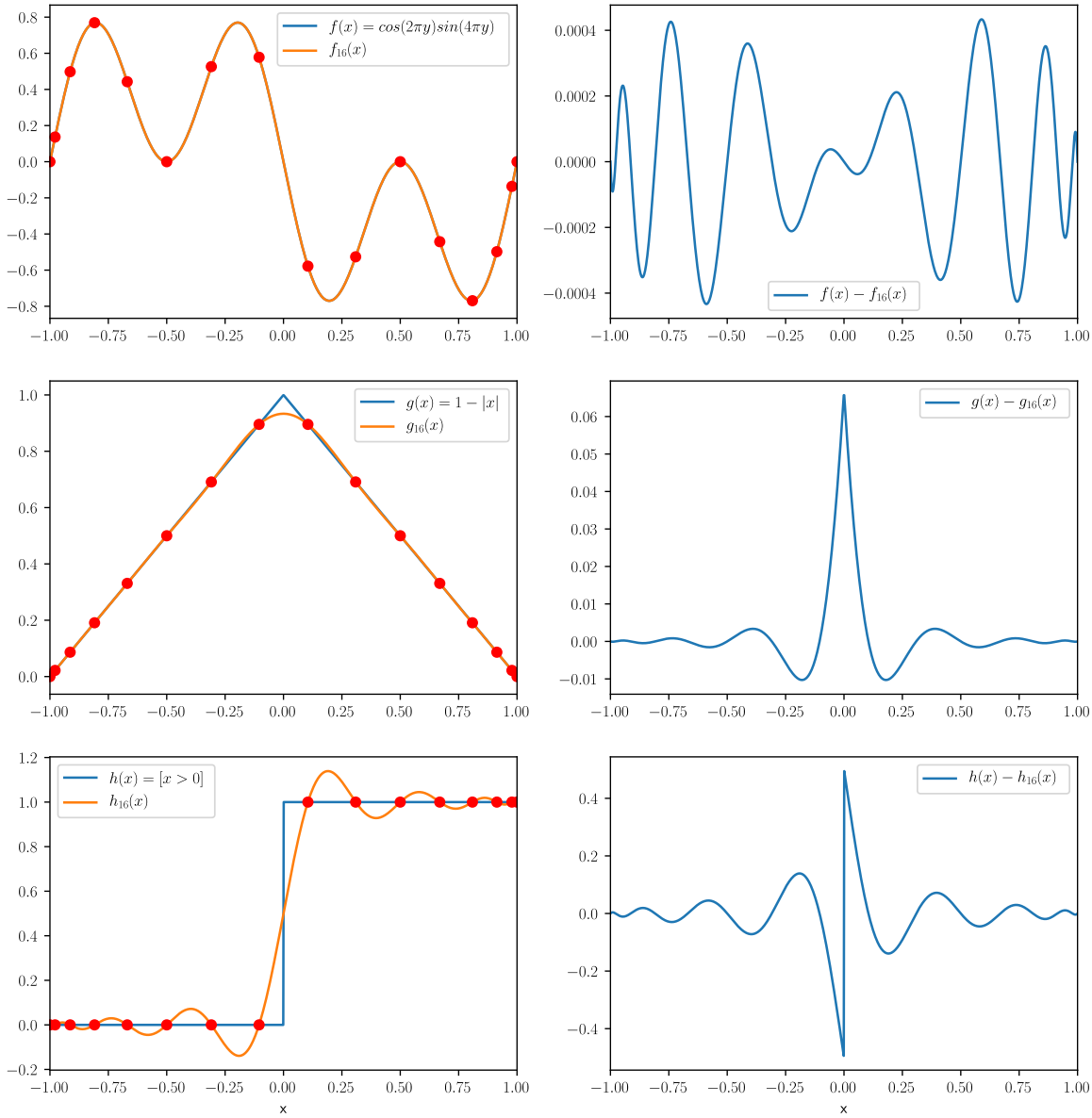


Figure 2.17 – Spectral Chebyshev interpolation of functions of varying regularity. The three functions considered here are all square-integrable functions with respect to weight function $h(x) = 1/\sqrt{1-x^2}$. For all $x \in [-1, 1]$ we define f, g and h similarly to the DFT interpolation presented on figure 2.13 as $f(x) = cos(2\pi y)sin(4\pi y) \in C^{\omega}(\mathbb{R})$ with $y = (x+1)/2$, $g(x) = 1 - |x| \in C^0(\mathbb{R})$ and $h(x) = \big[x > 0\big] \in C^0_I(\mathbb{R})$. Those functions have decreasing regularity and theoretical convergence rate is given by (2.120). Each leftmost plot represent the function $f(x)$ to be interpolated (blue) and its spectral Chebyshev interpolant $f_{16}(x)$ (orange) obtained by evaluating (2.71) with $N + 1 = 16$ samples. The 16 samples $f_j = f(x_j)$ with $x_j = cos(j\pi/15)) \in [-1, 1]$ used to compute the interpolation are shown as red points.

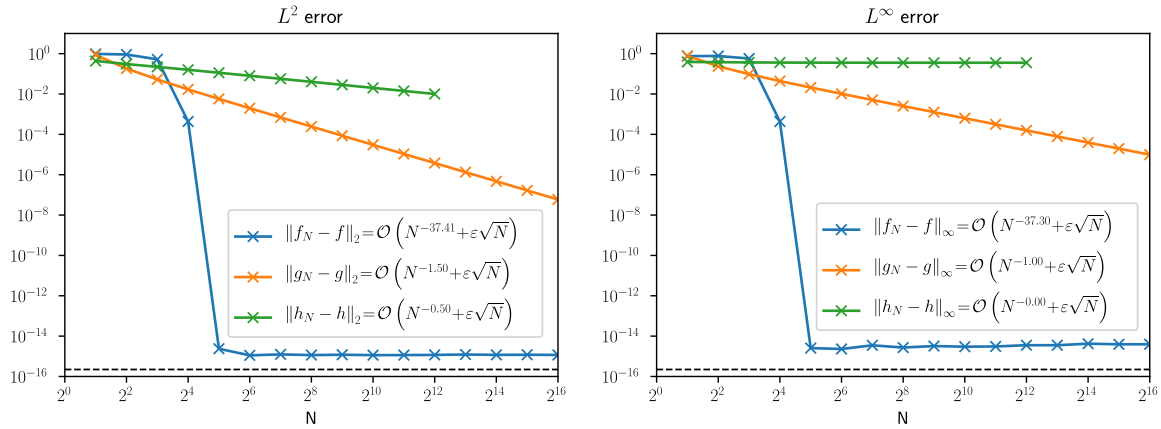


Figure 2.18 – Convergence of the GLC Chebyshev interpolation method.
See figure 2.17 for the definition of functions f, g and h. Here the coefficients $\widehat{\boldsymbol{f}}$ and $\widehat{\boldsymbol{g}}$ are computed efficiently by using the `DCT-I` while coefficients $\widehat{\boldsymbol{h}}$ are computed up to $N = 2^{12}$ by scalar products because of the discontinuity of the function $h(x)$. The black dotted line represents the machine error for double precision floating point numbers $y = \varepsilon \simeq 2.22 \times 10^{-16}$.

Given N+1 values of at the Gauss-Lobatto-Chebyshev points one can also approximate $f(x)$ by using the $N+1$ interpolating polynomials $\mathcal{L}_{N,j}(x)$ associated to the Chebyshev polynomial of degree N:

$$\mathcal{L}_{N,j}(x) = \frac{(-1)^{j+1}(1-x^2)T'_N(x)}{(1+\delta_{j0}+\delta_{jN})N^2(x-x_j)} \quad \forall\, j \in [\![0, N]\!] \tag{2.122a}$$

$$f(x) \simeq \sum_{j=0}^{N} \mathcal{L}_{N,j}(x)\, f(x_j) = \sum_{j=0}^{N} \mathcal{L}_{N,j}(x)\; f_j \tag{2.122b}$$

$$\mathcal{L}_{N,i}(x_j) = \delta_{ij} \quad \forall (i,j) \in [\![0, N]\!]^2 \tag{2.122c}$$

This leads to a first way to compute derivatives at the Chebyshev extreme points:

$$f^{(q)}(x) \simeq \sum_{j=0}^{N} \mathcal{L}^{(q)}_{N,j}(x)\; f_j \tag{2.123a}$$

$$\boldsymbol{f}' = D_N^q\, \boldsymbol{f} \tag{2.123b}$$

Multiplication by the first order Chebyshev differentiation matrix $D_N = \left(\dfrac{\partial \mathcal{L}_{N,j}}{\partial x}(x_i)\right)_{ij}$ transforms a vector data at the Gauss-Lobatto-Chebyshev points into approximative derivative on those same points. D_N is a matrix of size $(N+1) \times (N+1)$ and is explicitly defined by:

$$(D_N)_{00} = -(D_N)_{NN} = \frac{2N^2+1}{6}$$

$$(D_N)_{jj} = \frac{-x_j}{2(1-x_j^2)} \quad \forall\, j \in [\![1, N-1]\!]$$

$$(D_N)_{ij} = \frac{1+\delta_{i0}+\delta_{iN}}{1+\delta_{j0}+\delta_{jN}}\, \frac{(-1)^{i+j}}{x_i - x_j} \quad \forall\, i \neq j$$

The Chebyshev differentiation matrix is nilpotent of degree $N+1$ and has the interesting property of being skew-centrosymmetric implying that D_N^{2q} is centrosymmetric and D_N^{2q+1} is skew-centrosymmetric for all positive integers q. In practice we loose the symmetry properties when taking into account certain boundary conditions because D_N is modified to take into account the boundary conditions at $x = x_0 = -1$ and $x = x_N = 1$. Suppose we want to solve the following second order boundary value partial differential equation:

$$\sum_{i=0}^{N} a_i(x)\frac{\partial^i \phi}{\partial x^i}(x) = f(x) \text{ on }]-1,+1[\tag{2.124a}$$

$$\alpha_l\ \phi(-1) + \beta_l \frac{\partial \phi}{\partial x}(-1) = \gamma_l \tag{2.124b}$$

$$\alpha_r\ \phi(+1) + \beta_r \frac{\partial \phi}{\partial x}(+1) = \gamma_r \tag{2.124c}$$

$$(\alpha_l, \alpha_r) \neq (0,0) \tag{2.124d}$$

Let $x_0, \cdots, x_N$ be the N+1 Chebyshev-Gauss-Lobatto collocation points in [-1,+1]. We can define $A_i = \text{diag}(a_i(x_0), \cdots, a_i(x_N))$ for all $i \in [\![0, N]\!]$ and:

$$\boldsymbol{\phi} = \begin{pmatrix} \phi(x_0) \\ \phi(x_1) \\ \phi(x_2) \\ \vdots \\ \phi(x_{N-1}) \\ \phi(x_N) \end{pmatrix} \quad \boldsymbol{f} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \\ f(x_N) \end{pmatrix} \quad \overline{\boldsymbol{f}} = \begin{pmatrix} \gamma_r \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \\ \gamma_l \end{pmatrix} \tag{2.125}$$

$$L_N = \left[\sum_{i=0}^{N} A_i (D_N)^i\right] \quad \text{and} \quad \overline{L_N} = \begin{pmatrix} \alpha_r (I_{N+1})_{0j} + \beta_r (D_N)_{0j} \\ \hline (L_N)_{ij} \\ \hline \alpha_l (I_{N+1})_{Nj} + \beta_l (D_N)_{Nj} \end{pmatrix} \tag{2.126}$$

The solution to the linear system $\overline{L_N}\,\boldsymbol{\phi} = \overline{\boldsymbol{f}}$ satisfies the two boundary conditions (2.124b) at $x = x_0 = -1$ and (2.124c) at $x = x_N = +1$ while equation (2.124a) is satisfied at all the $N-1$ inner collocation points $x_1, \cdots, x_{N-1}$.

2.7.3 Chebyshev-Tau method

An other way to solve this system is to use the representation of $\phi(x)$ in the basis of Chebyshev polynomials. The derivatives can then be obtained in the spectral basis similarly to the DFT

via the multiplication of the coefficient by an upper triangular first derivative matrix $\widehat{D_N}$.

$$f_N(x) = \sum_{p=0}^{N} \widehat{f_p} T_p(x) \tag{2.127a}$$

$$f_N^{(q)}(x) = \sum_{p=0}^{N} \widehat{f_p} T_p^{(q)}(x) = \sum_{p=0}^{N} \widehat{f'_p} T_p(x) \tag{2.127b}$$

where $\widehat{\boldsymbol{f}'}$ can be obtained from $\widehat{\boldsymbol{f}}$ with the knowledge of the of the matrix $\widehat{D_N} = \left(\langle \partial_x T_i, T_j \rangle_h\right)_{ij}$ that transforms the spectral coefficients of the interpolant to the spectral coefficients of the derivative in the orthogonal Chebyshev basis:

$$\widehat{\boldsymbol{f}'} = (\widehat{D_N})^q \widehat{\boldsymbol{f}} \tag{2.128a}$$

$$(\widehat{D_N})_{ij} = \mathbb{1}_{2\mathbb{Z}+1}(i+j) \frac{j}{2-\delta_{i0}} \quad \forall (i,j) \in [\![0, N]\!]^2 \tag{2.128b}$$

The procedure to solve the problem is then same then for the Fourier transform:

- All spatial derivatives are computed in the spectral space with equation (2.128).
- All linear operations (additions, subtractions, multiplication by a scalar) can be equally made in the spectral or in the physical space:

$$(c_0\phi + c_1\psi)_N(x) = c_0\phi_N(x) + c_1\psi_N(x)$$
$$\mathcal{F}(c_0\boldsymbol{\phi} + c_1\boldsymbol{\psi}) = c_0\widehat{\boldsymbol{\phi}} + c_1\widehat{\boldsymbol{\psi}}$$

- All non-linear products are made in the real space, and the result is transfered back to the spectral space:

$$(\phi\,\psi)_N(x) = \phi_N(x)\,\psi_N(x)$$
$$\mathcal{F}(\boldsymbol{\phi} \odot \boldsymbol{\psi}) = \mathcal{F}\left(\mathcal{F}^{-1}\left(\hat{\boldsymbol{\phi}}\right) \odot \mathcal{F}^{-1}\left(\hat{\boldsymbol{\psi}}\right)\right)$$

  where special care may be taken to dealias the product [Uhlmann 2000]

If we suppose that all the a_i do not depend on x with $a_i(x) = c_i$ in equation (2.124a) then we obtain the system $\widehat{L_N}\,\widehat{\boldsymbol{\phi}} = \widehat{\boldsymbol{f}}$ with $\widehat{L_N} = \sum_{i=0}^{N} c_i \left(\widehat{D_N}\right)^i$. The left (2.124b) and right (2.124c) boundary conditions are then included in the two last rows of $\widehat{L_N}$. The value of the Chebyshev polynomials and their first derivative at left and right boundaries can be computed with the following formulas:

$$T_p^{(0)}(-1) = (-1)^p \qquad T_p^{(0)}(1) = 1 \tag{2.129a}$$

$$T_p^{(1)}(-1) = (-1)^{p+1}\,p^2 \qquad T_p^{(1)}(1) = p^2 \tag{2.129b}$$

Thus by defining $T_l = (T_p(-1))_p$, $T_l' = \left(T_p^{(1)}(-1)\right)_p$, $T_r = (T_p(1))_p$ and $T_r' = \left(T_p^{(1)}(1)\right)_p$ for $p \in [\![0, N]\!]$ we can include the boundary conditions in $\widehat{L}$ and $\widehat{f}$:

$$\overline{L_N} = \begin{pmatrix} (\widehat{L_N})_{ij} \\ \hline \alpha_l T_l + \beta_l T_l' \\ \alpha_r T_r + \beta_r T_r' \end{pmatrix} \quad \text{and} \quad \overline{\boldsymbol{f}} = \begin{pmatrix} \widehat{f}(x_0) \\ \widehat{f}(x_1) \\ \vdots \\ \widehat{f}(x_{N-2}) \\ \gamma_l \\ \gamma_r \end{pmatrix} \tag{2.130}$$

The solution to the linear system $\overline{L_N}\,\widehat{\boldsymbol{\phi}} = \overline{\boldsymbol{f}}$ satisfies the system including the two boundary conditions. Once the system has been solved, it is possible to recover the solution $\boldsymbol{\phi}$ at the Chebyshev extreme points by performing an inverse Chebyshev transform on $\widehat{\boldsymbol{\phi}}$, which is nothing more than a scaled inverse type one cosine transform. Similarly $\widehat{\boldsymbol{f}}$ is obtained from $\boldsymbol{f}$ by using a forward Chebyshev transform.

2.7.4 Chebyshev-Galerkin method

An alternative method to solve second order boundary value problems with Fourier-Chebyshev series is to take into account the boundary conditions directly in the basis, similarly to the real-to-real transforms of previous section. In the context of spectral methods this is known as the Galerkin method.

For the Chebyshev polynomials it is known that for homogeneous boundary conditions of the form (2.124b)-(2.124c) with $\gamma_l = \gamma_r = 0$, there exist a unique set of constants $(a_p, b_p) \in \mathbb{R}^{N+1} \times \mathbb{R}^{N+1}$ such that:

$$G_p(x) = T_p(x) + a_p\, T_{p+1}(x) + b_p\, T_{p+2}(x) \tag{2.131}$$

satisties the boundary conditions for given $(\alpha_l, \alpha_r, \beta_l, \beta_r)$. The resulting basis is not orthogonal but spans the solution space [Shen et al. 2011].

Here we propose to extend this result to non-homogeneous boundary conditions. The boundary values being given by equations (2.129), we can derive a system to solve for the coefficients that satify any boundary condition (with $\gamma_l \neq 0$ and $\gamma_r \neq 0$):

$$G_p^{(0)}(-1) = (-1)^p + a_p\,(-1)^{p+1} + b_p\,(-1)^{p+2} \tag{2.132a}$$
$$G_p^{(0)}(+1) = 1 + a_p + b_p \tag{2.132b}$$
$$G_p^{(1)}(-1) = (-1)^p\, p^2 + a_p\,(-1)^{p+1}\,(p+1)^2 + b_p\,(-1)^{p+2}\,(p+2)^2 \tag{2.132c}$$
$$G_p^{(1)}(+1) = p^2 + a_p\,(p+1)^2 + b_p\,(p+2)^2 \tag{2.132d}$$

the system being given by:

$$\alpha_l G_p^{(0)}(-1) + \beta_l G_p^{(1)}(-1) = \gamma_l \tag{2.133a}$$
$$\alpha_r G_p^{(0)}(+1) + \beta_r G_p^{(1)}(+1) = \gamma_r \tag{2.133b}$$

The general solution to this system can be found by any symbolic computation software and we obtain the following:

$$
\begin{aligned}
a_p &= \frac{A_p}{B_p} \quad \text{and} \quad b_p = \frac{C_p}{D_p} \\
A_p &= (-1)^{1-p}\left(\alpha_r\,\gamma_l + 4\,\beta_r\,\gamma_l(p+1) + \beta_r\,\gamma_l\,p^2\right) \\
&\qquad + \left(\alpha_l\,\gamma_r + 4(\alpha_l\,\beta_r - \alpha_r\,\beta_l + \beta_l\,\gamma_r)(p+1) + \beta_l\,\gamma_r\,p^2\right) \\
B_p &= (2\,\alpha_l\,\alpha_r + 5\,\alpha_l\,\beta_r + 5\,\alpha_r\,\beta_l + 8\,\beta_l\,\beta_r) + (6\,\alpha_l\,\beta_r + 6\,\alpha_r\,\beta_l + 24\,\beta_l\,\beta_r)p \\
&\qquad + (2\,\alpha_l\,\beta_r + 2\,\alpha_r\,\beta_l + 26\,\beta_l\,\beta_r)p^2 + (12\,\beta_l\,\beta_r)p^3 + (2\,\beta_l\,\beta_r)p^4 \\
C_p &= \left(-\alpha_l - \beta_l p^2 + (-1)^{-p}\,\gamma_l\right)\left(\alpha_r + \beta_r p^2 + 2\beta_r p + \beta_r\right) \\
&\qquad + \left(-\alpha_r - \beta_r p^2 + \gamma_r\right)\left(\alpha_l + \beta_l p^2 + 2\beta_l p + \beta_l\right) \\
D_p &= \left(\alpha_l + \beta_l p^2 + 2\beta_l p + \beta_l\right)\left(\alpha_r + \beta_r p^2 + 4\beta_r p + 4\beta_r\right) \\
&\qquad + \left(\alpha_l + \beta_l p^2 + 4\beta_l p + 4\beta_l\right)\left(\alpha_r + \beta_r p^2 + 2\beta_r p + \beta_r\right)
\end{aligned}
$$

The expression of a_p and b_p is greatly simplified for the common cases of Dirichlet-Dirichlet ($\alpha_l = \alpha_r = 1$), Dirichlet-Neumann ($\alpha_l = \beta_r = 1$), Neumann-Dirichlet ($\beta_l = \alpha_r = 1$) and Neumann-Neumann ($\beta_l = \beta_r = 1$) boundary conditions:

$(\boldsymbol{\alpha_l}, \boldsymbol{\beta_l}, \boldsymbol{\alpha_r}, \boldsymbol{\beta_r})$	$\boldsymbol{a_p}$	$\boldsymbol{b_p}$
$(1,0,1,0)$	$\frac{\gamma_r}{2} - \frac{(-1)^{-p}\,\gamma_l}{2}$	$\frac{\gamma_r}{2} + \frac{(-1)^{-p}\,\gamma_l}{2}$ - 1
$(1,0,0,1)$	$\frac{(\gamma_r + 4(p+1)) + (-1)^{1-p}\,(\gamma_l p^2 + 4\gamma_l p + 4\gamma_l)}{2p^2 + 6p + 5}$	$\frac{(\gamma_r - p^2) + \left((-1)^{-p}\,\gamma_l - 1\right)(p^2 + 2p + 1)}{2p^2 + 6p + 5}$
$(0,1,1,0)$	$\frac{\gamma_r p^2 + 4(\gamma_r - 1)(p+1) + (-1)^{1+p}\,\gamma_l}{2p^2 + 6p + 5}$	$\frac{((-1)^p\gamma_l - p^2) + (\gamma_r - 1)\,(p^2 + 2p + 1)}{2p^2 + 6p + 5}$
$(0,1,0,1)$	$\frac{\gamma_r + (-1)^{1+p}\,\gamma_l}{2\,(p^2 + 2p + 1)}$	$\frac{\gamma_r - 2p^2 + (-1)^p\gamma_l}{2\,(p^2 + 4p + 4)}$

Table 2.3 – Chebyshev-Galerkin coefficients (2.131) for usual boundary conditions

Once the coeffients a_p and b_p are known for given left and right boundary conditions, it is possible to express any function f in terms of Chebyshev and Chebyshev-Galerkin Fourier series:

$$
\begin{aligned}
f_N(x) &= \sum_{p=0}^{N} \widehat{f}_p T_p(x) \\
&= \sum_{p=0}^{N-2} \widetilde{f}_p G_p(x) \\
\text{with } G_p(x) &= T_p(x) + a_p\,T_{p+1}(x) + b_p\,T_{p+2}(x)
\end{aligned}
\tag{2.135}
$$

By identifying terms in the Chebyshev orthogonal basis, it is possible to express the $N+1$ Chebyshev coefficients $\widehat{f}$ in terms of the $N-1$ Chebyshev-Galerkin coefficients $\widetilde{f}$:

$$\widehat{f}_0 = \widetilde{f}_0 \tag{2.136a}$$
$$\widehat{f}_1 = \widetilde{f}_1 + a_0\widetilde{f}_0 \tag{2.136b}$$
$$\widehat{f}_p = \widetilde{f}_p + a_{p-1}\widetilde{f}_{p-1} + b_{p-2}\widetilde{f}_{p-2} \quad \forall p \in [\![2, N-2]\!] \tag{2.136c}$$
$$\widehat{f}_{N-1} = a_{N-2}\widetilde{f}_{N-2} + b_{N-3}\widetilde{f}_{N-3} \tag{2.136d}$$
$$\widehat{f}_N = b_{N-2}\widetilde{f}_{N-2} \tag{2.136e}$$

This gives an $\mathcal{O}(N)$ procedure to go from the Chebyshev-Galerkin coefficients to the Chebyshev coefficients and we denote R_c the associated matrix of size $(N+1, N-1)$ such that $\widehat{f} = R_c\,\widetilde{f}$. Because R_c is not a square matrix, we can not compute its inverse, fortunately there exist two different iterative approaches with the same complexity to compute $\widetilde{f}$ from $\widehat{f}$:

Forward algorithm:

$$\begin{aligned}\widetilde{f}_0 &= \widehat{f}_0\\ \widetilde{f}_1 &= \widehat{f}_1 - a_0\widetilde{f}_0\\ \widetilde{f}_p &= \widehat{f}_p - a_{p-1}\widetilde{f}_{p-1} - b_{p-2}\widetilde{f}_{p-2}\end{aligned}$$

Backward algorithm (defined for $b_p \neq 0$):

$$\begin{aligned}\widetilde{f}_{N-2} &= \widehat{f}_N/b_{N-2}\\ \widetilde{f}_{N-3} &= (\widehat{f}_{N-1} - a_{N-2}\widetilde{f}_{N-2})/b_{N-3}\\ \widetilde{f}_{N-2-p} &= (\widehat{f}_{N-p} - \widetilde{f}_{N-p} - a_{N-p-1}\widetilde{f}_{N-p-1})/b_{N-p}\end{aligned}$$

We denote R_g the matrix of size $(N-1, N+1)$ associated to one of those algorithms. With this definition we have $\widetilde{f} = R_g\,\widehat{f}$ under the assumption that the $N+1$ coefficients $\widehat{f}$ represent a function with the correct boundary conditions (2.124b) and (2.124c). With these matrices with have $R_g\,R_c = I_{N-2}$ but $R_c\,R_g \neq I_N$. The Chebyshev-Galerkin derivative matrix $\widetilde{D_N}$ of shape $(N-1, N-1)$ is obtained from the Chebyshev derivation matrix $\widehat{D_N}$ of shape $(N+1, N+1)$ by computing $\widetilde{D_N} = R_g\widehat{D_N}R_c$.

The procedure to find the solution to problem (2.124) with constant coefficients at the Chebyshev-Gauss-Lobatto points is then the following:

1. **Preprocessing:**
   (a) Compute the $N-1$ coefficients a_p and b_p corresponding to the left and right boundary condition by using equation (2.134).
   (b) Compute $\widehat{D_N}$ and $\widehat{L_N} = \sum_{i=0}^{N} c_i \left(\widehat{D_N}\right)^i$, $\widehat{L_N}$ is a square matrix of size $N+1$.
   (c) Compute R_c, R_g and $\widetilde{L_N} = R_g\widehat{L_N}R_c$, $\widetilde{L_N}$ is a square matrix of size $N-1$.
2. Compute the coefficients of $\widehat{\boldsymbol{f}}$ from $\boldsymbol{f}$ in $\mathcal{O}(N\log N)$ by using a scaled `DCT-I` transform.
3. Compute $\widetilde{\boldsymbol{f}} = R_g\widehat{\boldsymbol{f}}$ in $\mathcal{O}(N)$ operations.
4. Solve $\widetilde{L_N}\widetilde{\boldsymbol{\phi}} = \widetilde{\boldsymbol{f}}$ to find the coefficients of ϕ in the Chebyshev-Galerkin basis.
5. Compute $\widehat{\boldsymbol{\phi}} = R_c\widetilde{\boldsymbol{\phi}}$ in $\mathcal{O}(N)$ operations.
6. Compute the $\boldsymbol{\phi}$ from $\widehat{\boldsymbol{\phi}}$ in $\mathcal{O}(N\log N)$ by using a scaled `DCT-I` transform.

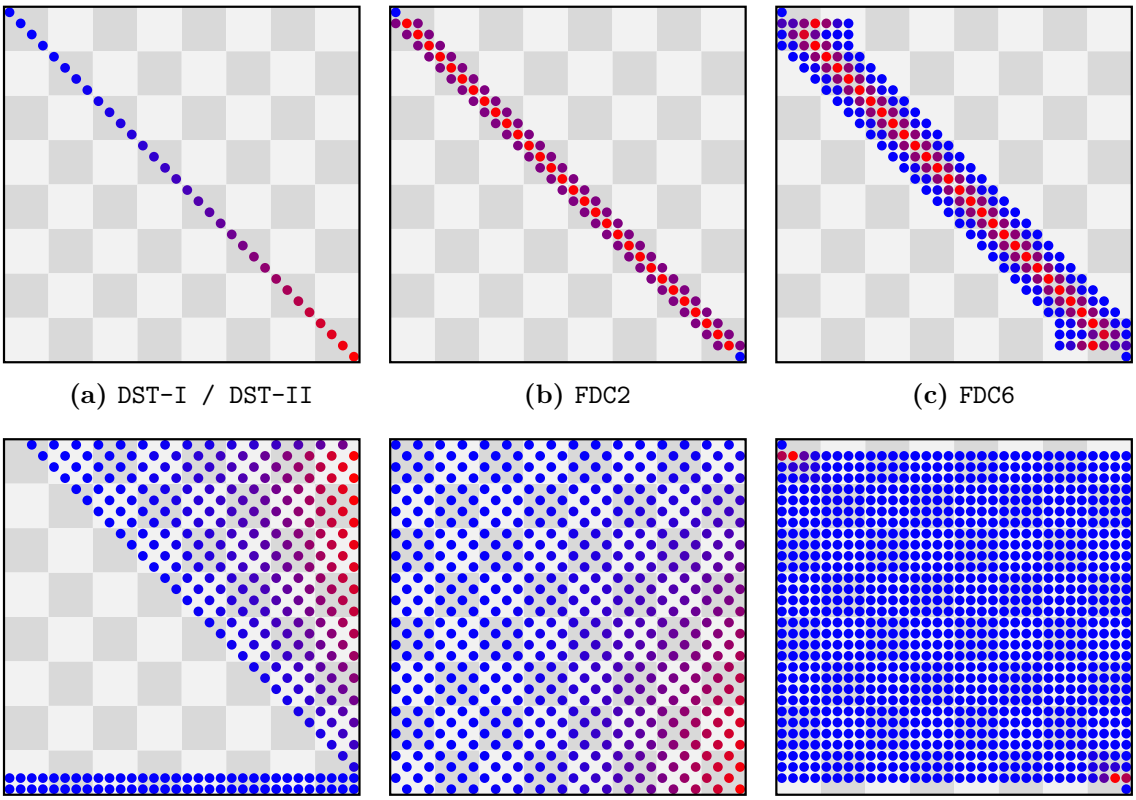
2.7.5 Comparison of the different methods

In this section we check the convergence of the different methods with respect to the following one-dimensional Poisson problem:

$$\frac{\partial \phi^2}{\partial x^2}(x) = f(x) = e^{4x} \text{ on } \Omega =]-1, 1[\tag{2.139a}$$

$$\phi(-1) = \phi(1) = 0 \tag{2.139b}$$

The analytical solution is $\phi(x) = (f(x) - xsinh(4) - cosh(4))/16$. As the problem features homogeneous Dirichlet boundary conditions we can use the `DST-I` and `DST-II` transforms with associated collocated or staggered grid. With those transforms we cannot expect better than second order convergence because of the Gibbs phenomenon: $f(1) \simeq 54.6 \gg 0$. We compare the results obtained by implicit centered finite differences and various spectral methods with the analytical solution. The associated linear systems are shown on figure 2.19 for a discretization of $N = 32$ nodes.



(a) `DST-I / DST-II` (b) `FDC2` (c) `FDC6`

(d) `Chebyshev-Tau` (e) `Chebyshev-Galerkin` (f) `Chebyshev collocation`

Figure 2.19 – Matrix layouts associated to the different methods ($N = 32$).
Each dot corresponds to a non-zero entry in the matrix. Coefficients are color graded from lowest (blue) to highest absolute value (red). `FDCx` corresponds to implicit centered finite differences of given spatial order (decentered on the boundaries when required).

For the discrete sine transforms, there is no linear system to solve because the forward transform performs the diagonalization of the problem. Implicit centered finite-differences schemes lead to banded Toeplitz matrices of increasing bandwidth. The Chebyshev collocation method requires the resolution of a dense matrix while the Chebyshev-Galerkin method yields a checkerboard matrix. Finally, the Chebyshev-Tau method gives an upper triangular checkerboard matrix. All those linear systems are solved without using any preconditioner. As it can be seen on figure 2.20, we recover the expected convergence rates for all methods.

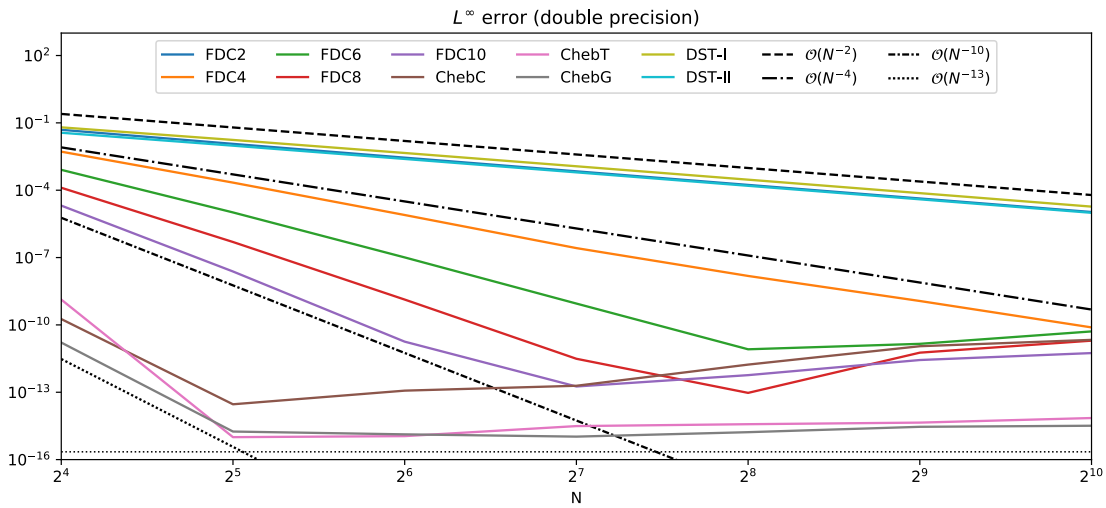


Figure 2.20 – Convergence of the different methods for a one-dimensional homogeneous Dirichlet Poisson problem. `FDCx` corresponds to implicit centered finite differences of given order, `ChebC` to the collocated Chebyshev method, `ChebT` to the Chebyshev-Tau method, `ChebG` to the Chebyshev-Galerkin method and `DST-x` to the discrete sine transforms.

Conclusion

In this chapter we built an hardware agnostic numerical method that allows us to solve the incompressible Navier-Stokes equations passively coupled with the transport of a scalar, possibly at high Schmidt number. The main idea behind the proposed numerical method is the use of a remeshed particle method. While this method is particularly adapted for the transport of scalars in high Schmidt number flows, the presence of an underlying grid allows to the use of efficient eulerian solvers. The resulting method consists in an hybrid particle, spectral and finite differences based numerical method especially designed to reduce the timestep constraints associated to high Schmidt number flows. Those numerical methods are further adapted in order to become accelerator-friendly.

Within this framework, the governing equations are discretized on a rectangular computational box and the domain is meshed by using regular grids. Boundary conditions are prescribed on the walls. The numerical method is not limited to periodic boundary conditions and can take into account other kind of boundary conditions such as homogeneous Dirichlet, homogeneous Neumann or even more general boundary conditions. The numerical method

is based on two different kinds of operator splitting, namely viscous and directional splitting. Viscous splitting decomposes the problem into subproblems that each can be solved with a specific set of numerical methods. All of those subproblems but the recovery of velocity from vorticity are further split directionally by using a Strang splitting of first or second order, yielding one-dimensional subproblems. This makes the corresponding numerical scheme dimension-agnostic and thus simpler to implement. It is predicted that the choice of a Cartesian discretization along with a simple data permutation strategy in between the treatment of different directions allows efficient implementations on vectorized hardware such as `CPU`s and `GPU`s.

Transport is solved at fixed velocity, one direction at a time, by using a high order remeshed particle method that does not suffer from a timestep restriction depending on the grid size (CFL) but rather on velocity gradients (LCFL). Stretching and external forcing are handled with directionally split explicit finite differences. Diffusion can be treated either by using explicit finite differences or with a spectral solver. When the overall timestep of the simulation is limited by some timestep associated to a diffusion subproblem, it may be better to choose an implicit spectral method that does not enforce any timestep restriction. Finally, the recovery of velocity from the vorticity is always done through a spectral solver. All of those operators are solved one after another within a simulation iteration with variable timestepping. Explicit finite differences and transport each enforce a minimum timestep for a given iteration. A wide variety of spectral methods has been presented to handle any second order boundary value problems. Periodicity is handled with the Fourier transform while homogeneous boundary conditions can be handled by sine and cosine transforms under certain assumptions. All the other cases are handled by Chebyshev polynomials, at the cost of increased computational complexity. The recovery of velocity from vorticity and diffusion can be solved by using a tensor product of spectral bases statisfying the right boundary conditions. With this spectral method, there is no linear system to solve unless Chebyshev polynomials are used.

In the next chapter we will see how to implement efficiently those numerical methods depending on the target hardware. Results obtained with this numerical framework as well as performance statistics will also be given. The split operators presented in this chapter constitute the basic bricks that will be used to solve every other incompressible Navier-Stokes problems.

Chapter 3

Implementation and High Performance Computing

Contents

3.1 The HySoP library . . . 133

3.1.1 Origin and current state of the library . . . 133

3.1.2 Design of the library . . . 137

3.1.3 `HySoP` from a user point of view . . . 143

3.1.4 Graph of operators and task parallelism . . . 146

3.1.5 Integration of numerical methods . . . 148

3.2 Implementation on hardware accelerators . . . 149

3.2.1 Arrays, buffers and memory handling . . . 150

3.2.2 `OpenCL` code generation framework . . . 152

3.2.3 Permutations and automatic tuning of kernel runtime parameters . . . 159

3.2.4 Directional advection operator . . . 164

3.2.5 Finite differences operators . . . 172

3.2.6 Spectral transforms . . . 175

3.2.7 Navier-Stokes runtime distribution . . . 178

3.2.8 Validation of the solver . . . 180

3.3 Distributed computing . . . 186

3.3.1 Domain decomposition . . . 186

3.3.2 Interprocess communications . . . 187

3.3.3 Field discretization . . . 192

3.3.4 Spectral solvers . . . 192

Introduction

One of the main objective of this thesis consists in the efficient implementation of the numerical schemes presented in the previous chapter. The resulting implementation should be able to operate on accelerators, possibly in a distributed environment. The numerical code should be easy to adapt to new architectures while retaining the possibility of operating on conventional machines for the sake of sustainability.

In a first part, we give an overall presentation of the `HySoP` library, the general high-performance computing framework in which we will combine multiple levels of parallelism. Within this framework, the description of a numerical algorithm is handled by building a graph of operators. This first global view on the library is the opportunity to focus on task parallelism.

In a second part, we focus on the specific care required for the implementation on hardware accelerators. Numerical methods are implemented once for multiple different architectures by relying on the `OpenCL` standard. We show that the regularity of the data structures combined to data permutations leads to coalesced memory accesses that are well suited for vectorized hardware. Performance-portability can achieved by the way of code generation techniques associated to automatic runtime kernel performance tuning. Benchmark results are given and analyzed for four reference configurations ranging from simple consumer grade desktop `CPU`s to professional grade `GPU`s. In particular we focus on the individual `OpenCL` implementation of each algorithm as well as runtimes statistics obtained for full two- and three-dimensional incompressible Navier-Stokes simulations. The numerical scheme is then validated on the Taylor-Green Vortex problem.

The last part is dedicated to the distributed implementation of the algorithms, domain decomposition and the specific challenges raised by distributed computing in the context of spectral transforms.

3.1 The HySoP library

HySoP is a library dedicated to high performance direct numerical simulation of fluid related problems based on semi-lagrangian particle methods, for hybrid architectures providing multiple compute devices including CPUs, GPUs or MICs. The high level functionalities and the user interface are mainly written in Python using the object oriented programming model. This choice was made because of the large software integration benefits it can provide [Sanner 1999]. Moreover, the object oriented programming model offers a flexible framework to implement scientific libraries when compared to the imperative programming model [Arge et al. 1997][Cary et al. 1997]. It is also a good choice for the users as the Python language is easy to use for beginners while experienced programmers can pick it up very quickly [Oliphant 2007a]. The numerical solvers are mostly implemented using compiled languages such as Fortran, C/C++, or OpenCL for obvious performance reasons. It is also possible to implement numerical algorithms using directly Python, which is an interpreted language, hence slower for critical code paths under heavy arithmetic or memory load. The Python language support is however the key for rapid development cycles of experimental features. It also allows to easily implement routines that compute simulation statistics during runtime, relieving most of the user post-processing efforts and enabling live simulation monitoring.

The compiled Fortran and C++ compute backends allow us to integrate a variety of external dependencies by connecting them to the main HySoP python module with interface wrappers such as F2PY or SWIG. Note that many scientific libraries already provide Python interfaces so that they can be directly used in python without needing the user to implement his own wrapper. In addition to the compiled languages, the library offers the possibility to compile generated code just-in-time during execution. This is the case for OpenCL, the language used to drive OpenCL-compatible accelerators, like GPUs, but also to translate python methods to fast machine code by using the Numba just-in-time compiler. Most of the dependencies that the HySoP library uses are described in appendix B.

This subsection provides implementation details of the HySoP library from its origin to the most recent developments (2012-2019). It aims to cover computational-backend agnostic design choices that allows us to discretize the variables and equations of a given fluid-related problem into subproblems, each being solved individually by some operator embedded into a directed acyclic graph. A first level of parallelism is explored here, namely task parallelism.

3.1.1 Origin and current state of the library

The name of the library, **Hy**brid **S**imulati**o**n with **P**articles (HySoP), comes from its original purpose to propose a high performance framework for flow simulations based on semi-lagrangian particle methods. This library got created under the impulsion of [Etancelin 2014]. To this day the scope of the library is a bit larger and the library has more or less become a framework to develop high performance computational fluid-related routines operating on fields discretized on regular meshes. The library still uses the original Fortran implementa-

tion [Lagaert et al. 2012] of the remeshed particle proposed by [Cottet et al. 2009b] and further developed by [Magni et al. 2012]. The original `Fortran` code, named `SCALES`, was capable of running on massively parallel `CPU` architectures by using the Message Passing Interface (`MPI`). At this time the advection remeshing routines were coupled with a fully pseudo-spectral Navier-Stokes solver.

Shortly after, `HySoP` was born, by wrapping a subset of the `SCALES` library, mainly the particle remeshing functions, using `F2PY` and by implementing a custom solver for the fluid simulation, arising from successive developments of the underlying numerical schemes [Balarac et al. 2014] [Lagaert et al. 2014]. Current developments of the library builds on the original implementation of [Etancelin et al. 2014] featuring an high order remeshing method and the porting of `CPU` intensive particle remeshing routines onto `GPU` accelerators by using the `OpenCL` standard [Cottet et al. 2014]. This was the genesis of the `OpenCL` computing backend in the `HySoP` library. During this phase, the directional splitting approach was introduced in order to optimize the memory accesses, and thus improve achieved the memory bandwidth, for the `OpenCL`-based remeshing kernels. In the mean time, an hybrid vortex penalization method was developed to be able to handle complex geometries on Cartesian grids [Mimeau et al. 2014] and to solve more challenging problems [Mimeau et al. 2015][Mimeau et al. 2017]. Apart from developing the `HySoP` library, current efforts focus on the integration of high order remeshed particles within the `YALES2` library [Moureau et al. 2011].

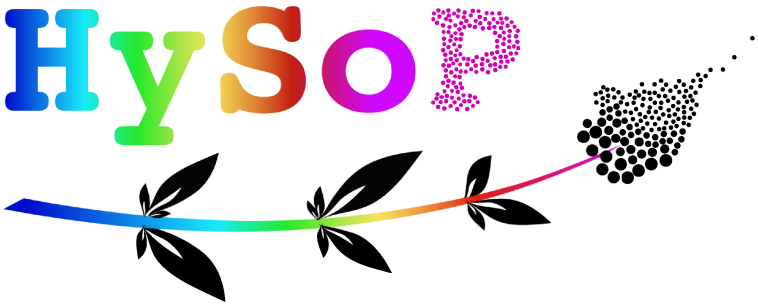


Figure 3.1 – `HySoP` logo representing an hyssop (*Hyssopus officinali*) plant whose seeds are transported by the wind, referring to the lagrangian nature of particle methods.

One of the main objective of this thesis was to continue the porting efforts towards accelerators for other sequential `CPU` operators that quickly became the new simulation bottleneck on `GPU`-enabled compute nodes. In addition to this performance objective, there was also the need to support homogeneous boundary conditions as the library historically only handled periodic boundary conditions which are not suitable for most sedimentation problems. Due to the huge number of features to add to the existing library and the existing partition of code bases, it was decided to fork the project to rethink the entire codebase. The main vision concerning the new implementation was to keep most of the parts that worked in the original library while using the experience gained during those first development years to improve what did not work as expected. From now on the original code prior to the fork will be referred as `hysop-origin` and the current state of the library as `hysop-current`.

Using lines of code (LOC) to measure software progress is like using kilograms for measuring progress in aircraft manufacturing. It is however a good metric to grasp where the developing efforts went during the transition period.

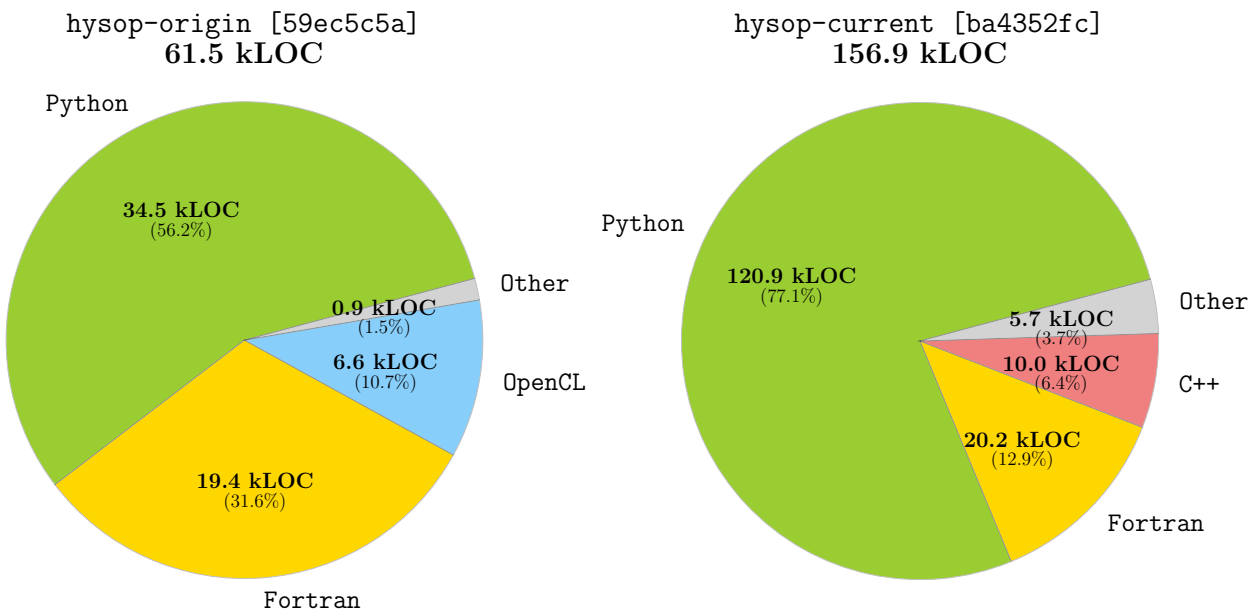


Figure 3.2 – Evolution of the programming languages used in the `HySoP` library.

Figure 3.2 shows that the entire code base has grown by a factor 2.6 in terms of total source code lines. Most of the efforts went into adding new features directly into the core of the library by using the `Python` programming language (x3.5). The `Fortran` backend, implementing legacy operators like the remeshing kernels and some spectral operators has remained roughly the same as the original implementation. The `C++` backend is completely new and mainly addresses non-periodic boundary conditions.

At first glance, the lack of `OpenCL` sources in `hysop-current` might be surprising for a version of the library that pretends to support more `OpenCL` operators compared to its predecessor. This is due to the change of philosophy in the handling of `OpenCL` source code, passing from parsed static kernels to fully code generated ones for more versatility (see section 3.2 for more information). Finally an increase can be seen in terms of tests and continuous integration, represented by the `Other` section that contains mainly scripts, build utilities and `Docker` files used to automate the testing process. During this period `Git` records 1034 changed files, 135k insertions and 62k deletions. The main differences between the two versions are summarized in table 3.1. Details about the original design of the library can be found mostly in [Etancelin 2014] and [Mimeau 2015].

Feature	origin	current	Commentary
Fortran support	✓	✓	The legacy Fortran backend is still supported.
C/C++ support	✗	✓	A new C++ backend has been added to the library.
OpenCL support	partial	full	Only GPU advection and diffusion were supported.
MPI support	full	partial	Currently spectral operators have no MPI support.
JIT support	OpenCL	OpenCL, Numba	Some critical NumPy routines are translated by LLVM.
OpenCL source code	file based	code generated	Implementation of a OpenCL code generator
OpenCL autotuner	✗	✓	The tuner guaranties portable OpenCL performances.
OpenCL symbolic	✗	✓	Generate OpenCL code from symbolic expressions.
Domain dimension	3D*	2D/3D/nD	2D simulations could be done by faking 3D ones.
Boundary cond.	only periodic	homogeneous	Added homogeneous Dirichlet and Neumann BC.
Data types	float/double	any	HySoP now supports all NumPy data types.
Fields/Parameters	scalar, vector	any	Added support for any type of tensor variable.
Memory ordering	transposition	noop	Passing from C to Fortran ordering has no cost.
Operator creation	manual	semi-automatic	Some operators are now automatically generated.
Topology creation	manual	automatic*	Manual topology specification is still possible.
Operator ordering	manual	automatic	Automatic ordering through dependency analysis.
Memory transfers	manual	automatic	Automatic memory transfers between CPU and GPU.
Buffer sharing	manual	automatic	Temporary buffers are shared across all operators.
Problem setup	manual	automatic	This feature enables quick prototyping.

Table 3.1 – Comparison of some features between hysop-origin and hysop-current.

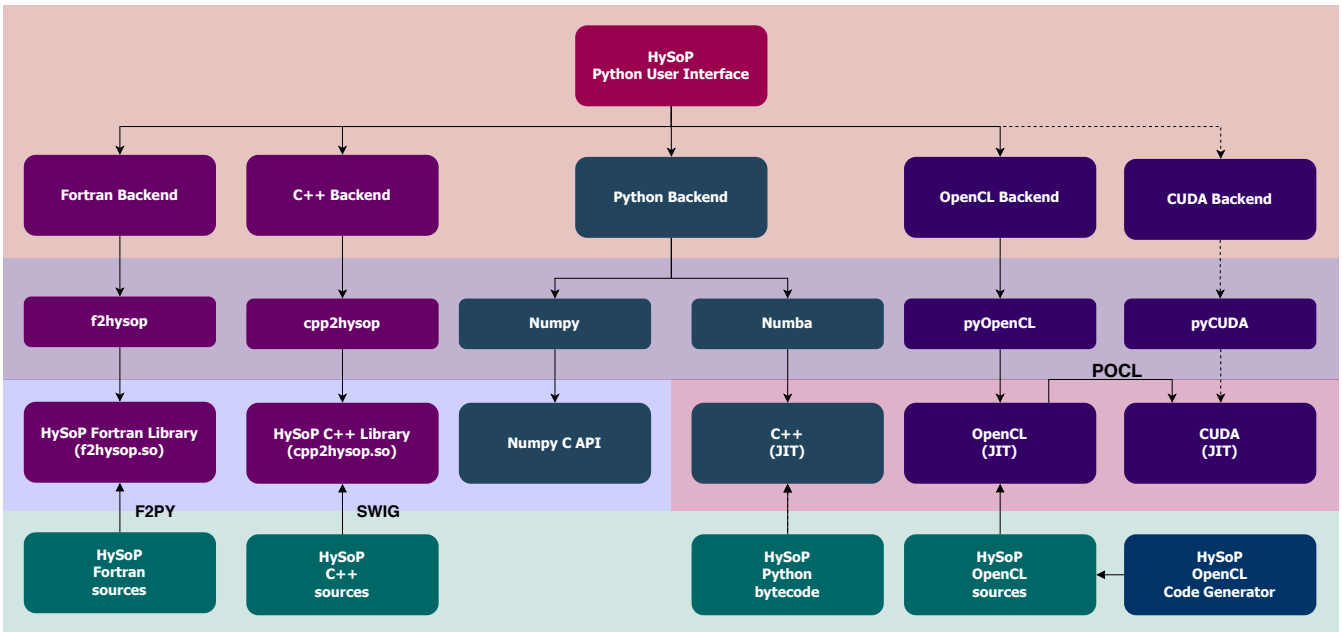


Figure 3.3 – Different computing backends present in the HySoP library.

3.1.2 Design of the library

As stated in the introduction, the library is designed around a `Python` core interface sitting on the top of different computing backends such as `Fortran`, `C++` and `OpenCL`. The end user needs to specify his problem using a `Python` script to interact with the library. Figure 3.3 show the backends interact with the base `Python` layer.

The code is organized around various mathematical and logical concepts.

Physical domain

The first thing the user has to do is to specify the domain of definition of the variables involved in the problem. A physical domain $\Omega \subset \mathbb{R}^n$ represents the physical domain where the simulation will take place. At this time the only available type of domain in the library are n-dimensional rectangular cuboids, obstacles being handled by using a vortex penalization method [Mimeau et al. 2016]. Rectangular boxes can be described by an origin $\boldsymbol{x}_{start} = (x_0, \cdots, x_{n_1})$ and the length of each of its edges $\boldsymbol{L} = (L_0, \cdots, L_n)$. By default the unit box is returned by setting $\boldsymbol{x}_{start} = \mathbf{0}$ and $\boldsymbol{L} = \mathbf{1}$. Apart from defining the physical domain of definition of variables, a domain may also provide information about domain periodicity $\boldsymbol{\mathcal{P}} = (\mathcal{P}_1, \cdots, \mathcal{P}_n)$.

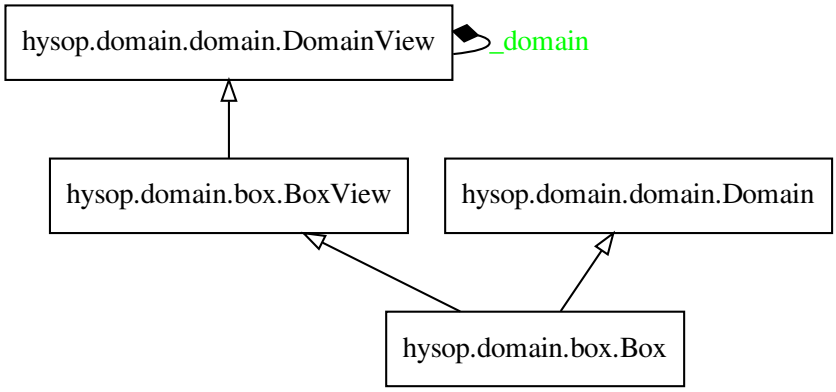


Figure 3.4 – Simplified `UML` diagram representing the `Domain` class hierarchy.

A `Box` is a `Domain` that owns all those properties $(\boldsymbol{L}, \boldsymbol{\mathcal{P}}, \cdots)$, while `BoxView` is a read-only access to those properties with axes permutated. In practice a `Box` is a `BoxView` with default axes $(1, \cdots, n)$. Views will prove themselves useful in the presence of local data permutations.

Continuous fields

Once a domain is defined, it becomes possible to create named continuous scalar, vector or tensor fields. In `HySoP`, a continuous field is an abstract object which represents the usual

vector field, in a mathematical sense, i.e some function which associates integer, real or complex valued scalar, vector or tensor to each point $\boldsymbol{x}$ of the space at a given time t:

$$\begin{aligned} f\colon \Omega \times T &\to \mathbb{K}^{\boldsymbol{m}} = \mathbb{K}^{m_1 \times m_2 \times \cdots \times m_M} \\ (\boldsymbol{x}, t) &\mapsto f(\boldsymbol{x}, t) \end{aligned}$$

with $\Omega \subset \mathbb{R}^n,\ T = [t_{start}, t_{end}],\ \mathbb{K} \subset \mathbb{C},\ n \in \mathbb{N}^*,\ M \in \mathbb{N}^*$ and $m_i \in \mathbb{N}^*\ \forall i \in [\![0, M[\![$.

In practice $M = 1$ for scalar fields, $M = 2$ for vector fields and $M = 3$ for second order tensor fields such as the gradient of velocity. Such objects are used as input and/or output variables of operators and must be defined with at least:

- **a name:** compulsory (required for logs, i/o and code generation).
- **a domain:** the n-dimensional physical domain Ω of definition of the field.

The underlying scalar data type that will be set upon discretization can also be specified. By default fields are set to data type `HYSOP_REAL` which will be either `float` (single precision floating point numbers) or `double` (double precision floating point numbers) depending on `HySoP` build configuration. Integer and complex data types are also supported. Default boundary conditions can also be specified. As an example if the field will only be discretized on boxed domains, one can specify left and right boundary conditions for each of the n axes.

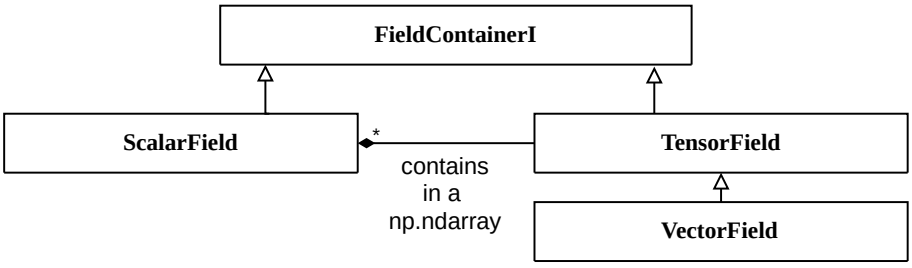


Figure 3.5 – Simplified `UML` diagram of the `ContinuousField` class hierarchy.

Scalar, vector and tensor fields are seen as a scalar field containers and each inherit the common `FieldContainerI` interface such that independent scalar field operations are implemented only once. Each scalar field is associated to a symbolic representation with the help of `Sympy`. Tensor fields are associated to `NumPy` arrays of scalar field symbols. Those symbolic representations can be used later within a code generation framework.

Continuous fields are later discretized on one or more topologies. Topologies are domain discretizations associated to a specific domain decomposition. Before this discretization step, no memory is dynamically allocated for fields.

Parameters

They constitute another source of time-dependent inputs and outputs for operators.

$$\begin{aligned} p\colon T &\to \mathbb{K}^{\boldsymbol{m}} = \mathbb{K}^{m_1 \times m_2 \times \cdots \times m_M} \\ t &\mapsto p(t) \end{aligned}$$

Together with continuous fields, they compose what we will call input and output variables. They are designed like continuous fields that have no space dependencies and follow exactly the same implementation design: `ScalarParameter` and `TensorParameter` can be of any data type and all inherit from a common `ScalarParameterContainerI` interface. They are associated to symbolic variables and the memory required by a given parameter is allocated at its creation (there is no spatial discretization step for parameters). The dependence in time can be removed by declaring the parameter constant and by providing an initial value.

Discretization parameters

A `DiscretizationParameter` instance contains information about the way to discretize a given type of domain (type of the mesh, distribution and number of mesh nodes). A `DiscretizationParameter` does not assume anything about the boundary conditions but the boundary conditions may alter the way fields are discretized because they impose a specific choice of spectral methods. A `CartesianDiscretizationParameter` just contains the number of cells $\boldsymbol{\mathcal{N}}^c$ a Cartesian grid (uniform space step $\boldsymbol{dx}$) should contain in each direction and is compatible with `Box` domains. Other `Box`-compatible variants include `ChebyshevDiscretizationParameter` that also contains the number Chebyshev-Gauss-Lobatto nodes (variable space step $\boldsymbol{dx}$) the resulting regular grid should contain in each direction (here $\boldsymbol{\mathcal{N}}^c$ corresponds the order of the Chebyshev polynomials). Finally `RegularGridDiscretizationParameter` allows the user to enforce the node distribution (uniform or CGL) on a per-axis basis. It can also be configured to deduce the node distributions from the knowledge of the scalar field left and right boundary conditions.

Compatible left and right boundary pairs include:

- Periodic boundary conditions : `PERIODIC-PERIODIC`
- Homogeneous boundary conditions: `ODD-ODD`, `ODD-EVEN`, `EVEN-ODD`, `EVEN-EVEN`
- General boundary conditions: `BC`(α_l, β_l, γ_l)-`BC`(α_r, β_r, γ_r)

The two first kind of boundary pairs are mapped to uniform sample spacings (compatible with the usual discrete Fourier transform and collocated real-to-real transforms). Axes for which general boundary conditions are imposed are mapped to Chebyshev-Gauss-Lobatto nodes (compatible with the fast Chebyshev transform). For performance reasons, the user has to order the axes in such a way that the general boundary condition pairs come first, followed by the periodic boundary pairs and finally homogeneous boundary pairs. This is so that spectral transforms are performed by chaining real-to-real (sine and cosine transforms), real-to-complex (FFT with Hermitian symmetry), complex-to-complex (complex FFT) and finally fast Chebyshev transforms (scaled type one cosine transforms).

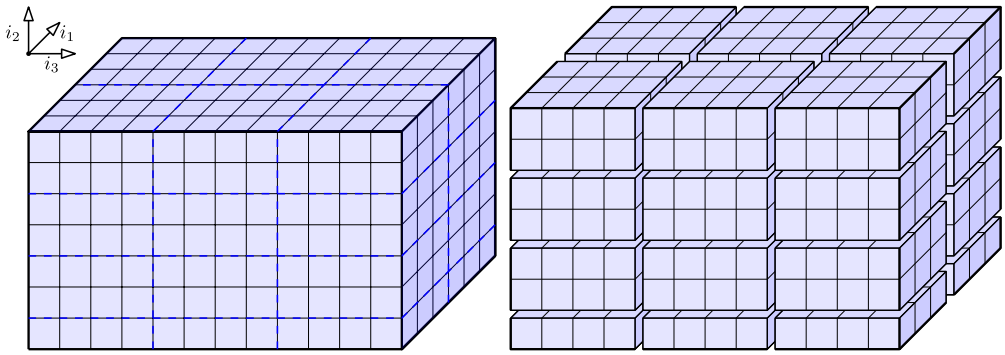
Discretization

A `Discretization` is obtained when bringing together a `Domain` and `DiscretizationParameter`. The most common type of discretization we will use is the `CartesianDiscretization` that is the result of applying a `CartesianDiscretizationParameter` to a `Box` domain. The boxed domain origin $\boldsymbol{x}_{min}$, size $\boldsymbol{L}$ and periodicity $\boldsymbol{\mathcal{P}}$ allows to compute the physical node coordinates $\boldsymbol{x_j}$. The resulting grid has size $\boldsymbol{\mathcal{N}}^v = \boldsymbol{\mathcal{N}}^c + 1 - \boldsymbol{\mathcal{P}}$ and the node coordinates are deduced from the specified node distribution. For regular grids we store only the per direction offsets $(x_{i,j})_{j\in[\![0,\mathcal{N}_i^v[\![}$ for every axe $i \in [\![0, n[\![$. The position of a grid node on a regular grid is then given by $\boldsymbol{x_j} = (x_{1,j_1}, x_{2,j_2}, \cdots, x_{n,j_n})$ and the different node distributions are recalled here:

- **Uniform distribution:** $x_{i,j} = x_{min,i} + j\, dx_i$ with $dx_i = L_i/\mathcal{N}_i^c$.
- **Gauss-Chebyshev-Lobatto distribution:** $x_{i,j} = x_{min,i} + L_i(1 - cos(j\pi/\mathcal{N}_i^c))/2$

Topologies and meshes

A `Topology` brings together a `Discretization`, a `MPI` communicator, a compute backend and a specified number of ghosts $\boldsymbol{G} \in \mathbb{N}^n$. It is responsible to generate a `Mesh` object and handle domain decomposition among all processes of the communicator. For a regular discretization of boxed domains this mainly consists into splitting the global regular grid into smaller local regular grids. Ghosts are extra boundary nodes required by some operators to compute values close to the local subdomain boundaries. A `Mesh` also maps the local-to-process mesh properties to global mesh properties. For the usual regular grid case, one obtain a `CartesianTopology` operating on a `MPI_Cart` communicator (with splitting mask $\boldsymbol{S} \in \{0,1\}^n$ and periodicity mask $\boldsymbol{\mathcal{P}} \in \{0,1\}^n$) on the top of a `CartesianMesh`. The splitting mask $\boldsymbol{S}$ controls the directions in which the domain can be split while the periodicity mask enables or disables the communications between processes handling global domain left and right boundaries. It is required that $\boldsymbol{S} \cdot \boldsymbol{S} \geq 1$ when the number of processes P is more than one so that domain decomposition can happen. The `CartesianTopology` provides facilities to perform local ghost exchanges and accumulation with direct and diagonal neighbor processes. The compute backend dictates the memory allocation policy (where the memory will be allocated) and drives eventual host-to-device and device-to-host memory exchanges required by any `OpenCL` context.



Overall Simulation Domain $\mathcal{B}$ Domain Decomposition

Compute backends

Currently there exist three kinds of compute backends that dictates where memory will be physically allocated and what memory exchanges are required to perform local data redistributions and inter-process communications:

- `HOST_BACKEND`: Memory is allocated in the main memory. This first kind of backend is used to implement `Python` operators (`PythonOperator`) or wrap `C++` and `Fortran` operators (`CppOperator` and `FortranOperator`). With this backend, `MPI` exchanges can be done directly from and to host buffers.

- `OPENCL_BACKEND`: Memory is allocated in the memory of a specific `OpenCL` device on a specific `OpenCL` platform. Dedicated `GPUs` and `MICs` have their own embedded memory and `CPUs` can use `OpenCL` buffers stored the main memory. Here two variants are possible depending on the location of the `OpenCL` buffers. `CPU` devices can map and unmap their memory directly to the main memory such that all `HOST_BACKEND` compatible operators and `MPI` exchanges works seamlessly. Devices providing their own memory require data exchanges between host and device memories. In this case `MPI` exchanges are done less efficiently trough temporary hosts buffers (unlike `CUDA`, `MPI` cannot work directly with `OpenCL` buffers [Wang et al. 2013]).

- `HYBRID_BACKEND`: Memory is allocated in the main memory and is partially allocated on every supplied backend that cannot map memory to host (mostly `OpenCL` devices with dedicated memory banks). This is the only backend that allows hybrid `CPU-GPU` computations or multi-device computations by using only one `MPI` process. `CPU` cores can be used by any `OpenCL` platform compatible with the target `CPU` (such as AMD, Intel and POCL `OpenCL` platforms) or in a multi-threaded context (`Numba`, `MKL-FFT`, `FFTW`).

Operators

An `Operator` represents a function that takes zero or more continuous fields and parameters as inputs and zero or more continuous fields and parameters as outputs. As one may expect, operators do not work with every type of topologies, discretizations and boundary conditions. Each operator proposes a default implementation (on a specific compute backend) and enforce requirements on the discretization of the fields and the layout of the topology on a per variable basis (`FieldRequirements`) and on a global variable basis (`OperatorRequirements`). Operators are usually bound to a compute backend trough the common interfaces (`PythonOperator`, `CppOperator`, `FortranOperator`, `OpenClOperator` and `HybridOperator`) that enforce default requirements such as the memory ordering and memory location.

Simulation

A `Simulation` instance is responsible to handle the time parameter t as well as the timestep parameter dt of a given discretized problem. It calls the underlying problem operators implementation in order at every timestep. The timestep can be either fixed or decided during the current timestep by any operator. Many timestep criterias are implemented in the library such as `CFL` and `LCFL` constraints.

Problems

A `Problem` is a sequence of operators that are organized in a directed acyclic graph (DAG). Operators are created by the user and inserted one by one into the graph. The nodes represent operators while edges represent read and write dependencies of scalar fields and parameters. The order of application of operations is deduced by performing a topological sort of this graph. Apart from giving the order of operations this graph also allows us to extract task parallelism. Subproblems can be represented with subgraphs or operator generators and can be inserted directly into a Problem.

Discrete fields

Discrete fields are continuous fields that have been discretized on a given topology. A single continuous field can be discretized on a given domain on multiple different topologies (different mesh, number of ghosts, compute backend and domain decomposition). The `DiscreteField` class hierarchy is a bit more complicated because it has to handle views. In general `DiscreteScalarField` is a `ScalarField` discretized on a `Mesh` handled by some `Topology`. A `CartesianDiscreteScalarField` is a `ScalarField` discretized on a `CartesianMesh` handled a `CartesianTopology` backed up by a `MPI_Cart` communicator. All processes handling a given `DiscreteScalarField` allocate their own local data (typically one value per local mesh node). The size and location of the memory depends on the chosen discretization, topology and compute backend. For Cartesian meshes, the resulting local subgrids are also regular grids that can be allocated as one big local chunk of contiguous memory.

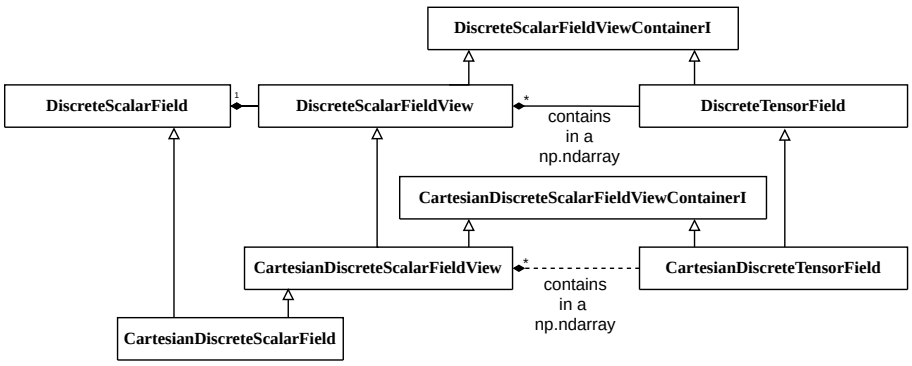


Figure 3.6 – Simplified `UML` diagram of the `DiscreteField` class hierarchy.

A `DiscreteScalarFieldView` is a read-only access to a given `DiscreteScalarField` property (but not necessarily to the discrete field's data). As for continuous fields, discrete fields come with their tensor counterparts (which are M-dimensional containers of `DiscreteScalarFieldViews`). As usual all those types inherit common container interfaces to group common functionalities. An operator can declare to require some temporary discrete fields. In this case the discrete fields data is automatically provided by a temporary buffer memory pool that is common to all operators. The validity of the content of a temporary field (or any other kind of temporary buffer) is tied to the lifetime of the current applied operator.

Topology state

A `TopologyState` is a type of object that can alter the functionality of a `View` (such as `DomainView`, `TopologyView`, `MeshView`, `DiscreteFieldView` and their child class variants). One property is independent of the type of topology and record whether the underlying data should be considered to be read-only or read-write (only relevant to `DiscreteField`s which are objects that own data). A view that is set up with a read-only state will only return host and device buffers marked as read-only (whenever the associated compute backend offers this possibility). The topology state of a `CartesianTopology` is extended to handle a current local transposition state and `C` or `Fortran` ordering. This comes from the fact that once the boxed domain has been decomposed, each process allocates a unique contiguous chunk of local memory for all its required discrete fields. Those two additional states record the actual logical representation of this memory.

Operators are implemented by accessing to domain, topology, mesh, discrete field information through views that are set up during the construction of the problem (the graph of operators) where each topology states are tracked and updated depending on operator requirements. For example accessing the global grid size $\boldsymbol{\mathcal{N}}^v = (\mathcal{N}_z^v \mathcal{N}_y^v, \mathcal{N}_x^v)$ of a three-dimensional discrete field trough a view will return $(\mathcal{N}_y^v, \mathcal{N}_x^v, \mathcal{N}_z^v)$ if the actual local transposition state is set to $(2, 3, 1)$. An operator that enforces the last axis (x-axis) to be continuous in memory for a given field has to ask for either $(1, 2, 3)$ or $(2, 1, 3)$ transposition states trough a `FieldRequirement` for the considered input or output field. The graph generator is then responsible to modify the last known state of the field by inserting automatically operators such as local transpositions to satisfy the operator requirements. The graph builder also automatically handles memory transfers between topologies with different compute backends (host-host, host-device and device-device transfers), and data redistribution for discrete fields that are defined on different topologies.

3.1.3 `HySoP` from a user point of view

In practice, the user only defines its variables (fields and parameters) and builds a problem by inserting operators. Operators can be obtained through simplified operator interfaces that gather all the available implementations. The user specifies associated input and output variables along with their discretizations (most of the time the knowledge of $\boldsymbol{\mathcal{N}}^c$ is enough for uniform Cartesian grids). The library then handles the rest of the operations by generating a directed graph of operators, finding common topologies between operators depending on their requirements and discretizing the fields. Prior to simulation, the initialization of the fields can be done with simple `Python` methods. A typical user script looks like this:

```
#> Compute initial vorticity formula symbolically and translate symbolic
#  expressions to equivalent numerical functions by using numpy and sympy
import numpy as np, sympy as sm
x, y = sm.symbols('x y', real=True)
u, v = sm.tanh(30*(0.25-sm.Abs(y-0.5))), 0.5*sm.sin(2*sm.pi*x)
w = v.diff(x) - u.diff(y)
U,V,W  = (sm.lambdify((x,y), f) for f in (u,v,w))
```

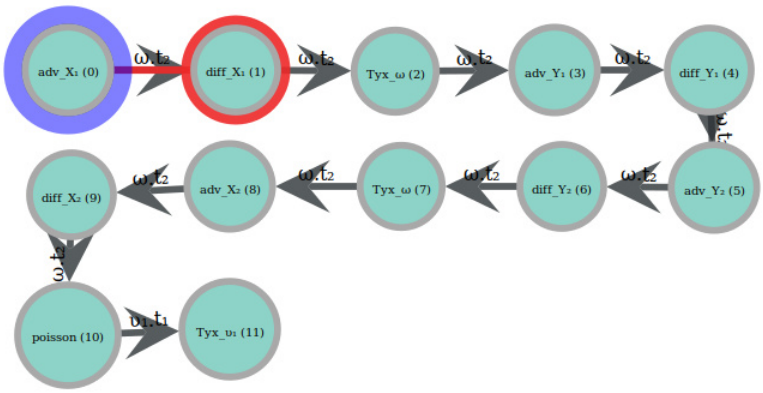
```
def init_velocity(data, coords, component):
    data[...] = U(*coords) if (component==0) else V(*coords)
def init_vorticity(data, coords, component):
    data[...] = np.nan_to_num(W(*coords))

from hysop import Box, Simulation, Problem, MPIParams
from hysop.constants import Implementation
from hysop.defaults  import VelocityField, VorticityField, TimeParameters
from hysop.operators import (DirectionalAdvection, DirectionalDiffusion,
                                 StrangSplitting, PoissonCurl)
ndim, npts = 2, (128,128)

#> Define domain, default MPI communicator and common operator parameters
box = Box(dim=ndim)
mpi_params = MPIParams(comm=box.task_comm)
extra_params = {'mpi_params': mpi_params, 'implementation': Implementation.OPENCL}

#> Variables (time, timestep, velocity, vorticity)
t, dt = TimeParameters()
velo  = VelocityField(box)
vorti = VorticityField(velo)

#> Directionally split advection and diffusion of vorticity (second order Strang)
advection_dir = DirectionalAdvection(velocity=velo, advected_fields=vorti, dt=dt,
     velocity_cfl=2, variables={velo: npts, vorti: npts}, name='adv', **extra_params)
diffusion_dir = DirectionalDiffusion(fields=vorti, coeffs=1e-4, dt=dt,
                        variables={vorti: npts}, name='diff', **extra_params)
splitting = StrangSplitting(ndim, order=2)
splitting.push_operators(advection_dir, diffusion_dir)

#> Poisson operator to recover the velocity from the vorticity
poisson = PoissonCurl(velocity=velo, vorticity=vorti, name='poisson',
                      variables={velo:npts, vorti: npts}, **extra_params)

#> Create and setup the problem (graph of operators)
problem = Problem()
problem.insert(splitting, poisson)
problem.build()

#> Create a simulation, initialize fields and solve the problem
simu = Simulation(start=0.0, end=1.25, dt0=1e-2, t=t, dt=dt)
problem.initialize_field(velo,  formula=init_velocity)
problem.initialize_field(vorti, formula=init_vorticity)
problem.solve(simu)
```

Those fifty lines of `Python` solve a fully periodic two-dimensional Navier-Stokes equation with fixed timestep, discretized on a Cartesian grid of size $\mathcal{N}^c = (128, 128)$ entirely on the `OpenCL` compute backend (default `OpenCL` platform and device are assumed here). It solves the double shear layer case presented in [Brown 1995] (p.17) by using the same ideas as algorithm 4: advection and diffusion of vorticity $\boldsymbol{\omega}$ at fixed velocity $\boldsymbol{u}$ by using the directional splitting approach (introduced in section 2.5), followed by the recovery of the velocity from the vorticity using a spectral Poisson-curl operator (section 2.6). The full example including variable timestep is available as one of the many examples provided by the library, see `hysop/examples/shear_layer/shear_layer.py`.

Because nothing has been specified by the user, the library assumes that all numerical methods use their default parameters. The library defaults to:

- **Time integrator:** explicit second order Runge-Kutta scheme (`RK2`).
- **Finite differences:** explicit fourth order centered finite differences (`FDC4`)
- **Interpolation:** linear interpolation (`LINEAR`)
- **Remeshing kernel:** C^2 with four preserved moments $\Lambda_{4,2}$ (`L42`)

The `DirectionalDiffusion` operator defaults to explicit finite differences with default time integrator (`RK2`+`FDC4`) and `DirectionalAdvection` operator to default time integrator, linear interpolation of velocity and default remeshing kernel (`RK2`+`L42`+`LINEAR`). Default boundary conditions and discretization are all set to `PERIODIC` and `CartesianDiscretization` so that the `PoissonCurl` operator uses a Fourier spectral solver on a regular Cartesian grid. The library then deduces the number of required ghosts depending on the order of the numericals methods, build the topologies and the graph of operators. The resulting DAG can be obtained by calling `problem.display()` right after `problem.build()`. Operators, represented by nodes have default names that can be overridden by the user (here 'adv', 'diff', 'poisson' corresponds to `DirectionalAdvection`, `DirectionalDiffusion` and `PoissonCurl` operators). For performance reasons, the library automatically inserts `LocalTranpose` operators that are responsible to make the current axe contiguous in memory in directional splitting contexts (default name 'T'). The edges show the variable dependencies between operators. Edge dependencies are shown as discrete fields (continuous field + topology index) or parameters:



3.1.4 Graph of operators and task parallelism

The `HySoP` library is responsible to generate a Directed Acyclic Graph of operators based on user input. In this graph, the vertices represent tasks to be performed (operators), and the edges represent constraints that one task must be performed before another. This DAG of tasks is then analyzed to extract independent execution paths that can be scheduled on available hardware [Kwok et al. 1999][Simon 2018][Canon et al. 2018]. Once the user has described and inserted all operators, the graph builder proceeds as the following:

1. **Generate initial directed graph:** Iterate over all operators as ordered by the user and compute the following on a per operator basis:

   (a) Iterate over operator input fields and check their current topology state. If the current topology state does not match one of the operator required topology states, do not insert this operator directly into the graph but insert as many as required local permutations and topology redistributions as required to fulfill current operator needs. Data redistribution include host-device memory transfers as well as inter-process communications.

   (b) Handle input dependencies: For each input scalar field, insert an edge from the operator lastly wrote the field to the current one. Add this operator to the reading record of all input fields.

   (c) Handle output dependencies: For each output scalar field, insert an edge from all operators that are reading the field to the current one. Clear the reading and writing records of all output fields and add this operator to the writing record. From the knowledge of all the input field topology states, compute resulting output field topology states and update the topology state record of all output fields.

   An example of initial DAG obtained after this step is given on figure 3.7.

2. **Insert additional operators to match input topology states**: Additional local permutations and redistributions of data may be required in order to loop over a problem (output states have to match input states). The graph is not closed so that is does not contain any directed cycles.

3. **Perform topological sort:** Apart from drastically reducing the number of edges contained in the graph, the result of the topological ordering gives a valid execution sequence of the tasks.

4. **Extract independent execution queues:** Iterate on operators in topological order to find out which one can be executed concurrently. Generate a new work queue until all independent operators are affected (see figure 3.8).

Independent tasks can be distributed on different compute nodes (different processes) or locally on the same compute nodes by using multi-threading, `OpenCL` device fission or asynchronous programming techniques (or a mix of all those methods). Support of compute-node level task parallelism has already been implemented in [Etancelin 2014].

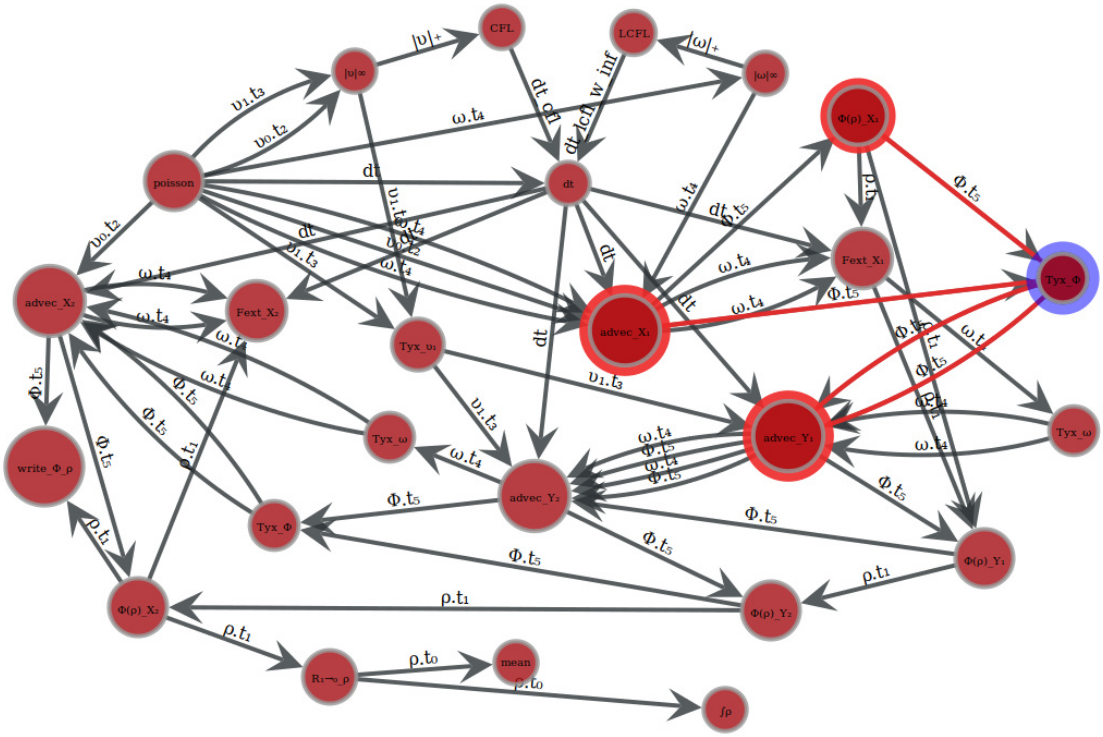


Figure 3.7 – Example of directed graph of operators obtained after step one

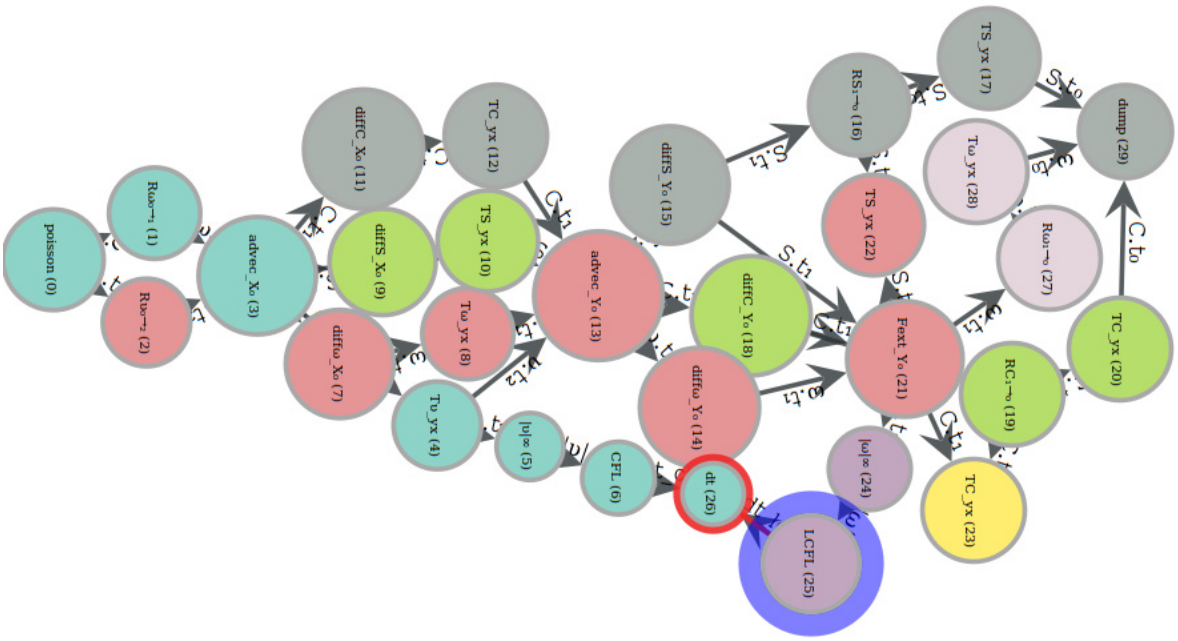


Figure 3.8 – Example of DAG of operators obtained after step four: Operators are represented by the nodes and each color represent an independent execution path. On this graph there can be up to seven different operators that are executed at the same time.

3.1.5 Integration of numerical methods

When required, the `HySoP` library dynamically generates the exact numerical coefficients of the numerical schemes. This includes finite-differences coefficients and remeshing formulas (polynomial coefficients), which values can be obtained by solving a corresponding linear system (equation 2.37). Those linear systems are solved by using infinite precision rational numbers based linear solvers provided either by `Sympy` or `Python-FLINT` `Python` modules. Implementing a new explicit Runge-Kutta scheme only requires the user to provide a Butcher tableau (equation 2.31). Those numerical schemes are used later for the implementation of host-side operators but also device-side operators such as in the `OpenCL` code generation framework. The code generation framework allows kernels to be specialized and optimized exactly for user chosen numerical methods. Once generated, coefficients are cached to disk.

For efficiency reasons, explicit centered finite differences associated to multi-stage time integrators are treated in one go without any ghost exchanges happening in the middle of the process. In a given direction, this implies to multiply the number of required ghosts g by the number of stages s, resulting in a ghost layer of size $s\,g$. The number of required ghosts g is at least the spatial order of approximation of the derivatives divided by two. This means that for fourth order centered finite differences `FDC4` (requires 2 ghosts vertices, see section 2.4.1) and a second order Runge-Kutta time integrator `RK2` (two integration steps), the ghost layer will be of size 4. With this configuration, one-dimensional diffusion would be solved as the following:

$$u = u(x,t), \text{ solve } \frac{\partial u}{\partial t} = \frac{\partial u^2}{\partial x^2} \text{ on } \Omega \text{ from } t = t_0 \text{ to } t_0 + dt$$

$$u_i^{k,0} = u_i^n \quad \forall i \in [\![-4, N+3]\!]$$

$$F_i^{k,0} = \left[\frac{-u_{i-2}^{k,0} + 16u_{i-1}^{k,0} - 30u_i^{k,0} + 16u_{i+1}^{k,0} - u_{i+2}^{k,0}}{12\ dx^2}\right] \quad \forall i \in [\![-2, N+1]\!]$$

$$u_i^{k,1} = u_i^{k,0} + \frac{dt}{2}\ F_i^{k,0} \quad \forall i \in [\![-2, N+1]\!]$$

$$F_i^{k,1} = \left[\frac{-u_{i-2}^{k,1} + 16u_{i-1}^{k,1} - 30u_i^{k,1} + 16u_{i+1}^{k,1} - u_{i+2}^{k,1}}{12\ dx^2}\right] \quad \forall i \in [\![0, N-1]\!]$$

$$u_i^{k+1} = u_i^{k,0} + dt\ F_i^{k,1} \quad \forall i \in [\![0, N-1]\!]$$

u_{-4}^k	u_{-3}^k	u_{-2}^k	u_{-1}^k	u_0^k	u_1^k	u_2^k	u_3^k	u_4^k	u_5^k	u_6^k	u_7^k	u_8^k	u_9^k	u_N^k	u_{N+1}^k	u_{N+2}^k	u_{N+3}^k
$u_{-4}^{k,0}$	$u_{-3}^{k,0}$	$u_{-2}^{k,0}$	$u_{-1}^{k,0}$	$u_0^{k,0}$	$u_1^{k,0}$	$u_2^{k,0}$	$u_3^{k,0}$	$u_4^{k,0}$	$u_5^{k,0}$	$u_6^{k,0}$	$u_7^{k,0}$	$u_8^{k,0}$	$u_9^{k,0}$	$u_N^{k,0}$	$u_{N+1}^{k,0}$	$u_{N+2}^{k,0}$	$u_{N+3}^{k,0}$
		$F_{-2}^{k,0}$	$F_{-1}^{k,0}$	$F_0^{k,0}$	$F_1^{k,0}$	$F_2^{k,0}$	$F_3^{k,0}$	$F_4^{k,0}$	$F_5^{k,0}$	$F_6^{k,0}$	$F_7^{k,0}$	$F_8^{k,0}$	$F_9^{k,0}$	$F_N^{k,0}$	$F_{N+1}^{k,0}$		
		$u_{-2}^{k,1}$	$u_{-1}^{k,1}$	$u_0^{k,1}$	$u_1^{k,1}$	$u_2^{k,1}$	$u_3^{k,1}$	$u_4^{k,1}$	$u_5^{k,1}$	$u_6^{k,1}$	$u_7^{k,1}$	$u_8^{k,1}$	$u_9^{k,1}$	$u_N^{k,1}$	$u_{N+1}^{k,1}$		
				$F_0^{k,1}$	$F_1^{k,1}$	$F_2^{k,1}$	$F_3^{k,1}$	$F_4^{k,1}$	$F_5^{k,1}$	$F_6^{k,1}$	$F_7^{k,1}$	$F_8^{k,1}$	$F_9^{k,1}$				
				$u_0^{k,2}$	$u_1^{k,2}$	$u_2^{k,2}$	$u_3^{k,2}$	$u_4^{k,2}$	$u_5^{k,2}$	$u_6^{k,2}$	$u_7^{k,2}$	$u_8^{k,2}$	$u_9^{k,2}$				

3.2 Implementation on hardware accelerators

One of the main objective of this thesis was to pursue the OpenCL development efforts towards hardware-accelerated numerical simulation in HySoP, initiated by [Etancelin 2014]. Graphics Processing Units (GPUs) and other accelerators promise tremendous advantages in throughput over conventional processor architectures. Those advantages ideally result in a large reduction of execution time for suitable compute or bandwidth-bound algorithms. However, execution time is not the only time scale to consider when comparing architectures. The development time for a scientific code will, in many cases, constitute a significant fraction of its useful lifespan. CUDA and OpenCL are the two competing standards capable of driving GPUs [Su et al. 2012b]. They both provide a language that is a subset of C or C++ and a runtime library dedicated to compile and execute code on compatible target devices. A short overview and comparison of both those standards is proposed in [Tompson et al. 2012].

Within the HySoP library, the choice of the OpenCL standard over CUDA has been made so that more devices can be supported within a single implementation (such as CPUs, MICs and GPUs from any vendor). Under fair comparison, OpenCL offers roughly the sames performances guaranties than CUDA [Fang et al. 2011]. The HySoP library requires the OpenCL platforms to be compliant with the OpenCL 1.2 standard:

- OpenCL 1.1 offers memory operations on subregions of a buffer including read, write and copy of 1D, 2D, or 3D rectangular subregions [Khronos2010]. This is required to perform efficient boundary layer exchanges (ghost exchanges).

- OpenCL 1.2 introduces device partitioning and IEEE 754 compliance for single-precision floating point math [Khronos2011] required for the correctness of single-precision numerical routines.

While the OpenCL standard provides a unified abstraction to widely divergent hardware architectures, performance portability remains a difficult problem, even for contemporary competing architectures available today [Klöckner et al. 2012]. As an example, writing efficient software for GPUs using low-level programming environments such as OpenCL or CUDA requires the developer to map computation directly onto GPU architectural design (number of compute units, on-chip memory type and size, designed arithmetic intensity, optimal memory access patterns, ...). Architectural design often being unavailable to the programmer, GPU programming relies extensively on experimentation and microbenchmarking.

This section describes the new strategies that have been incorporated into the HySoP library to reduce the time passed in OpenCL development cycles and to guaranty some performance portability across nowadays, and hopefully future generations of architectures. The resulting implementation is able to run fully accelerated Navier-Stokes simulations on CPUs and GPUs. It can be extended by users without prior OpenCL experience by using a code generation framework based on symbolic expressions. Benchmark results are given for four reference compute platforms ranging from consumer grade CPUs to server grade GPUs (section 1.4.5). The resulting implementation is validated on a three-dimensional reference simulation.

3.2.1 Arrays, buffers and memory handling

All the memory is allocated prior to simulation during the building of the problem. Problem building is constituted of the following three steps:

1. **Initialization:** Each operator contained in the problem is initialized, operator topology requirements are set up (min and max number of ghosts $\boldsymbol{G}$, splitting mask $\boldsymbol{S}$, local transposition state $\boldsymbol{T}$, `C` or `Fortran` data ordering, compute backend, ...).

2. **Discretization:** the topology requirements are collected at a scalar field level and compatible requirements are merged together in order to create the minimal number of different topologies for a given scalar field. Topologies are created and continuous scalar fields are discretized on each of their associated topologies. The graph of operators is created and simplified with topological sort, giving the order of application as well as independent execution of operators.

3. **Setup:** Each operator asks for extra temporary buffers to work with. Those extra work buffers can be of any shape and data type and can even require special memory alignment. Temporary fields memory requirements are automatically registered. All the memory requests are collected, reduced to one-dimensional contiguous chunks of memory and allocated in common memory pools. Each memory sharing group (execution queue and compute backend pair) gets its own memory pool. Those raw contiguous buffers are then sliced, reshaped and viewed as any other data type to suit each of the memory requests. Temporary buffers obtained by this mechanism are only considered valid while the operator is being applied because of this memory sharing policy (the temporary buffer content may be overwritten by any subsequent operator). This of importance since `GPU` memory is usually a precious ressource [Rupp 2013].

Dynamic allocation of memory is a slow operation and is not suited for time critical sections of the code such as the execution of an operator. This is mainly why all the memory is preallocated in the discretization and setup steps. It is not released until the very end of the simulation through a call to `problem.finalize()`. The memory is either provided by `Python` or `OpenCL` buffers which are respectively wrapped by `numpy.ndarray` or `pyopencl.Array` n-dimensional array types. The `NumPy` library is responsible to allocate host side memory while the `PyOpenCL` library is responsible to allocate device memory. A `pyopencl.Array` is a `numpy.ndarray` work-alike that stores its data and performs its computations on a given `OpenCL` compute device. `PyOpenCL` provides some functionalities that are equivalent to `numpy` through a runtime code generation framework provided by the library itself [Klöckner et al. 2012]. This framework includes elementwise algebraic operations, reductions and scans with automatic type promotion. Basically, each different array operation triggers the code generator, the compilation of the generated code and the call to the resulting `OpenCL` kernel (a kernel refers to a function that can be executed on a `OpenCL` device). As an example, the expression $a+b*sin(c)$ where a, b and c are three n-dimensional arrays will generate, compile and execute three different kernels ($sin(\bullet)$, $\bullet * \bullet$ and $\bullet + \bullet$) for every different shape and data type of the operands (the library also supports broadcasting of scalars). It also requires to dynamically allocate three temporary arrays of the same shape and datatype as the intermediate result

for every kernel. This leads to the generation of kernels with smaller arithmetic intensities and dynamic allocation of temporaries. Evaluating array expressions by using overloaded operators is thus less efficient. It is possible to use the lower level kernel generation interfaces of the `PyOpenCL` library (`ElementwiseKernel`, `ReductionKernel` and `GenericScanKernel`) to generate kernels that evaluate multi-stage expressions on several operands at once. Moreover, generated kernel objects can be kept so that they can be used again later.

We take advantage of this kernel generation framework to extend the support of n-dimensional arrays to a broader range of `NumPy` functionalities. The `HySoP` array implementation include support for the following subset of `NumPy` functionalities: binary operations (elementwise bit operations), logic functions (truth value testing, array contents, logical operations and comparisons), mathematical functions (trigonometric functions, hyperbolic functions, rounding, sums and products, exponents and logarithms, other special functions, arithmetic operations, complex numbers functions and miscellaneous), random sampling (uniform distributions), statistics (order statistics, averages and variances). See https://docs.scipy.org/doc/numpy/reference/routines.html for corresponding numerical routines. The support of standard host-to-host sliced array memory transfer is extended to host-to-device, device-to-host and device-to-device memory transfers by performing multiple calls to `clEnqueueCopyBufferRect` so that the user never bothers to call any low-level `OpenCL` function. It is powerful enough to support and encourage the creation of custom application by its users. All of the implemented `NumPy`-like routines are tested against the reference `NumPy` ones for various shapes and data type associations such that the `HySoP` array interface closely mimic the `NumPy` one. This test is one of the many tests present in the `HySoP` test suite that is driven by the continuous integration software development practice [Duvall et al. 2007][Meyer 2014].

The need for such feature-complete implementations of `numpy` compatible array interfaces targeting accelerators has recently led to the release of two new `Python` modules, namely `CuPy` for `CUDA`-compatible arrays [Nishino et al. 2017] and `ClPy` for `OpenCL`-compatible arrays [Higuchi et al. 2019]. Some other interesting `C++` libraries such as `Thrust`, `VexCL` and `ArrayFire` also offers n-dimensional array support for various compute backends (although they do not support the `NumPy` interface). `Thrust` is a library that offers parallel algorithms on arrays based on the Standard Template Library (STL) on `C++`, `CUDA`, `OpenMP` and `TBB` compute backends [Bell et al. 2012]. Multiple expressions involving arrays are automatically merged by using `C++` expression templates [Veldhuizen 1995] and iterators. `VexCL` provide the same ideas for `OpenCL` [Demidov 2012]. Unfortunately those two libraries do not offer any `Python` bindings. `ArrayFire` has array support for `CPU`, `CUDA` and `OpenCL` compute backends [Malcolm et al. 2012]. It recently got `Python` support through the `arrayfire-python` module [Chrzeszczyk 2017]. Multiple expressions involving arrays are automatically merged by its own just-in-time engine. Those libraries provide productive high-performance computational software ecosystems for creators of scientific libraries. All the `Python` solutions being posterior to the beginning to this work, the `HySoP` module uses its own implementation of n-dimensional array functionalities backed up by `PyOpenCL` code generation capabilities.

3.2.2 OpenCL code generation framework

OpenCL 1.2 is a subset of the C language, and because target architectures are often not known is advance, OpenCL sources are usually compiled at runtime. Just-in-time (JIT) compilation is thus the natural way to obtain executable binaries (kernels) in OpenCL. For simple application, the source code can be loaded from external files or directly embedded into the application as a literal string constants. When the source code has to be modified to suit some of the user inputs, the JIT capability makes it possible to generate code at runtime instead. This is known as Run-Time Code Generation (RTCG) and is used in all of the array libraries presented in the previous section. Code can be generated by three different methods:

- **Simple string formatting:** Using raw Python builtin string formatting capabilities (PEP 3101) is already sufficient for a large range of applications. It can be used to inject specific values for given keywords such as the data type and loop ranges. This simple technique can compensate for the lack of generic programming capabilities such as C++ templates that are available in CUDA and OpenCL 2.x kernels. Note that it is also possible to use the C-preprocessor [Stallman et al. 1987] within OpenCL source code and to provide custom defines at compilation.
- **Textual templating:** When the generated code depends on some control flow while the code variants are textually related, it may be easier to use a templating engine that can do more than simple keyword substitution. As an example, the templating engine used by PyOpenCL to provide code generation capabilities such as ElementwiseKernel is Jinja2 [Ronacher 2008].
- **Code generation from abstract syntax trees (AST):** When the generated code variants are not textually related, it may become appropriate to introduce a full representation of the target code in the host language (here Python). Abstract syntax trees allow code to be generated by using all the facilities provided by the host language. In Python, C-like code generation can be done with a variety modules such as CodePy, cgen and cfile but the most mature AST based code generator seems to be provided by Sympy (even if it does not offer direct OpenCL support). A list of scientific libraries using AST based code generation techniques can be found in [Terrel 2011].

Runtime code generation has been used in the HySoP library since the first OpenCL implementation [Etancelin 2014]. The library was capable of dynamically generating efficient advection, remeshing, diffusion and transposition OpenCL kernels by using file-based static sources, the C-preprocessor to select the numerical methods and a custom source parser to enable the vectorization of expressions. This method would be classified as a simple string formatting technique. The main disadvantage with this approach is that each new numerical method requires sources and C macros to be modified, even if it is just a simple variant of an existing one. The source code is also obfuscated in the presence of many preprocessor macros.

In the present work, this code generation framework has been replaced by two complementary AST-based approaches:

1. An OpenCL code generation framework based on Python context managers (PEP 343) capable of generating any OpenCL kernel directly in Python. Context managers are used to handle code sections and code blocks such that the code generator does reflect the structure of the generated code [Filiba 2012]. Inside a block, each individual line of code is then generated by using simple Python string formatting capabilities. This process is assisted by many helper classes representing OpenCL variables, arrays, and functions to be called.

2. An OpenCL code generator based on the Sympy AST representation. This is particularly useful to automatically generate and vectorize symbolic expressions supplied by the user.

The code generating task is facilitated by the PyOpenCL library that already provides mappings between OpenCL vector types and NumPy types and a way to convert NumPy structured types to OpenCL ones (structured data has no reason to be identically laid out in memory between the host and device because of alignment concerns). The first code generator is responsible to generate all the OpenCL source codes at kernel and function level (type declaration, kernel, functions, arguments, variables). Together with simple string formatting, it can be used to generate any source code directly in Python. The following piece of code generates a custom kernel that performs n-dimensional array copy with arbitrary shape and stride up to $n = 3$:

```
class CopyKernel(KernelCodeGenerator):
  def __init__(self, work_dim, known_vars, **kwds):
    kargs = self.gen_kernel_arguments(work_dim, 'float', **kwds)
    super(CopyKernel, self).__init__(name='copy_kernel', work_dim=work_dim,
        kernel_args=kargs, known_vars=known_vars, **kwds)
    self.gencode(known_vars=known_vars, **kwds)

  def gen_kernel_arguments(self, work_dim, btype, **kwds):
    kargs = ArgDict()
    kargs['src'] = CodegenVariable(name='src', ctype=btype, nl=True, ptr=True,
      ptr_restrict=True, add_impl_const=True, storage='__global', const=True, **kwds)
    kargs['dst'] = CodegenVariable(name='dst', ctype=btype, nl=True, ptr=True,
      ptr_restrict=True, add_impl_const=True, storage='__global', **kwds)
    kargs['array_shape'] = CodegenVectorClBuiltin('N', 'int', work_dim, **kwds)
    return kargs

  def gencode(s, **kwds):
    loop_index  = CodegenVectorClBuiltin('k', 'int', s.work_dim, **kwds)
    src, dst, lid, local_size, global_size, array_shape = map(s.vars.get, ('src',
      'dst', 'local_id', 'local_size', 'global_size', 'array_shape'))
```

```
    loop_context = lambda i: s._for_('{j}={start}; {j}<{N}; {j}+={step}'.format(
      j=loop_index[i], N=array_shape[i],
      start=0 if i else lid[i], step=global_size[i] if i else local_size[i]))
    loop_contexts = tuple(loop_context(i) for i in xrange(s.work_dim-1,-1,-1))

    array_strides = tuple('*'.join(tuple(str(array_shape[j])
                for j in xrange(i))) if i else '1'
                for i in xrange(s.work_dim))
    array_offset = '+'.join('{}*{}'.format(loop_index[i], array_strides[i])
                for i in xrange(s.work_dim))

    with s._kernel_():
      s.decl_aligned_vars(lid, local_size, global_size, const=True)
      s.decl_vars(loop_index)
      with contextlib.nested(*loop_contexts):
        dst.affect(s, init=src[array_offset], i=array_offset)
```

Context managers (such as `_block_`, `_kernel_`, `_function_`, `_for_`, `_if_` and their variants) help to keep code blocks organized. The development process is simplified with helper routines such as `edit()` that opens the generated code in the default terminal editor and `test_compile()` that tries to compile the code for a designated `OpenCL` device. The previous code generates the following code for `work_dim` set to 3:

```
__kernel void copy_kernel(__global const float *const restrict src,
                          __global       float *const restrict dst,
                          int3 N) {

  const int3 lid = (int3)(get_local_id(0),get_local_id(1),get_local_id(2));
  const int3 L   = (int3)(get_local_size(0),get_local_size(1),get_local_size(2));
  const int3 G   = (int3)(get_global_size(0),get_global_size(1),get_global_size(2));

  int3 k;
  for (k.z=0; k.z<N.z; k.z+=G.z) {
    for (k.y=0; k.y<N.y; k.y+=G.y) {
      for (k.x=lid.x; k.x<N.x; k.x+=L.x) {
        dst[k.x*1+k.y*N.x+k.z*N.x*N.y] = src[k.x*1+k.y*N.x+k.z*N.x*N.y];
      }
    }
  }
}
```

The code generator offers helper routine to handle `struct`, `union`, `enum` and various type of variables such as `PyOpenCL` arrays that each can have given different offset, strides and data

type. Functions are implemented as separate code generators such that they can be reused across different kernels (and other functions). The code generator uses by default eleven ordered code sections to generate valid `OpenCL`: the first block consists in `C`-preprocessor macros and is followed by `enum` definitions (enums cannot be forward declared) and `union`, `struct`, `function` and `kernel` prototypes. Those six header sections are followed by `union` and `struct` declarations, global scope constants and finally `function` and `kernel` implementations. The direct implication of runtime code generation is that most variables are now known at compile time. Within the previous code sample, this include the array shape `N` but also local and global work sizes (`L` and `G` which specify respectively the number of work-items that make up a work-group and the total number of global work-items that have to be spawned). The knowledge of those variables allows the compiler to optimize the generated code appropriately. To give an example, the same code is regenerated with `work_dim` set to 2 and by using the `known_vars` capabilities of the generator and providing the value of `N`, `G` and `L`:

```
/* kernel_prototypes */
__kernel __attribute__((reqd_work_group_size(512,1,1)))
void copy_kernel(__global const float *restrict src,
                 __global       float *restrict dst);

/* global_scope_constants */
__constant const int2 N = (int2)((+1024),(+1024));

/* kernel_declarations */
__kernel void copy_kernel(__global const float *const restrict src,
                          __global       float *const restrict dst) {

  const int2 lid = (int2)(get_local_id(0),get_local_id(1));
  const int2 L   = (int2)((+512),(+1));
  const int2 G   = (int2)((+512),(+512));

  int2 k;
  for (k.y=0; k.y<N.y; k.y+=G.y) {
    for (k.x=lid.x; k.x<N.x; k.x+=L.x) {
      dst[k.x*1+k.y*N.x] = src[k.x*1+k.y*N.x];
    }
  }
}
```

The generated kernel signature now exposes one less argument (that has become a global scope constant) and enforces an explicit `work_group_size` that is set to $(512, 1, 1)$. This simple kernel already raise the problem of choosing appropriate values for the work-group size and the global work size for given device and array shape. In general, the values of `L` and `G` are obtained by a simple heuristic. For a given device, better values may be obtained by performing kernel microbenchmarking.

Sympy offers a way to generate compilable code directly from Sympy expressions. (see http://docs.sympy.org/latest/modules/utilities/codegen.html). In particular its C code generator can be extended by implementing code printers to handle symbolic representations of additional OpenCL constructs. Each Sympy symbolic expression is represented by an underlying AST where each node is either a subexpression or a concrete symbol. With such a representation it is possible to determine the OpenCL type (int, float8, custom complex type, ...) of each node by taking into account the implicit casting rules of the language. It also makes it possible to inject additional explicit cast and conversion expressions to handle the non-implicit cases of promotion or demotion of builtin vector types (see [Khronos2011], section 6.2). The idea behind this approach is to propose to the users (and implementers) a way to generate efficient OpenCL based operators without having to use the tedious class-based code generator to generate custom elementwise kernels.

Code generation from symbolic expressions use the class-based code generator under the hood to generate the final code. It is mainly capable of generating elementwise operations, the iteration shape being determined by field of array symbols present in the expressions. The generated code targets contiguous cached memory accesses and the optimal kernel configuration and runtime parameters are automatically determined by kernel microbenchmarking (see subsection 3.2.3). The resulting symbolic DSL also incorporate features aimed at generating kernels that are able to integrate PDEs by using explicit finite differences with explicit Runge-Kutta time integrators. The basic symbolic codegen interface allows to mimic the capabilities provided by PyOpenCL elementwise kernels but with additional automatic tuning of kernel runtime parameters. Symbolic operator interfaces are also provided such that once a kernel has been generated, it is automatically wrapped into an operator and its topology requirements are automatically set up depending on determined numerical methods. Finally the process of splitting expressions according to directional derivatives (as presented in section 2.5) is automated to generate directional operators from general symbolic expressions.

To sum up, given four arrays (a, b, c, d) of given shape and data type it is possible to compute $d = a + b * sin(d)$ on the OpenCL backend by using one of the following code generation frameworks provided by the HySoP library:

- Simple elementwise operations on arrays based on the PyOpenCL stack:
```
kernel = array_backend.nary_op((A,B,C), (D,),
            operation='y0[i] = x0[i]+x1[i]*sin(x2[i])',
            build_kernel_launcher=True)
```
- Elementwise operations from symbolic expressions with automatic parameter tuning:
```
As, Bs, Cs, Ds = OpenClElementwiseKernelGenerator.arrays_to_symbols(A,B,C,D)
expr = Assignment(Ds, As+Bs*sympy.sin(Cs))
kernel = kernel_generator.elementwise('custom_kernel', expr)
```
- A more general code generator requiring the implementation of a custom kernel class.
```
class CustomKernel(KernelCodeGenerator):
    ...
```

The last method offers more flexibility but requires the user to master the OpenCL standard.

The HySoP symbolic code generator offers good performances when compared to state of the art array libraries. The performance is first compared for OpenCL CPU platforms where HySoP outperforms all considered array libraries. The Xeon E5-2695v4 achieves 57.99 GB/s (42.5% of peak memory bandwidth) while the Core i7-9700K achieves 14.16 GB/s (66.5% of peak memory bandwidth) for double-precision arrays of $512^3 = 2^{27}$ elements (1GiB / array).

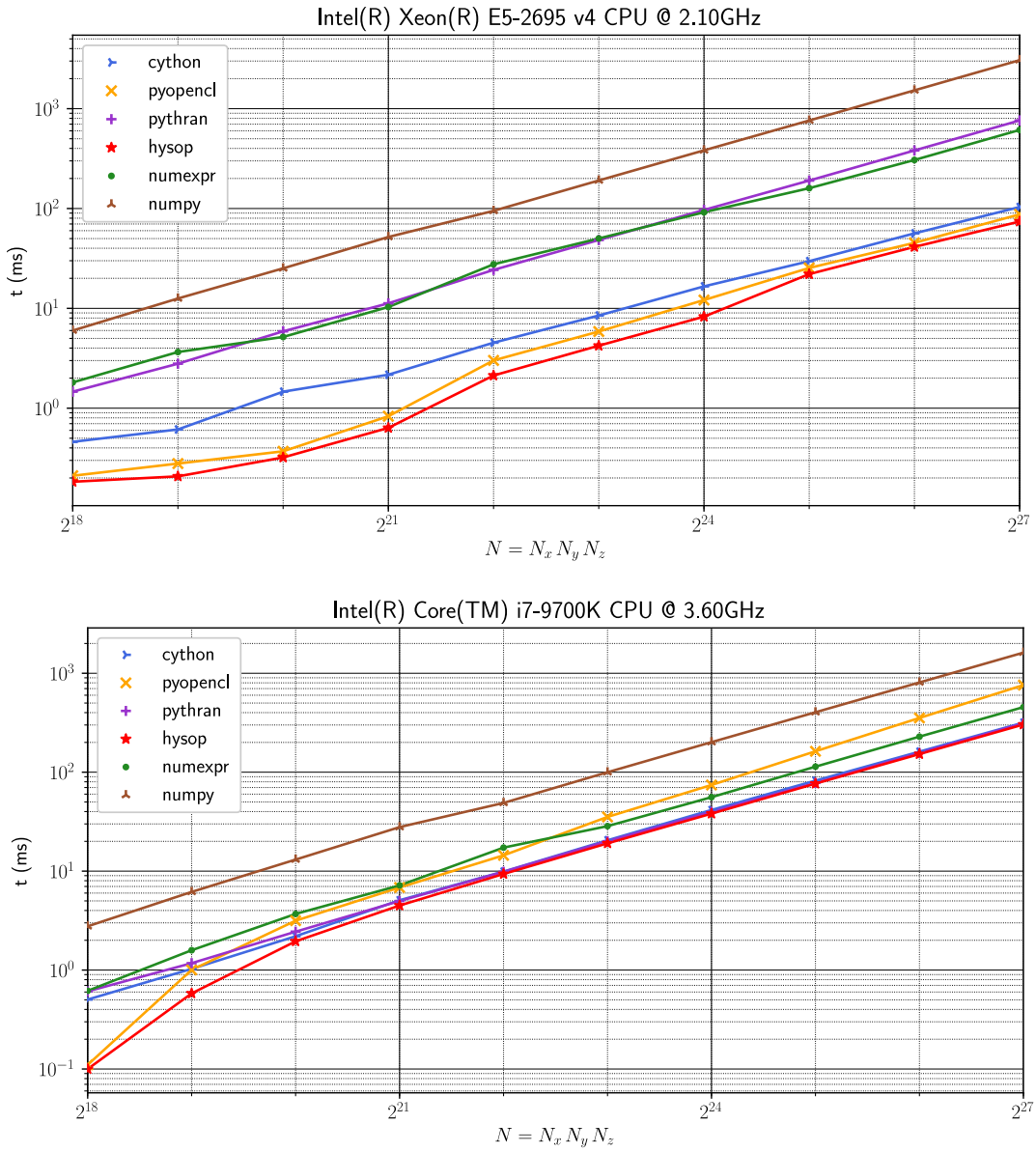


Figure 3.9 – Three dimensional array benchmark on CPU: Mean runtime required to compute d=a+b*sin(c) over 1024 runs where a, b, c, and d are double-precision arrays of shape (N_z, N_y, N_x). This computation is run on the following array backends: Numpy [Van Der Walt et al. 2011] (sin, multiply, add), Numexpr [Cooke et al. 2017], Pythran [Guelton et al. 2015] (-DUSE_XSIMD -fopenmp -march=native -O2), Cython [Behnel et al. 2011] (-fopenmp -march=native -O2), PyOpenCl [Klöckner et al. 2012] (ElementwiseKernel), and autotuned HySoP symbolic code generator (MEASURE). The NumPy implementation is not multithreaded.

All considered `OpenCL` and `CUDA GPU` platforms perform roughly the same under enough workload. `CUDA` based solutions (`PyCuda` and `cupy`) exhibit 45% faster runtimes than their `OpenCL` counterparts (`PyOpenCl` and `HySoP`) for the consumer grade `GPU` (`GeForce RTX 2080Ti`). The generated code being roughly the same between `PyOpenCl` and `PyCuda`, this could be explained by differing kernel compilation flags between the two platforms [Fang et al. 2011]. The `Tesla V100-SXM2` achieves `774.4 GB/s` (86.0% of peak bandwidth) while the `GeForce RTX 2080Ti` achieves `263.4 GB/s` (42.8% of peak bandwidth vs. 62.1% for `CUDA`).

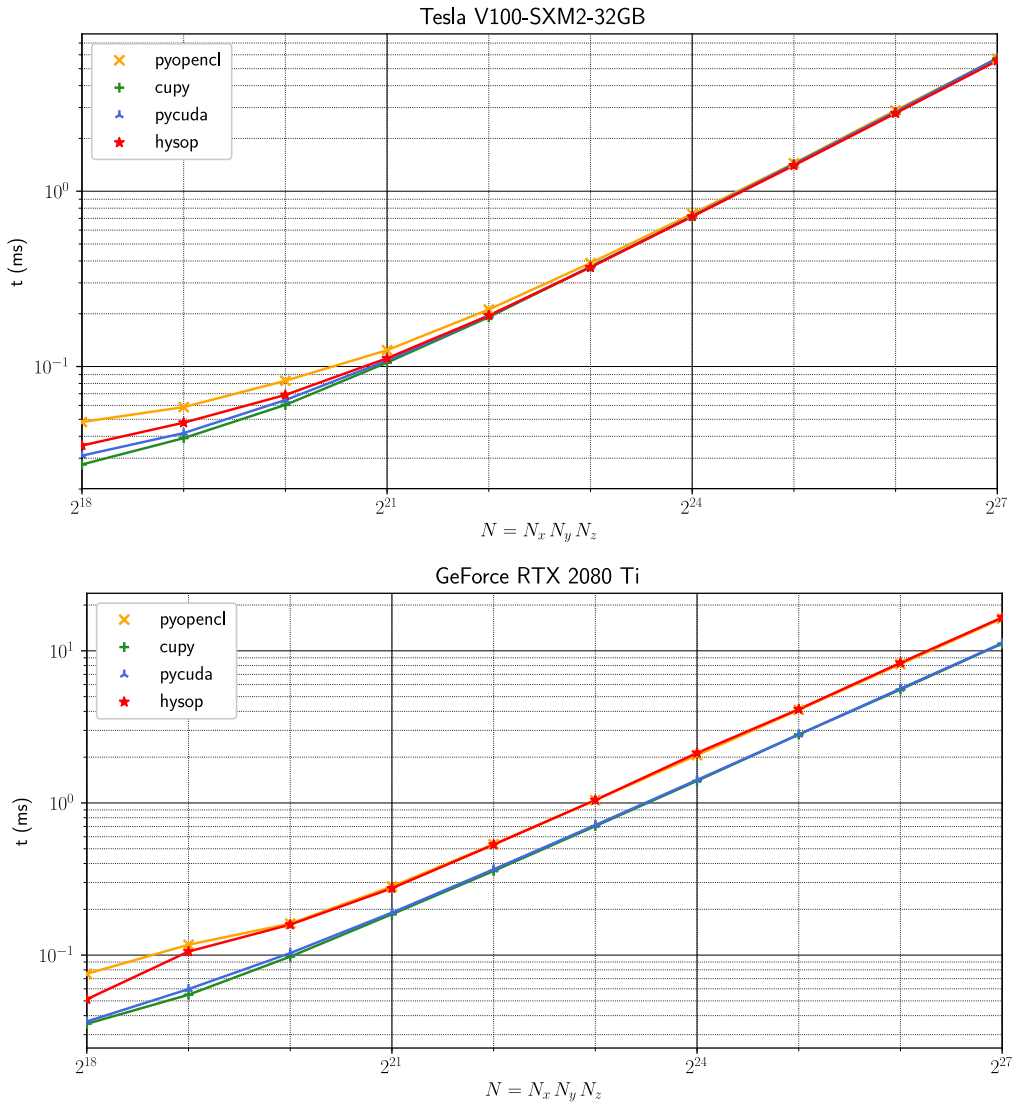


Figure 3.10 – Three dimensional array benchmark on `GPU`: Mean runtime required to compute `d=a+b*sin(c)` over 1024 runs where a, b, c, and d are double-precision arrays of shape (N_z, N_y, N_x). This computation is run on the following array backends: `PyOpenCl` and `PyCuda` [Klöckner et al. 2012] (`ElementwiseKernel`), `CuPy` [Nishino et al. 2017] and autotuned `HySoP` symbolic `OpenCL` code generator (`MEASURE`). `PyCuda` and `CuPy` are `CUDA` based libraries whereas `PyOpenCL` and `HySoP` are based on `OpenCL` (using `Nvidia OpenCL` platforms). The kernel is compute-bound on the consumer-grade `GPU`.

3.2.3 Permutations and automatic tuning of kernel runtime parameters

Performance auto-tuning is a well established technique which has been applied in a variety of high performance libraries targeting CPUs with multi-level caches, such as ATLAS for dense linear algebra [Whaley et al. 1998], OSKI for sparse linear algebra [Vuduc et al. 2005] Datta2009 for stencil based computations [Datta et al. 2009], and FFTW for fast Fourier transforms [Frigo et al. 2012]. It has been naturally suggested as a solution to the OpenCL performance portability problem [Fang et al. 2011]. Determining the best set of optimizations that have to be applied to a kernel to be executed on a given OpenCL device is a challenging problem [Dolbeau et al. 2013]. For instance, some kernel optimizations may result in excellent performance on GPUs, but reduce performance on CPUs. In the case the target device is not known in advance, the choice of the best kernel execution parameters has to rely on some runtime kernel microbenchmarking strategy. Such OpenCL kernel autotuning strategy has been successfully applied to the same application specific problems as on the CPU: stencil based computations [Datta et al. 2008], fast Fourier transforms (MPFFT [Li et al. 2013]), sparse linear algebra (clSpMV [Su et al. 2012a]), and dense linear algebra (clMagma [Cao et al. 2014]).

Numerical simulations performed with HySoP with the OpenCL backend generate tens to hundreds of different OpenCL kernels, each of them requiring a specific configuration to achieve the best performance. Moving an application to a different device often requires a new optimization configuration for each kernel [Grauer-Gray et al. 2012]. The aim of automatic kernel parameter tuning is to guaranty some performance portability by relying solely on empirical data, instead of any performance model or a priori knowledge of the target compute device [Du et al. 2012]. In section 1.4.5 we already identified at least three parameters to optimize in order to get closer to device peak memory bandwidth: local and global grid size and vectorization. The global and local grid sizes can easily be tuned at runtime without performing any modification to the code and constitute parameters of choice for quick kernel autotuning [Spafford et al. 2010]. In practice, other kernel optimization parameters are mostly application dependent [Zhang et al. 2013].

In the present work, all kernels are code generated and autotuned during problem setup and the same kernels are then called a certain number of times at every timestep. A unique kernel may be called thousands of time during a single simulation. It is thus computationally efficient to spend some time in the exploration of the kernel parameter space.

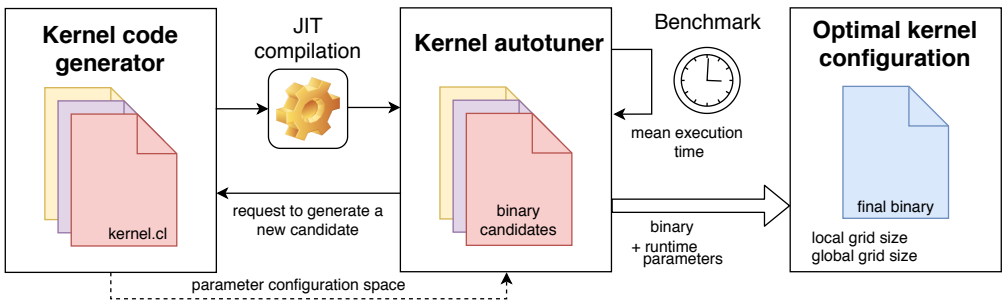


Figure 3.11 – Interactions between the kernel code generator and kernel autotuner

Many different methods exist to explore the parameter space, from simple brute-force approaches to autotuners based on efficient discrete search algorithms [Werkhoven 2019] or even on machine learning [Falch et al. 2015]. Those two last approaches are more efficient since they only explore a subset of the entire tuning parameter configuration space. Here we choose simplicity and opt for a classical brute forcing algorithm as provided by [Nugteren et al. 2015] (see figure 3.11). The autotuner is implemented directly into the `HySoP` library and provides a planning interface similar to `FFTW` to reduce the time passed during the autotuning step. It features four levels of parameter sets (enabled with the planning rigor flags `ESTIMATE`, `MEASURE`, `PATIENT` and `EXHAUSTIVE`) depending on how much time the user is willing to pay during the planning phase.

One kernel is code generated per kernel parameters sets (including hard-coded runtime parameters such as local and global grid size) and is benchmarked by averaging statistics over a given amount of successive runs. Kernels that underperform the actual best candidate over some threshold value (usually 120%) are automatically pruned out from the optimization process. The procedure then consists into keeping half of the best candidates and doubling the number of successive runs to refine bench statistics until there is only one kernel left. Once the autotuner has found an optimal set of parameters, the results are cached on disk for the actual device configuration, allowing faster subsequent runs. The case of the local permutation kernels is interesting in this context because of the large space of kernel generation parameters.

Transposition or permutations kernels are generated by using the following set of classical optimizations:

- Only coalesced memory transactions from and to device global memory.
- Use of a square window of local memory (called a tile) to handle transposition between the axis contiguous in memory (the last one) and its corresponding permutation. The array is permutated tile by tile out of place (inplace permutation can also be done easily for 2D arrays but requires two tiles in local memory).
- Minimize bank conflicts on local memory transposition by offsetting each line by one element. Shared memory usually is divided into equally-sized memory modules, called banks. They are organized such that successive words are assigned to successive banks. These banks can be accessed simultaneously, and to achieve maximum bandwidth work-items should access local memory associated with different banks
- Use of diagonal work-group reordering to minimize partition camping. As with local memory, global memory is often divided into multiple partitions that have all to be accessed simultaneously by work-items of a given work-group to achieve global maximal bandwidth. Given a work-group index, the position of tile is found by using diagonal array coordinates.

Those optimizations, described in [Ruetsch et al. 2009] and [Baqais et al. 2013], can be generalized to permutations in any dimensions. The kernel autotuner tries to find optimal

parameters such as the tile size, global and local sizes and has to enable or disable the use of diagonal coordinates and tile offsetting. The maximum tile size is determined by the device available local memory per work-group: `32kiB` for the two `CPUs` and `48kiB` for the two `GPUs`. To transpose `4B` elements (`float` or `int`), this leads to maximum tiles size of 90 and 110 (90×90 and 110×110 elements). For 2D inplace transpositions, two tiles are required and the maximal tile size drops to 64 and 78 respectively. In practice the autotuner only explore tile sizes that are powers of 2 (and of the form $2^i\,3^j$ in `EXHAUSTIVE` mode).

The `HySoP` library provides options to configure the kernel autotuner and to dump resulting kernel execution statistics. As an example, the following statistics represent all the different configurations generated for the transposition of a single-precision floating point array of shape $(16384, 16384)$:

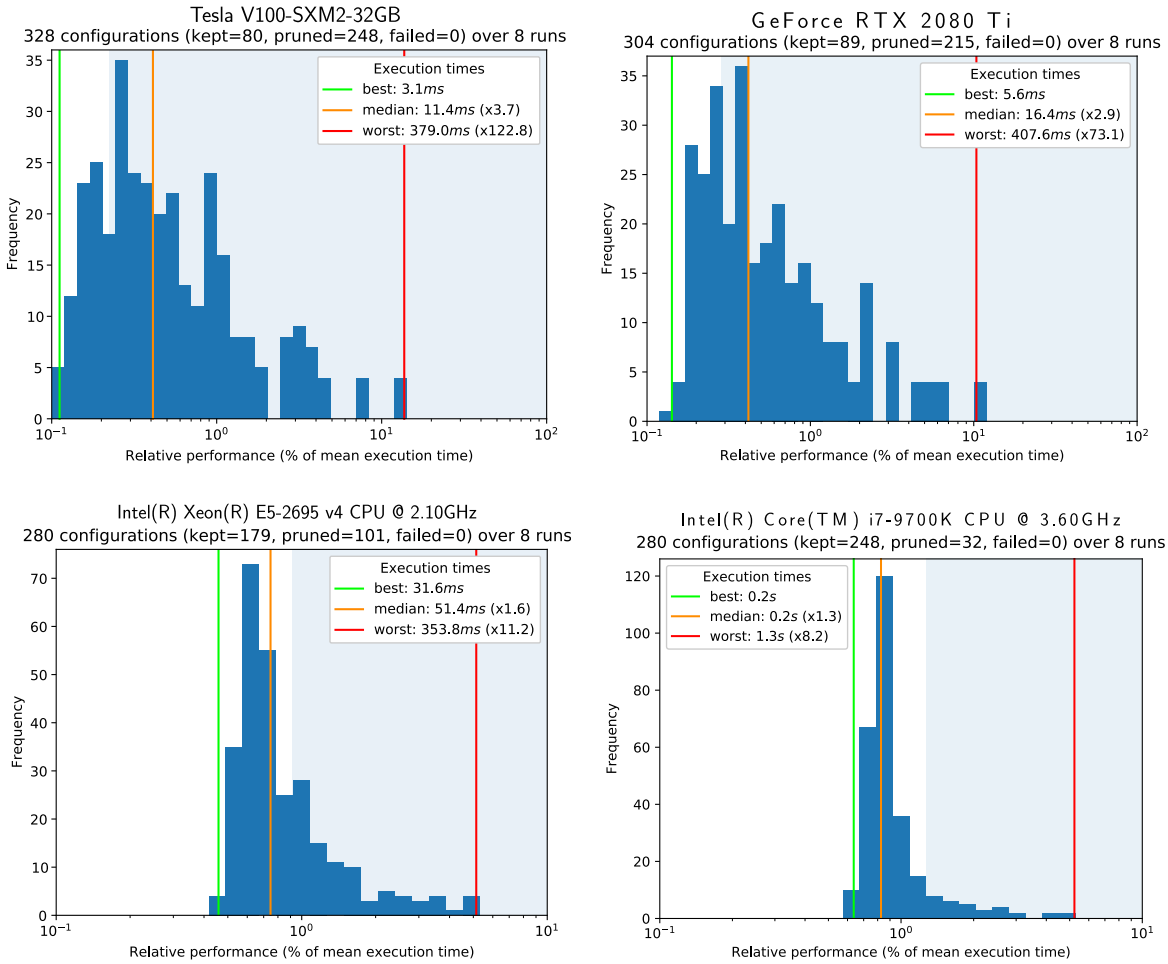


Figure 3.12 – Kernel statistics obtained for `2D` transposition of arrays of size 16384^2
The kernel autotuner generate hundreds of different kernels which mean runtimes are measured initially over 8 runs. The resulting distribution of relative runtimes (with respect to the mean mean-runtime across all benched kernels) is represented in an histogram with log scale. Sky blue overlay correspond to kernels that have been pruned out during the procedure. Green, orange and red vertical lines represent, respectively, best, median and worst runtime.

The kernel autotuner gives the following optimal parameters on the different devices for two and three-dimensional arrays of size 16384^2 and 512^3:

Device	Axes	Shape	DC	TO	TS	G	L	t (ms)	BDW (GB/s)
E5-2695v4	(2,1)	16384^2	✗	✗	32	(4096,512)	(8,1)	31.6	69.0 (50%)
E5-2695v4	(1,3,2)	512^3	✗	✗	64	(64,64,512)	(8,8,1)	20.5	52.4 (38%)
E5-2695v4	(2,1,3)	512^3	✗	✓	64	(2,512,512)	(1,1,1)	17.7	60.7 (44%)
E5-2695v4	(3,1,2)	512^3	✗	✓	32	(16,16,512)	(1,1,1)	23.3	46.1 (34%)
E5-2695v4	(2,3,1)	512^3	✗	✓	64	(8,512,8)	(1,1,1)	24.0	44.7 (33%)
E5-2695v4	(3,2,1)	512^3	✗	✓	64	(8,512,8)	(1,1,1)	22.0	48.8 (36%)
i7-9700K	(2,1)	16384^2	✗	✗	64	(8192,256)	(32,1)	152.6	14.1 (66%)
i7-9700K	(1,3,2)	512^3	✗	✗	64	(64,16,512)	(8,2,1)	84.6	12.7 (59%)
i7-9700K	(2,1,3)	512^3	✗	✓	64	(64,512,512)	(16,1,1)	90.6	11.9 (56%)
i7-9700K	(3,1,2)	512^3	✗	✗	64	(64,16,512)	(8,2,1)	99.6	10.8 (48%)
i7-9700K	(2,3,1)	512^3	✗	✗	64	(64,512,8)	(8,1,1)	100.9	10.6 (48%)
i7-9700K	(3,2,1)	512^3	✗	✗	64	(64,512,8)	(8,1,1)	89.8	12.0 (54%)
V100-SXM2	(2,1)	16384^2	✗	✓	64	(16384,4096)	(64,16)	3.1	693 (77%)
V100-SXM2	(1,3,2)	512^3	✗	✓	64	(512,64,512)	(64,8,1)	1.4	767 (85%)
V100-SXM2	(2,1,3)	512^3	✗	✓	32	(128,512,512)	(64,1,1)	1.4	767 (85%)
V100-SXM2	(3,1,2)	512^3	✗	✓	32	(256,256,512)	(16,16,1)	1.4	767 (85%)
V100-SXM2	(2,3,1)	512^3	✓	✓	32	(512,512,16)	(32,1,1)	2.2	488 (54%)
V100-SXM2	(3,2,1)	512^3	✗	✓	32	(512,512,16)	(32,1,1)	2.2	488 (54%)
RTX 2080Ti	(2,1)	16384^2	✗	✓	32	(16384,4096)	(32,8)	5.6	385 (62%)
RTX 2080Ti	(1,3,2)	512^3	✗	✓	32	(512,64,512)	(32,4,1)	2.0	537 (87%)
RTX 2080Ti	(2,1,3)	512^3	✗	✗	32	(128,512,512)	(128,1,1)	2.0	537 (87%)
RTX 2080Ti	(3,1,2)	512^3	✗	✓	16	(512,256,512)	(16,8,1)	2.0	537 (87%)
RTX 2080Ti	(2,3,1)	512^3	✗	✗	32	(512,512,16)	(32,1,1)	2.9	370 (60%)
RTX 2080Ti	(3,2,1)	512^3	✗	✗	32	(512,512,16)	(32,1,1)	3.0	358 (58%)

Table 3.2 – Kernel autotuning results for 2D and 3D OpenCL permutation kernels: Best configuration determined by the kernel autotuner in PATIENT mode for two and three-dimensional single-precision floating point arrays of given shape. Mean runtime value is obtained by at least 128 runs of the best resulting kernel. Axes corresponds to the state of axes after permutation starting from unpermutated configuration (1,2) and (1,2,3) where the first axis (resp. the last axis) has the greatest (resp. smallest) stride in memory. DC corresponds to diagonal coordinates, TO to tile offsetting, TS to tile size, BDW to achieved global memory bandwidth and G and L to global and local grid sizes. Achieved global memory bandwidth is also given in percentage of theoretical peak memory bandwidth of the target device.

Apart from the dual CPU configuration, every possible permutations for two and three-dimensional arrays achieve more than 50% of the device theoretical peak global memory bandwidth. On the two GPUs , some permutations even achieve more than 85% of the maximal theoretical value. Diagonal coordinates is an optimization that could be disabled for every benched platforms whereas tile offsetting seems to help in half the cases. Those performances are in agreement with dedicated multidimensional array permutation libraries targeting CUDA such as TTC (Tensor Transposition Compiler [Springer et al. 2016], cuTT (CUDATensor Transpose library [Hynninen et al. 2017] and TTLG (Tensor Transposition Library for GPUs [Vedurada et al. 2018]). To the author's knowledge there exist no such dedicated dense tensor permutation library for OpenCL. Performance on CPUs is also comparable to HPTT (High Performance Tensor Transposition, [Springer et al. 2017]).

The `Intel OpenCL` platform seems unable to surpass half the bandwidth in a multi-socket configuration (dual `Xeon E5-2695v4` with four memory channels per `CPU`). This behaviour was already seen on `clpeak` and `elementwise` benchmarks, and could be the result of the memory policy that is in place for the allocation of the `CPU OpenCL` buffer on the underlying NUMA architecture [Lameter 2013]. We recall that a NUMA node is defined as an area where all memory has the same speed as seen from a particular `CPU` (a node can contain multiple `CPUs`). We study the effect of the memory policy on a quad socket `Intel Xeon E7-8860 v4` platform with a theoretical peak global memory bandwidth of `51.2 GB/s` per node (one node containing one `CPU`). The kernel considered here is the 2D transposition of single-precision floating point arrays of size 16384^2. To enforce memory policies we use the `NUMA` library (through its `Python` binding of the same name) and the `numactl` utility [Kleen 2005].

A first batch of statistics is collected with the `numactl` utility:

- Only one `CPU` with local allocation (`numactl --cpunodebind=0 --membind=0`) yields 80.7ms mean execution time, or `26.6GB/s` (52.0% of the peak memory bandwidth of one memory node).
- All `CPUs` with allocation on `NUMA` node 0 (`numactl --cpunodebind=0,1,2,3 --membind=0`) yields 83.5ms mean execution time, or `25.7GB/s` (50.0% of the peak memory bandwidth of one memory node).
- Only one `CPU` with local allocation interleaved on all nodes (`numactl --cpunodebind=0 --interleave=0,1,2,3`) yields 62.3ms mean execution time, or `34.5GB/s` (16.8% of the peak memory bandwidth of all memory nodes).
- All `CPUs` with local allocation (`numactl --cpunodebind=0,1,2,3 --localalloc`) yields 28.8ms mean execution time, or `74.5B/s` (36% of the peak memory bandwidth of all memory nodes).

A second batch of statistics is gathered directly in `OpenCL` by using device fission capabilities offered by the `OpenCL 1.2` standard [Sych 2013] and `Intel` specific extension `cl_intel_device_partition_by_names` [Yariv et al. 2013]. We create four non-overlapping sub-devices containing specific compute units based on `libnuma.node_to_cpu`, splitting the `OpenCL` device into the four subdevices corresponding to the four physical `CPUs`. The best kernel obtained with the command `numactl --cpunodebind=0 --membind=0` is recovered and compiled for every of those subdevices. One array buffer is allocated on each NUMA node and each device is benched on all those four buffers by using a subdevice-specific queue.

		Node			
		0	**1**	**2**	**3**
Node	**0**	10	21	21	21
	1	21	10	21	21
	2	21	21	10	21
	3	21	21	21	10

(a) Node distance as obtained with `numactl`

		OpenCL Subdevice			
		0	**1**	**2**	**3**
Buffer	**0**	81.5	124.5	123.6	124.6
	1	125.2	81.4	124.1	124.6
	2	124.7	124.3	81.2	124.3
	3	125.2	124.3	124.2	81.3

(b) Mean runtime for 2D transposition (ms)

Table 3.3 – Performance of transposition on a quad socket NUMA architecture

The joint use of the four subdevices results in mean runtimes of $27.4ms$ which is similar with the $28.8ms$ result obtained with `numactl --cpunodebind=0,1,2,3 --localalloc`. Enforcing memory policies with `numactl` or by manual device fission does not seem to help much in this context, the maximal obtained mean relative bandwidth being of the order of 37% (and 50% for a single socket). As a point of comparison, the `STREAM` memory bandwidth benchmark [Bergstrom 2011] achieves up to 54% of peak memory bandwidth for its best run.

3.2.4 Directional advection operator

In this subsection we study the performance of the directional advection operator. As seen in chapter 2, the transport of a scalar θ in a velocity field $\boldsymbol{u}$ is decomposed into n (or $2n$) substeps with directional splitting, where n is the dimension of the problem. Each substep is further decomposed as transport, remeshing and local permutation of data such that the next advected axis becomes contiguous in memory. At a first sight, this results into enqueuing $3n$ kernels for a first order or $6n$ kernels for second order Strang splitting, but we also have to take into account ghost exchanges and ghost accumulation (even for serial processes).

Given a timestep dt that fulfills a LCFL condition (2.23), a user chosen CFL condition (2.22) and a direction i, directional advection consists into the following steps:

1. Interpolate velocity to the scalar grid by using an interpolation scheme (only if the two grids have different resolutions).
2. Create particles aligned with the underlying scalar field grid and perform particle advection at fixed velocity $u_i(\boldsymbol{x})$ by using an explicit Runge-Kutta time integrator 2.3.2. Particles may be transported up to $C \geq dt \left\|u_i\right\|_\infty / dx_i$ cells outside of the domain (where C is obtained from the arbitrary CFL condition). The required ghosts for the velocity and input scalar is thus $1 + \lfloor C \rfloor$.
3. Remesh all the particles with algorithm 5 by using a remeshing kernel $\Lambda_{p,r}$ that has support $[-s, s]$ with $s = 1 + p/2$. This requires $1 + \lfloor C \rfloor + s$ ghosts to remesh the output scalar field.
4. Accumulate ghost remeshed quantities to the left and to the right processes (with respect to the current direction). For a serial process with periodic boundary conditions in the current direction, this will accumulate left ghost values to rightmost compute grid values.
5. Update ghosts by performing a full ghost exchange.
6. Perform a local permutation such that the next axis becomes contiguous in memory. In 2D, this corresponds to a transposition but in 3D multiple different permutations are available.

Within a multi-grid context where the scalar is discretized on a finer grid, the number of scalar ghosts is multiplied by the grid ratio in the considered direction. The user chosen CFL number should be such that ghost exchange has a reasonable cost (see section 3.3.2).

The `OpenCL` implementation in mainly based on the one of [Etancelin 2014] and has been fit into the code generation and auto-tuning framework. Each line that is contiguous in memory represent and independent one-dimensional transport problem that is treated iteratively by one work-group (by iterating on sublines which length is a multiple of the work-group size). During the advection step, the velocity is only read once and is cached in local memory to allow for efficient linear interpolations. During the remeshing step, the particles are remeshed to local memory before being written back to global memory. In the case local memory do not improve performance, this caching behaviour can be disabled by the autotuner.

Advection kernel

The arithmetic intensity of the advection kernel only depends on the Runge-Kutta scheme (the Butcher tableau), and the chosen interpolation scheme. In practice we only use linear interpolation during the advection step (a more elaborate interpolation method may be used to interpolate velocity to the fine grid as a pre-processing step). For fields discretized on $\boldsymbol{\mathcal{N}}^v$ points and a Runge-Kutta scheme of s steps, the advection step performs s evaluations of the velocity per particle along with $2s + k - 1$ additions and $2s + k$ multiplications per particles where $s - 1 \leq k \leq s(s+1)/2$ represent the number of non-zero coefficients a_{ij} contained in the Butcher tableau (see section 2.3.2). Evaluation of velocity has no additional cost on the first stage (particles are grid aligned) and for all subsequent stages it cost approximately $\simeq$ 5 `FLOP / particle`, see equation (2.27)). The computation of the initial particle position requires one multiplication per particle. The total number of particles is $N = \prod_{i=1}^{n} \mathcal{N}_i^v$ and if we consider all operations we obtain a total of $(4s + 2k + 5s)N$ operations. The total number of memory transaction in terms of elements is N reads of velocity and N writes of resulting particle position ($2N$ memory transactions). If we denote S the element size in bytes this gives a total arithmetic intensity of $(9s + 2k)/(2S)$ `FLOP/B`. The following table shows the usual cases of single- and double-precision Euler, RK2, RK3, RK4 and RK4_38 Runge-Kutta methods:

	s	k	FLOP/E	FLOP/B (SP)	FLOP/B (DP)
Euler	1	0	4.5	1.1	0.6
RK2	2	1	10	2.5	1.3
RK3	3	2	15.5	3.9	1.9
RK4	4	3	21	5.3	2.6
RK4_38	4	5	23	5.8	2.9

Table 3.4 – Approximative arithmetic intensity of the advection kernels

In practice, the arithmetic intensity also depends on vectorization and the size of input at constant CFL number. The total memory that has to be read include velocity ghosts and is thus larger than N. The vectorization also plays a role because some particle independent scalar quantities can be computed once for multiple particles at a time.

Because computing arithmetic intensity by hand is a tricky task, even for simple algorithms, we rely on an automatic tool to collect kernel execution statistics from code generated kernels.

This tool is based on Oclgrind, a virtual OpenCL device simulator which goal is to provide a platform for creating tools to help OpenCL development [Price et al. 2015]. The simulator is built on an interpreter for the LLVM IR (LLVM intermediate representation [Lattner et al. 2004]) and provide a lightweight method to run kernels in isolation from any OpenCL host code. This interface is provided via the oclgrind-kernel command, which takes as input a simulator file (*.sim) describing how to run the kernel (location of the OpenCL source code, configuration of kernel arguments, and global and local sizes). Perhaps the most interesting feature of Oclgrind is that it provides a simple plugin interface that allows third-party developers to extend its functionality. This interface allows a plugin to be notified when various events occur within the simulated kernel, such as an instruction being executed or memory being accessed. Plugins are implemented in C++ by extending the virtual oclgrind::Plugin class and have to be compiled as external dynamically loadable libraries.

As most of HySoP kernels are code generated such kernel simulation files can easily be generated by the autotuner that has the full knowledge of the source code and associated kernel arguments. By adding a simple callback support, the autotuner can run arbitrary user scripts with the knowledge of the source location, simulation file and runtime statistics. Such a script can run oclgrind-kernel with a custom plugin and analyze its outputs. The present work use this approach with a custom plugin that computes instructions statistics occurring during the simulation of some target kernel. The plugin collects data about integer and floating point operations as well as memory transactions (private/local/constant/global) and outputs them to disk for further analysis. An example of statistics obtained with this plugin is given appendix (C). The user script can then proceed to compute the effective floating point operations per byte FLOP/B and integer operations per byte IOP/B by using global memory data (all target kernels only load and store a given piece of global data once). The mean runtime being provided by the autotuner, it is then possible to compute mean achieved bandwidth and mean achieved compute statistics.

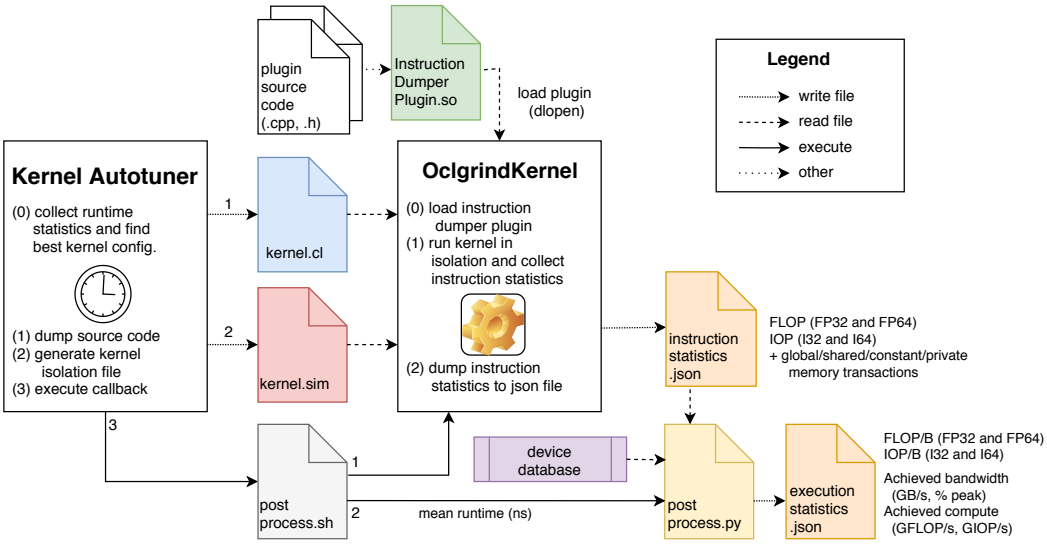


Figure 3.13 – Overview of the process to gather kernel execution statistics

We use this method to collect statistics of directional advection on the `Tesla V100-SXM2 GPU` for various Runge-Kutta schemes and grid resolutions. Full results, including execution times are available in tables C.1 and C.2 and corresponding rooflines are plotted here:

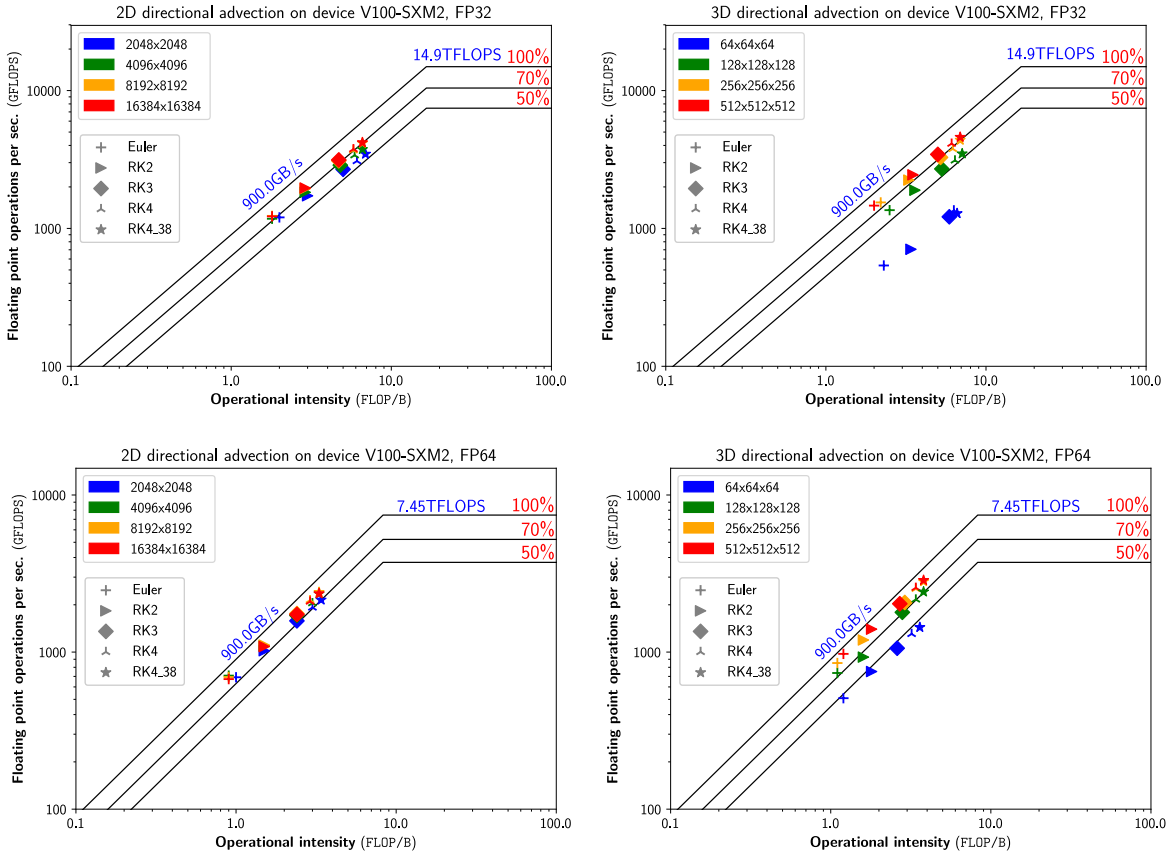


Figure 3.14 – Roofline of 2D and 3D, single- and double-precision advection

From figure 3.14 it is clear that advection is a memory-bound problem on the `Tesla GPU` and that arithmetic intensity increases with integration order in time. Achieved global memory bandwidth increases with the size of the array. This is expected because one work-group is scheduled per line (particles are all independent) and device occupancy increases when more work-groups are scheduled at a time. The achieved bandwidth also depends on the size of the work-groups (number of work items) and associated workload. Although the smallest 3D problem spawns twice the number of work-groups when compared to the smallest 2D problem (4096 vs. 2048), the 2D problem achieves much higher bandwidth simply because the smallest 2D problem is 16 times bigger than the smallest 3D problem (2048^2 vs. 64^3). Under enough workload the directional advection kernels achieve at least 70% of global peak theoretical memory bandwidth for single-precision and 80% for double-precision, the maximal achieved bandwidth being of 86.9%. Thanks to the massive memory bandwidth provided by the second generation of High Bandwidth Memory (`HBM2`) stacked onto the device, it takes only 1.62 milliseconds to transport 512^3 ($\simeq$ 134.2 millions) particles by using single-precision. Put the other way around, this represent one particle every 12 picoseconds or 83 giga-particles per second. Once all particles have been spawned and advected, the remeshing kernel is responsible to remesh those free particles onto the grid.

Remeshing kernel

The main problem with remeshing is that consecutive particles can be accumulated at the same time to the same local memory locations by neighboring work-items. The historical `OpenCL` implementation of the library is based on the assumption that because of the regularity of the velocity (imposed by the LCFL), the maximal number of particles per cell is kept small after the advection step. Under this hypothesis, local memory data races can be avoided by enforcing a given work-item to treat a certain amount of consecutive neighboring particles (usually 2, 4, 8 or 16 to fit `OpenCL` vector builtins). In practice this implementation has shown good results on many different test cases but it tends to reduce device occupancy (reduced work-group sizes). A new implementation of the remeshing kernel has been introduced by using atomic additions in local memory. Unfortunately most devices do not provide `64bit` atomic intrinsics so this second implementation may only be available for single-precision floating point numbers on certain devices such as `NVidia GPUs`. Atomic operations on floating point values are usually not readily available and are thus implemented with the usual `union + atomic_cmpxchg` workaround [Hamuraru 2016]. Apart from these parameters, the autotuner has to find the optimal vectorization and workload per work-item. As before, results are given for two- and three-dimensional, single- and double-precision arrays in tables C.3 and C.4 and the corresponding rooflines are depicted here:

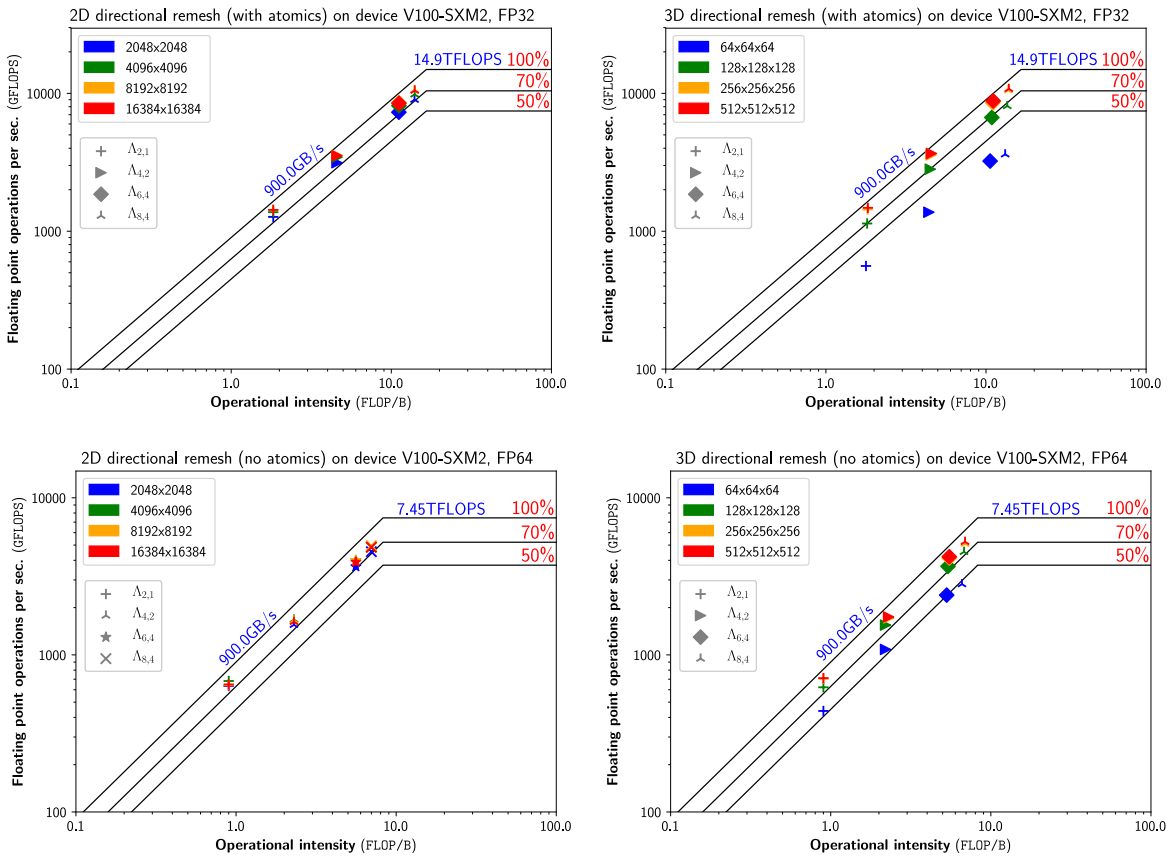


Figure 3.15 – Roofline of 2D and 3D, single- and double-precision remeshing

The remeshing formula $\Lambda_{p,r}$ is a piecewise polynomial of a certain order that is evaluated with Horner's method. High order remeshing formulas that preserve multiple moments with decent overall regularity tend to be compute-bound but multiple scalars can be advected at a time to save on expensive calls to the remeshing formula. Because additional scalars decrease arithmetic intensity, we only consider the best case scenario (in terms memory bandwidth) of a single advected scalar. As it can be seen on figure 3.15, for a single scalar the `Tesla V100 GPU` and the $\Lambda_{8,4}$ remeshing formula, we obtain an algorithm that is at the same time compute and memory bound (the sweet spot of designed arithmetic intensity). Out of the 32 single-precision problem configurations, the kernel autotuner has chosen the implementation based on atomic operations 30 times. For all those cases the preferred vectorization parameter alternates between 1 and 2 and under high load the achieved relative global memory bandwidth lies between 83.1% and 90.0%. As there is no double-precision atomic support on this device, all the double-precision results arrise from the version without atomic intrinsics (where the preferred vectorization becomes 2). For those cases, the achieved relative bandwidth is between 77.0% and 86.6%. The $\Lambda_{8,4}$ remeshing kernel exceeds `10 TFLOPS` of compute for the single-precision (`10779 GFLOPS` or 72.3% of the device capabilities) and about half of it for the double-precision version. For all cases the achieved performance is roughly 16 picoseconds per remeshed particles or 64 giga-particles per second (single-precision).

Overall n-dimensional transport performance

The n-dimensional transport of a scalar is obtained by chaining n advection and remeshing operators interleaved with n permutations of the scalar. The graph builder automatically determines that each of the velocity components have to be in permutated state so that velocity data is permutated only once at problem initialization (in fact the velocity components are just initialized in permutated state). This gives a total of $3n$ operators for first order Strang splitting and this number doubles for second order Strang splitting. As only two-dimensional square arrays support inplace transpositions, inplace two-dimensional rectangle and three-dimensional permutations are performed out-of-place to a temporary buffer, followed by a device-to-device copy to the source array. This decreases the permutation performance by a factor of two (transposition is memory bound). Those $3n$ operators map to $3n$ `OpenCL` kernels plus n device-to-device copies (except for 2D square discretizations), plus additional copy and accumulation of hyperplane layers required for ghost exchange and ghost accumulation. As some `OpenCL` platforms do not correctly support device-to-device rectangular copies from and to non-overlapping subregions of the same buffer, ghost exchange and ghost accumulation are done inefficiently by using temporary preallocated device buffers and by performing twice the required number of copies. A ghost exchange costs 4 copies per direction and a ghost accumulation 6 copies and two accumulation kernels. Each directional advection operation require a directional ghost accumulation and a full ghost exchange. For advection, this brings the total number of ghost related copy kernels to $d(4d + 6)$. Besides paying extra kernel launch latency, the device-to-device copies of discontinuous chunks of data perform poorly. The solution to this problem would be to implement custom ghost exchange and ghost accumulation kernels. This would however not improve the distributed implementation of the library because device-to-device copies of ghosts layers have to be replaced with device-to-host and host-to-device copies from and to host buffers (to perform `MPI` exchanges).

Figure 3.16 show single-precision kernel execution statistics gathered on 100 iterations of n-dimensional advection on the `NVidia GeForce RTX 2080Ti GPU` for various configurations of time integrators and remeshing kernels:

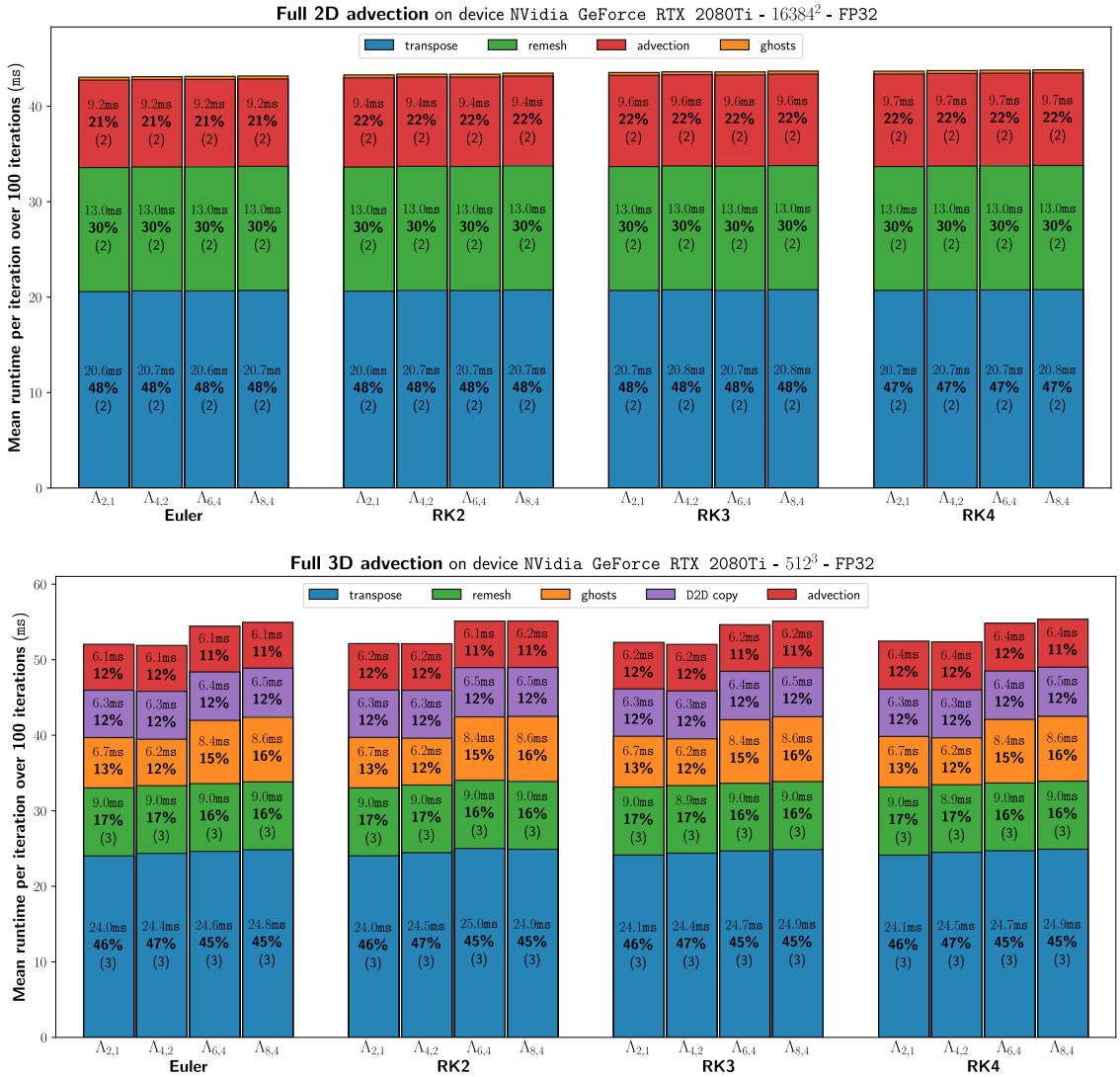


Figure 3.16 – Distribution of runtime for n-dimensional single-precision transport Mean runtime and runtime ratio is given for different kind of kernels. Each bar corresponds to a specific configuration (Runge-Kutta scheme and remeshing formula). `D2D` copy corresponds to copies required for permutations and `ghosts` to ghost exchange and accumulation. The number between parenthesis, when given, corresponds to the number of calls per iteration.

From figure 3.16 it is clear that the whole advection problem is memory bound on the consumer-grade `GPU`, as it was the case for the server-grade `GPU` in the previous roofline analyses. In 2D, every numerical method results into the same runtime. In 3D the situation is the same but the runtime is increased by roughly 5% due to increased number of ghosts required by higher order remeshing formula $\Lambda_{p,r}$ (the support of the remeshing kernel increases with the number of even preserved moments p).

Due to the directional splitting approach and differences in the number of copies between the 2D and 3D cases the overall achieved performance for transport depends on the number of dimensions. On this device, a particle can be transported every 160 picoseconds on a 2D domain (6.3 giga-particles per second) and every 410 picoseconds in 3D (2.44 giga-particles per second) regardless of the order of the methods. Permutations constitute most of the runtime (45 to 48%) and further optimization of the transposition kernels could lead up to 30% improvement in total runtime. We saw that different permutations perform differently and at this time the operator graph builder does not perform microbenchmarking to determine the best permutation axes. Another improvement track would be to use one of the `CUDA` transposition library and to convert the generated `CUDA` code to `OpenCL` (or more elegantly to implement `OpenCL` code generation in one of those frameworks). Proper `CUDA` support also constitute a long-term goal of the `HySoP` library.

The same benchmark run on the dual `CPU` device reveals that transport is compute-bound mainly by the remeshing step (figure 3.17). On this device the advection step takes longer with increasing time integration order (Runge-Kutta schemes). Remeshing also have bigger runtimes for remeshing formulas with increasing number of preserved moments and take up to 59% of the runtime for the $\Lambda_{8,4}$ formula. Achieved performance range from 1099 to 2127 picoseconds per particle (0.5 to 0.9 giga-particles per second). Two-dimensional single-precision performance is roughly ten times worse than the consumer `GPU` for approximately a factor four in price tag. For double-precision computations this performance gap reduces (transport is compute-bound in double-precision on consumer `GPUs` due to architecture design).

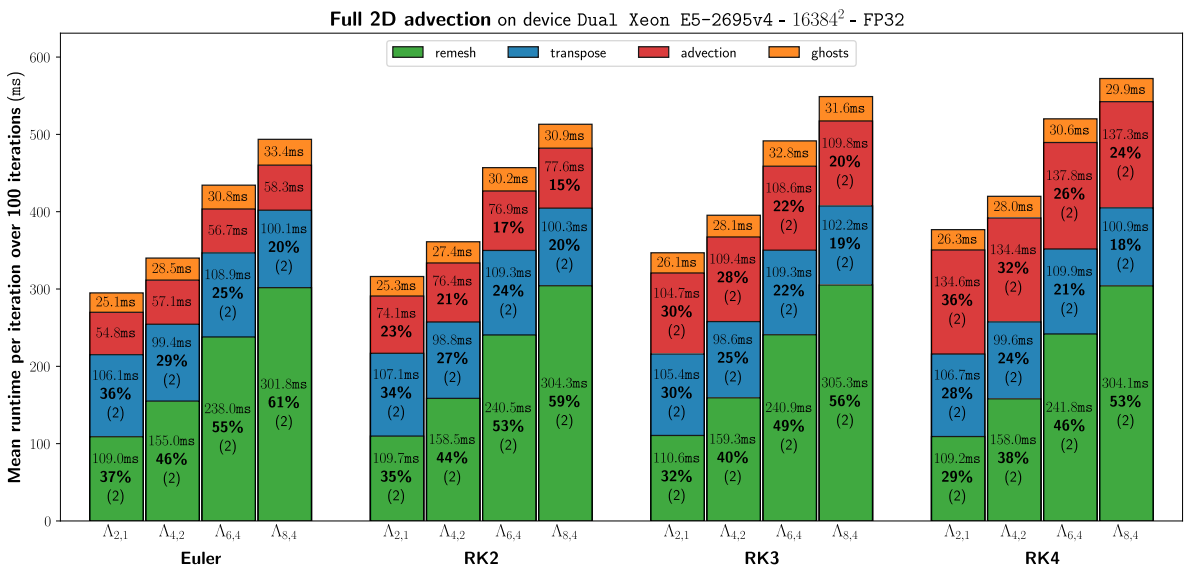


Figure 3.17 – Distribution of runtime for single-precision transport on the `CPU`
Mean runtime and runtime ratio is given for different kind of kernels. Each bar corresponds to a configuration consisting in a specific Runge-Kutta scheme that affects advection and a specific remeshing formula $\Lambda_{p,r}$ that is used during the remeshing step. `D2D` copy corresponds to copies required for transpositions and `ghosts` to ghost exchange and accumulation. The number between parenthesis, when given, corresponds to the number of calls per iteration.

3.2.5 Finite differences operators

Finite difference operators such as diffusion and stretching are all implemented by using the symbolic code generation framework. Given a symbolic expression containing n-dimensional scalar fields and spatial derivatives of scalar fields (but no mixed spatial derivatives) the expression is automatically split into n expressions corresponding to what we did manually in section 2.5. This is done by simple depth-first `Sympy` expression `AST` traversal. Once the directional symbolic expressions are found, spatial symbolic variables are permutated so that all memory accesses become coalesced. As for advection and remeshing, it will be the graph builder's responsibility to inject permutations operators prior to each directional operator. The symbolic code generator then generates kernels corresponding to those expressions by mapping the symbolic representation to `OpenCL`. Each expression requires multiple analysis and transformation steps until the final code is generated.

Because expressions are directional, we always fall in the case of independent 1D problems. As before, a work-group is responsible to handle one contiguous line by iterating by sublines. All global memory loads and stores happen only once, and data is cached in local memory in order not to be memory-bound. Symbolic expressions are vectorized at code generation depending on a vectorization parameter. As for all other code generated kernels, the autotuner tries different combinations of parameters to find the best kernel candidate. The whole process is depicted on figure 3.18. The customization of the numerical method is done by passing method arguments such as the type of time integrator or the order of centered finite differences that have to be used to approximate the derivatives. The number of required ghosts per scalar field and per direction is extracted from the symbolic expressions and the knowledge of the methods. Because multiple symbolic expressions can be supplied to integrate multiple scalar fields at once with right hand sides that depend on all those fields and their spatial derivatives (like in equation 2.44) this analysis is not straightforward.

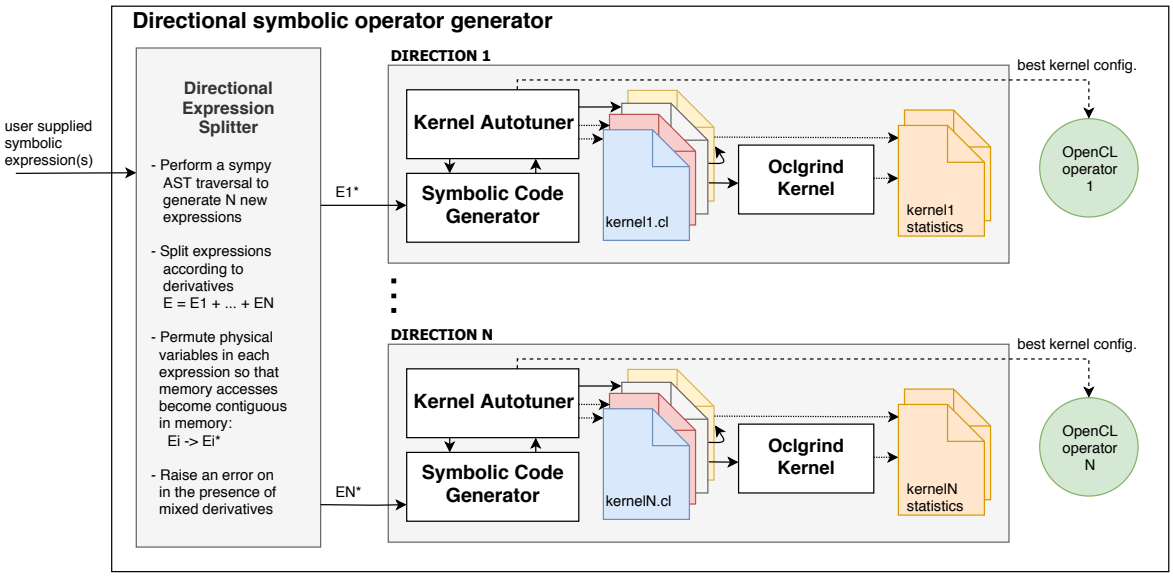


Figure 3.18 – Overview of the directional symbolic code generation process

At this time, symbolic code generation only work for input and output scalar fields discretized on the same Cartesian grid (but with varying number of ghosts). This means that the symbolic code generator do not handle on the fly interpolation of fields defined on coarser grids. Some frequently used operators such as finite differences based diffusion and stretching are directly provided by the library. Withing this framework, a user can easily generate custom `OpenCL` finite difference operators. As an example diffusion and conservative stretching can be generated by building the following symbolic expressions:

```
import numpy as np
from hysop import Box
from hysop.defaults import VelocityField, VorticityField, \
                           TimeParameters, ViscosityParameter
from hysop.symbolic.field import Assignment, div, laplacian

box    = Box(dim=3)
velo   = VelocityField(box)
vorti  = VorticityField(velo)
t, dt  = TimeParameters()
nu     = ViscosityParameter(1e-3)

# Get associated symbolic variables
xs, ts = box.s, t.s
Us, Ws, nus = velocity.s, vorticity.s, nu.s

# Vorticity diffusion (three scalar expressions)
lhs = Ss.diff(ts)
rhs = nus*laplacian(Ws, xs)
print Assignment.assign(lhs, rhs)

# Conservative stretching (three scalar expressions)
lhs = Ws.diff(ts)
rhs = div(np.outer(Us, Ws).freeze(), xs)
print Assignment.assign(lhs, rhs)
```

This will print the following symbolic expressions:

```
(∂ω₀/∂t = ν*(∂²ω₀/∂x₀² + ∂²ω₀/∂x₁² + ∂²ω₀/∂x₂²))
(∂ω₁/∂t = ν*(∂²ω₁/∂x₀² + ∂²ω₁/∂x₁² + ∂²ω₁/∂x₂²))
(∂ω₂/∂t = ν*(∂²ω₂/∂x₀² + ∂²ω₂/∂x₁² + ∂²ω₂/∂x₂²))
```

and

```
(∂ω₀/∂t = ∂[υ₀*ω₀]/∂x₀ + ∂[υ₀*ω₁]/∂x₁ + ∂[υ₀*ω₂]/∂x₂)
(∂ω₁/∂t = ∂[υ₁*ω₀]/∂x₀ + ∂[υ₁*ω₁]/∂x₁ + ∂[υ₁*ω₂]/∂x₂)
(∂ω₂/∂t = ∂[υ₂*ω₀]/∂x₀ + ∂[υ₂*ω₁]/∂x₁ + ∂[υ₂*ω₂]/∂x₂)
```

To save on memory bandwidth, stretching and diffusion kernels can be merged together by just summing the two right hand sides together. Note that here all directional expressions will be equivalent and generate exactly the same kernels. Autotuning will only happen once.

Kernels statistics are collected for 3D single- and double-precision directional operators on 512^3 fields on the `Nvidia Tesla V100-SXM2 GPU`. Diffusion and stretching results are available in table C.5 and C.6. The corresponding rooflines for the stretching operator are depicted here:

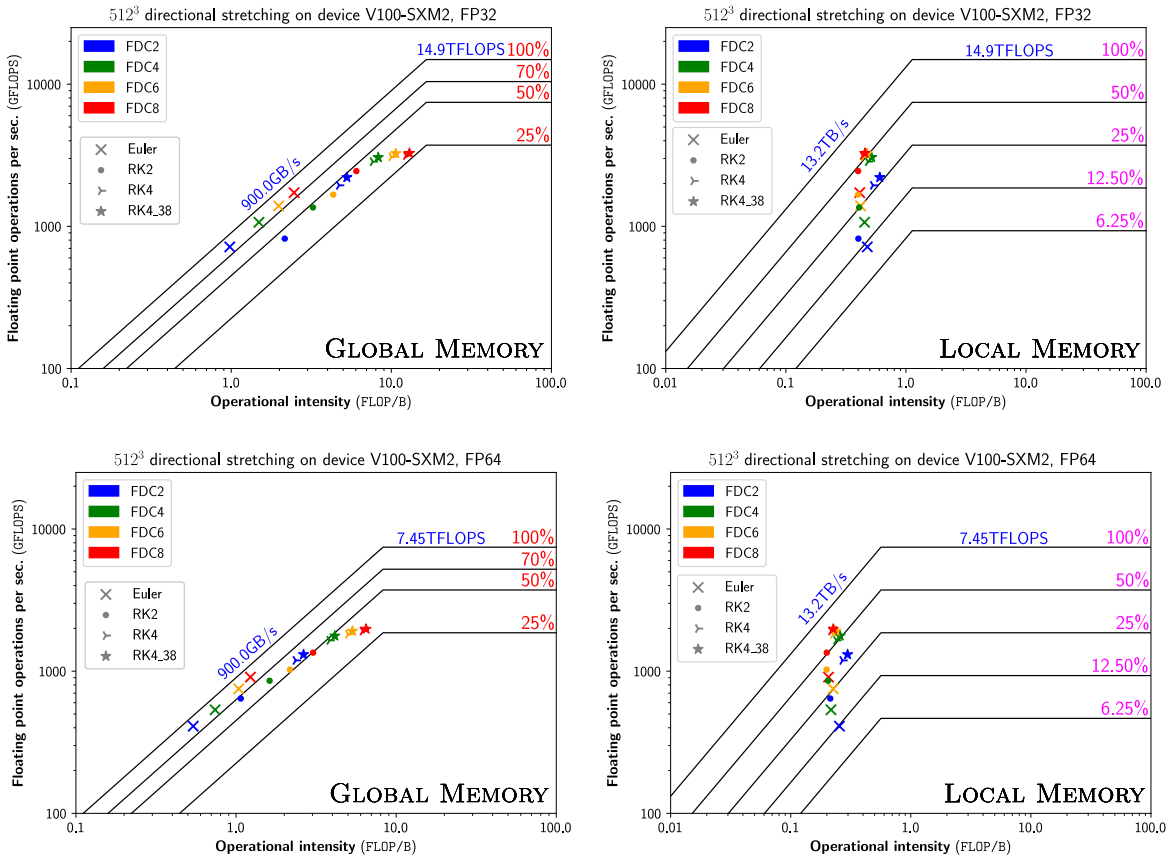


Figure 3.19 – Rooflines of single- and double-precision directional stretching

Those kernels are memory-bound and achieved relative global memory bandwidth lies between 26 and 84% depending on the number of stages of the Runge-Kutta time integrator and finite difference order (`FDCx`). The theoretical local memory bandwidth can be computed from [Jia et al. 2018] for the `Volta GPU` architecture which leads to `13.2TB/s` at device base clock (`1290MHz`). Achieved local memory bandwidth goes as high as `9.0TB/s` for the `RK4-FDC8` configuration (68% of device capabilities). As all memory accesses are coalesced and without bank conflicts, it is unclear why higher local memory bandwidth cannot be achieved, and the lack of proper `OpenCL` profiling tools for `NVidia GPUs` does not help. Without further analysis it is assumed that those performance results are linked to increasing pressure on local memory coupled to local memory barriers (one barrier per time integrator stage) and possible memory alignment issues. A local memory barrier is a work-group level construct that will flush any variables to local memory. It is required between integrator stages so that local memory transactions are visible to all work-items in given work-group. The alternative would be to call the kernel s times where s is the number of stages. This would decrease the overall performance of the operator because of additional global memory transactions.

3.2.6 Spectral transforms

The last kind of operators that we need to be able to solve an incompressible Navier-Stokes problem are spectral operators. This include the recovery of velocity from vorticity (Poisson problem with computation of curl, see algorithm 9) and spectral diffusion (algorithm 7). As we saw in section 2.6.4, the boundary conditions directly determine what kind of solver to use, the simplest configuration being a fully periodic domain, in which case only forward and backward discrete Fourier transforms are required. All kind of homogeneous boundary conditions can be solved by using real-to-real transforms (DCT and DST variants). General boundary conditions are solved spectrally by using the most suitable representation in Chebyshev polynomials (with Tau or Galerkin methods). In this subsection we do not consider the distributed case.

The general idea is to build a n-dimensional transform by tensor product of one-dimensional transforms, interleaved with data permutations for exactly the same memory coalescing reasons that justified directional operator splitting. Additionally we impose a forward transform ordering for efficiency: first real-to-real transforms (homogeneous BCs), followed by a real-to-complex transform (first periodic BC), followed by complex-to-complex transforms (additional periodic BCs) and finally fast Chebyshev transforms (general BCs). This transform ordering imposes the ordering of corresponding boundary conditions and thus the physical ordering of the problem to be solved. Backward transforms just follow the reversed order with corresponding inverse transforms. Spectral operators always perform the following steps: first forward transform input scalar fields from physical to spectral space, compute output scalar fields in spectral space and finally transform them back to physical space.

Let $\boldsymbol{\mathcal{N}}^v$ be the size of the grid discretization and $N = \prod_{i=1}^{n} \mathcal{N}_i^v$ be the total number of grid points. We saw that all forward and backward transforms can be implemented in

$$\mathcal{O}\left(N \log N\right) = \mathcal{O}\left(\sum_{i=0}^{n} \left[\prod_{\substack{j=1 \\ j \neq i}}^{n} \mathcal{N}_j^v\right] \mathcal{N}_i^v \log \mathcal{N}_i^v\right)$$

complexity with some variant of the Fast Fourier Transform. This is usually the most costly step in spectral operators because the post-processing step can be done in only $\mathcal{O}\left(N\right)$ complexity. The presence of general boundary conditions can drastically increase the post-processing step up to $\mathcal{O}\left(N^3\right)$ due to the necessity of solving a general linear system of size $N \times N$. However in practice there is rarely more than one axis that has general boundary conditions and in this case the complexity drops to

$$\mathcal{O}\left(N(\mathcal{N}_n^v)^2\right) = \mathcal{O}\left(\left(\prod_{j=1}^{n-1} \mathcal{N}_j^v\right) (\mathcal{N}_n^v)^3\right)$$

because there would be $N/\mathcal{N}_n^v$ independent linear systems to solve.

In practice many `FFT` libraries exist targeting classical `CPUs` such as `FFTW`, `FFTPACK` and `MKL-FFT` but also accelerators such as the `OpenCL` library `clFFT` and the `CUDA` li-

brary cuFFT. Those libraries are accessible in Python through pyFFTW (FFTW wrapper), NumPy (C FFTPACK wrapper), SciPy (Fortran FFTPACK wrapper), mkl_fft (MKL-FFT wrapper), gpyfft (clFFT wrapper) and scikit-cuda (cuFFT wrapper). All those libraries and wrappers but the CUDA based ones are usable in the HySoP library and are described in appendix B. The performance varies greatly from one FFT backend to another and we use gearshifft, a benchmark suite for heterogeneous implementations of FFTs [Steinbach et al. 2017], to gather runtimes for reference 2D DFTs of increasing size (see figure 3.20). Without any surprises, vendor specific libraries are always faster and open-source alternatives such as FFTW and clFFT lie behind with a two- to ten-fold performance hit.

Those libraries come with various levels of support for FFT related transforms. This includes available type of transforms (real-to-real, real-to-complex, complex-to-real, complex-to-complex), their sizes (usually a product of powers of 2, 3, 5, 7 and so on), their type (single or double-precision transforms) and whether or not they offer multithreading or multiprocessing implementations. The main problem for GPU compatible libraries (clFFT and cuFFT) is that none of them provide the real-to-real sine and cosine transforms required to implement solvers with homogeneous boundary conditions. Fortunately it is possible to implement DCT-I, II and III as well as DST-I, II and III by using classical DFTs with $\mathcal{O}(N)$ pre- or post-processing steps. The general idea is to transform input or output signals by reordering and multiplication by some twiddle factors. The method may also depend on the signal size [Wang 1985][Chan et al. 1990][Chan et al. 1992]. With those methods DCT-II, DCT-III, DST-II and DST-III can be computed efficiently in $\mathcal{O}(N \log N)$ complexity and $\mathcal{O}(\varepsilon \log N)$ root mean square error where N is the size of the transform and ε the machine floating-point relative precision. For the DCT-I and the DST-I transforms there exist $\mathcal{O}(N \log N)$ algorithms with $\mathcal{O}\left(\varepsilon\sqrt{N}\right)$ RMS error. We use the same approach as FFTW and we use $\mathcal{O}(2N \log 2N)$ implementations with $\mathcal{O}(\varepsilon \log N)$ error to get comparable accuracy between all implementations [Frigo et al. 2005]. We use the pre- and post-callback facilities offered by the clFFT library to perform those steps without requiring additional global memory transactions.

The n-dimensional transforms are then computed by iteratively performing batched one-dimensional transforms followed by local data permutations with the local permutation kernels introduced earlier. In order to do this all FFT backends are wrapped to fit a common FFT planning and queuing interface that has to handle all types of transforms, data movements, data permutations and zero-filling. The planner then proceeds to generate all required compute plans by using two additional temporary buffers to perform the required transforms. The best case scenario is when all of the data can fit inside the device's memory, in which case the transform is entirely handled on the GPU. In all others cases, the data has to pass from the device to the host, leading to a drastic performance drop due to memory transfers. As an example the NVidia Tesla V100-SXM2 offers 900GB/s of global memory bandwidth but host-device memory transfers are capped at 16GB/s (more then fifty times less). With such a reduced effective memory bandwidth, the FFT computation is likely to be memory bound.

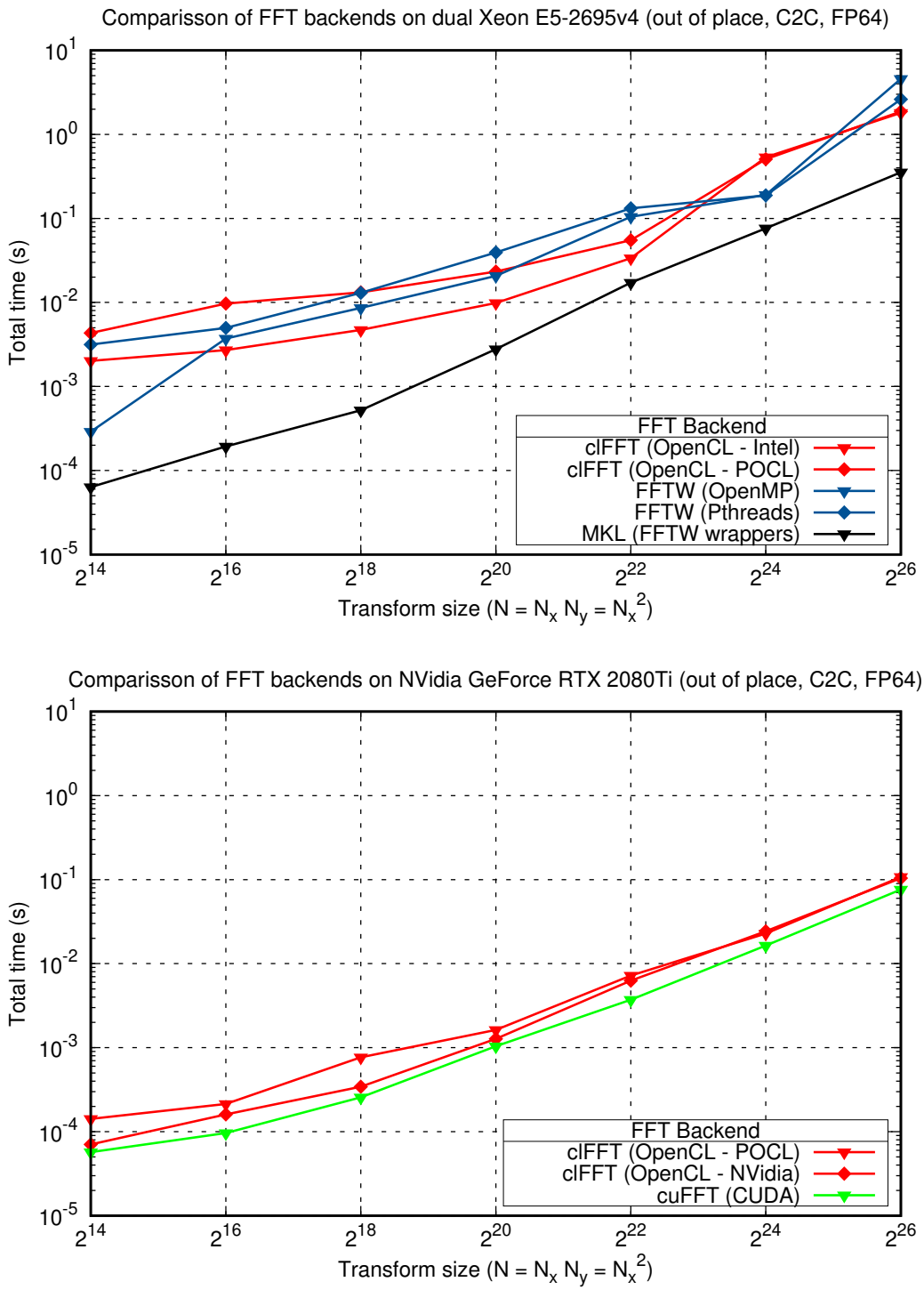


Figure 3.20 – `FFT` benchmark, 2D complex-to-complex discrete Fourier transforms Comparison of different `CPU` and `GPU` compatible `FFT` backends. Each point represent the mean over 64 runs of complex-to-complex DFT of size $N = n^2$. `FFTW` is compiled with `ICC 19.0.5` and run with either the `OpenMP` or the `Pthreads` threading layer with planning rigor flag set to `FFTW_MEASURE`. The `POCL OpenCL` platform provides support for `CPUs` but also `Nvidia GPUs` by generating directly `PTX` for the target device.

3.2.7 Navier-Stokes runtime distribution

Now that all operators have been written to work with `OpenCL`, we can run 2D and 3D incompressible Navier-Stokes simulations in their velocity-vorticity formulation (1.28). We recall that stretching only appears in the 3D case. We run 100 iterations of the problem on a fully periodic domain to analyse the average cost of one iteration without computing any timestep criteria. Spectral operators include curl computation, the poisson solver and diffusion of vorticity. Advection and stretching are split on the grid with a first order Strang splitting. Figure 3.21 shows statistics obtained on the dual `CPU` device while figure 3.22 exposes 2D and 3D statistics obtained on the two `GPU` devices. The `transpose` category now takes into account local permutations due to the directional splitting but also the permutations required to perform `FFT`s. The `D2D` category represent device-to-device copies requires for the permutations and for out-of-place `FFT`s. `diffusion`, `curl` and `poisson` categories correspond to the post-transform elemenwise kernels that are required to compute output fields in the Fourier space.

For 2D simulations the only relevant parameter seems to be the order of the remeshing kernel that goes from 12 to 17% of total runtime. Fourier transforms and permutations represent 50% of the total runtime, followed by ghosts (18%), remeshing and advection ($< 10\%$). Apart from the small increase in runtime due to high order remeshing (+6%) all the mean iteration times fluctuate around $50ms$. In the case of 3D simulations, data movements represent about 60% of the runtime (permutations, ghost exchanges and device-to-device copies). Stretching is sensible to both the order of the finite differences and the number of integration stages and represents 10 to 25% of total runtime. The highest order in space and time configuration only cost about 25% more runtime than the lowest order configuration. The `CPU` platform seems to be really inefficient in terms of ghost exchanges and is already compute bound at the advection step.

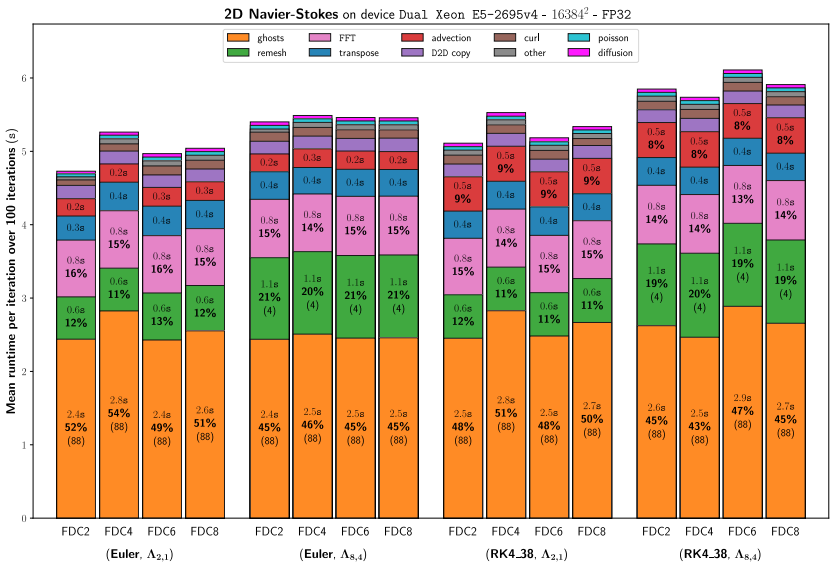


Figure 3.21 – Distribution of runtime for single-precision Navier-Stokes on `CPU`

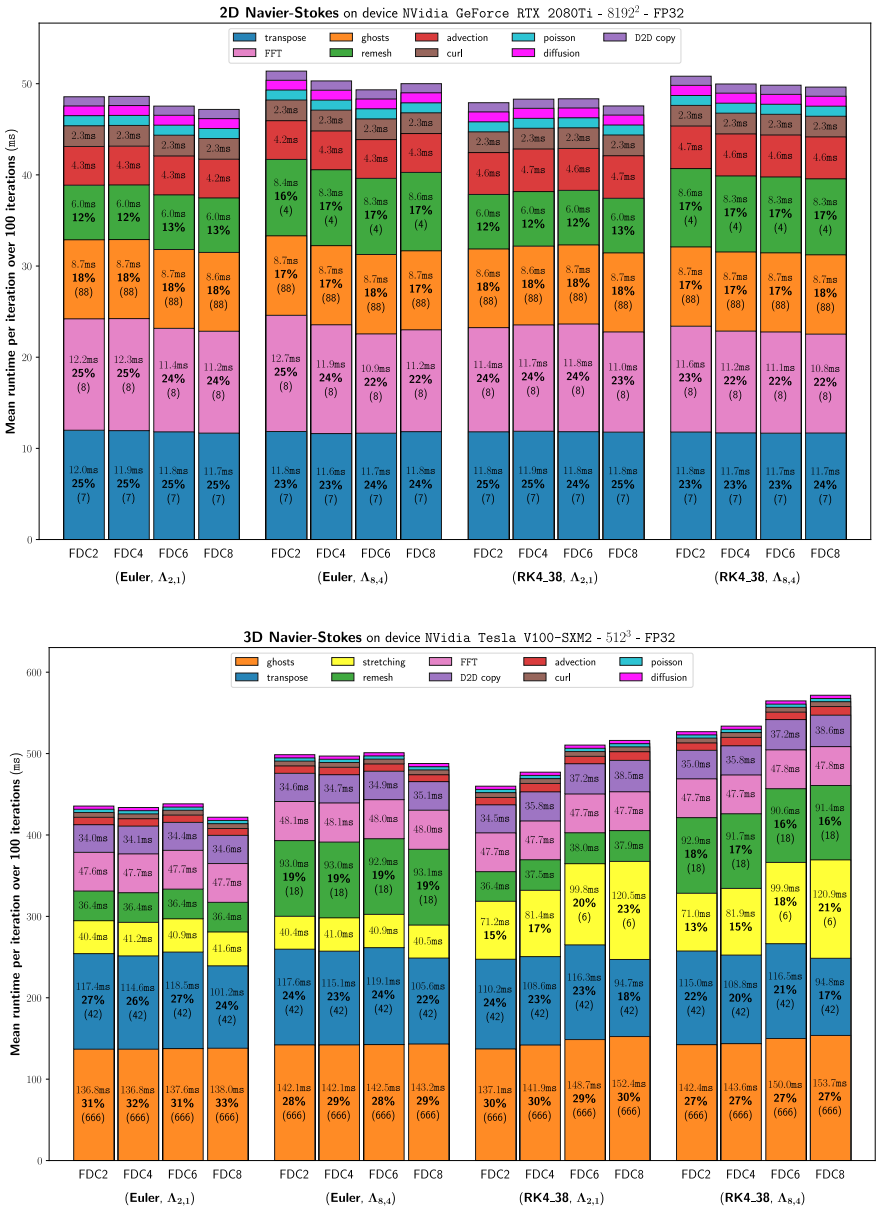


Figure 3.22 – Distribution of runtime for single-precision Navier-Stoke on GPU
Mean runtime and runtime ratio is given for different kind of kernels. Each bar corresponds to a specific configuration (time integrator, remeshing formula and finite differences order). The number between parenthesis, when given, corresponds to the number of calls per iteration.

Overall this corresponds in the worst case scenario to 1.3×10^9 vertices per second (0.74 ns/vertex) on the `NVidia GeForce RTX 2080Ti` for the 2D case and 240×10^6 vertices per second (4.2 ns/vertex) on the `NVidia Tesla V100 SXM2` for the 3D case. The dual `CPU` setup only provides 43×10^6 vertices per second (22 ns/vertex) in 2D and 7.1×10^6 vertices per second (142 ns/vertex) in 3D. From those results it is clear that the best way to achieve even better performances is to optimize and possibly eliminate useless ghost exchanges. The solver should also benefit from improved bandwidth in permutation kernels.

3.2.8 Validation of the solver

In this section we are interested in the validation of the `OpenCL` solver and associated numerical methods with the Taylor-Green vortex benchmark, which constitutes a relevant three-dimensional test case before applying those methods to more complex models. The simulation is handled in a fully periodic cubic domain of size $[0, 2\pi]^3$ and the evolution of viscous incompressible flow is described by the Navier-Stokes equations in their velocity-vorticity formulation:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{3.1a}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{3.1b}$$

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{\omega} = (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} + \nu \Delta \boldsymbol{\omega} \tag{3.1c}$$

Velocity is recovered with

$$\Delta \boldsymbol{\psi} = -\boldsymbol{\omega} \tag{3.2a}$$

$$\boldsymbol{u} = \nabla \times \boldsymbol{\psi} \tag{3.2b}$$

The velocity is initialized with the following smooth conditions:

$$u_x(\boldsymbol{x}, t = 0) = sin(x)cos(y)cos(z) \tag{3.3a}$$

$$u_y(\boldsymbol{x}, t = 0) = -cos(x)sin(y)cos(z) \tag{3.3b}$$

$$u_z(\boldsymbol{x}, t = 0) = 0 \tag{3.3c}$$

All fields are discretized on a cubic grid with n points in each directions ($N = n^3$) and the viscosity is given by $\nu = 1/Re = 1/1600$. The Poisson solver and diffusion of vorticity are computed spectrally whereas transport and stretching are solved respectively with remeshed particles and finite differences. The whole simulation is run from $t = 0$ to $t = 20$ on a single `OpenCL` device. For all runs, the problem always fits in the device's embedded memory.

We first consider a fixed timestep $dt = 10^{-3}$ (dimensionless time) and two configurations:

1. A first configuration using single-precision floating point numbers, a second order time integrator `RK2`, a remeshing formula that preserves the four first moments $\Lambda_{4,2}$, fourth order centered finite differences `FDC4` and first order Strang splitting.

2. A second configuration featuring double-precision floating point numbers, a fourth order time integrator `RK4_38`, a remeshing formula that preserves the eight first moments $\Lambda_{8,4}$, eighth order centered finite differences `FDC8` and second order Strang splitting.

Those two configurations are representative of default and highest order numerical methods available in the `HySoP` library. They should thus be representative of average and worst runtime per iteration. On the `NVidia Tesla V100-SXM2` the first configuration can be run up to $N = 896^3$ while the second configuration up to $N = 512^3$. The expected compute complexity of one timestep is $\mathcal{O}(N \log N)$ because of the `FFT`s.

This is effectively what we get in practice as seen on figure 3.23.

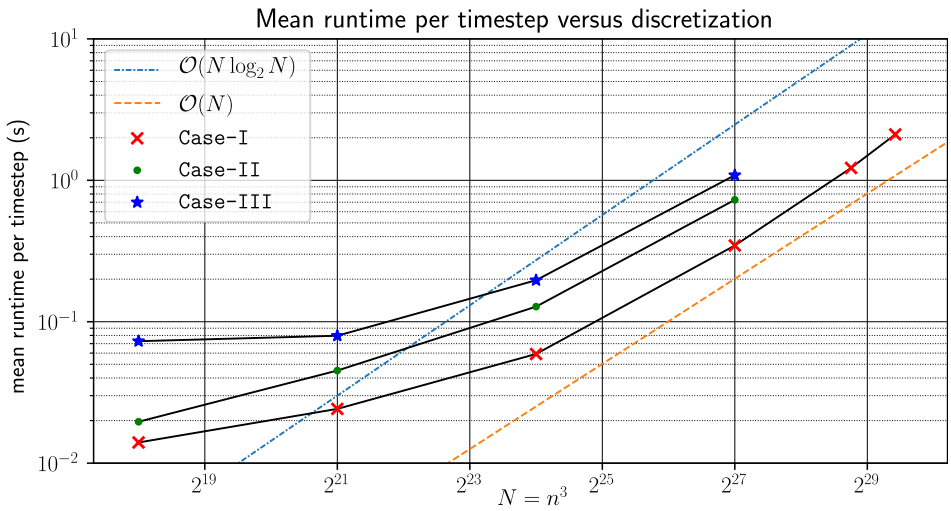


Figure 3.23 – Mean runtime per timestep for the resolution of the Taylor-Green vortex problem. Results averaged over at least 20000 iterations on grids of increasing size.

Comparison with reference data

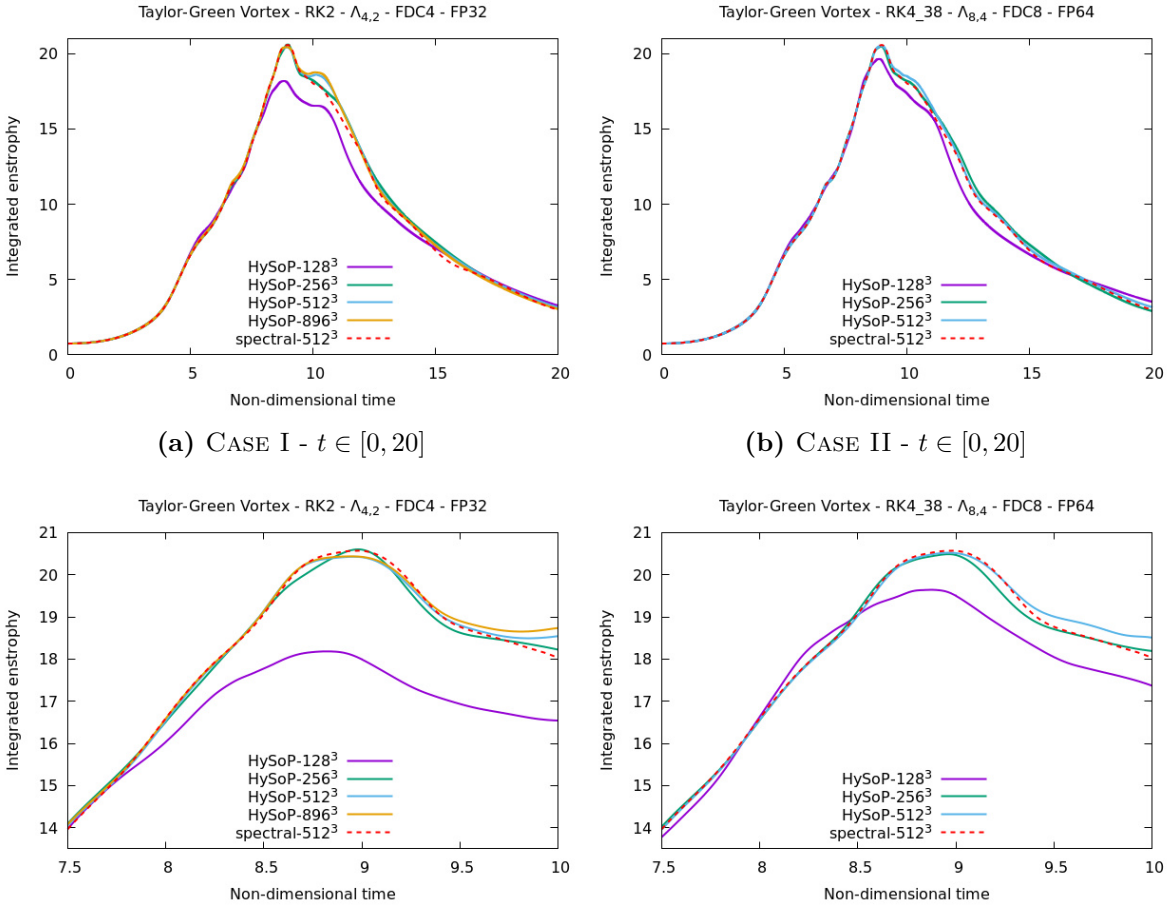
The results will be compared to two reference flow solutions obtained by dealiased pseudo-spectral solvers. The first reference solution is obtained with a dimensionless timestep $dt = 10^{-3}$ and a low storage 3-steps Runge-Kutta scheme [Hillewaert 2012]. The second reference solution uses a fourth order Runge-Kutta time integrator with a `CFL` set to 0.75 [Van Rees et al. 2011]. These two reference pseudo-spectral solvers have been grid-converged to a 512^3 grid.

The first parameter of interest is the temporal evolution of the enstrophy integrated on the domain:

$$\mathcal{E}(t) = \frac{1}{|\Omega|} \int_\Omega \boldsymbol{\omega}(t) \cdot \boldsymbol{\omega}(t) \, \mathrm{d}\Omega \tag{3.4}$$

The second set of reference data consists in the isocontours of the dimensionless norm of the vorticity at $t = 8$ and $t = 9$ in the plane $x = 0$. Reference enstrophy and isocontour data are also available from [Van Rees et al. 2011] and [Hillewaert 2012].

It can be clearly seen on figure 3.24 that both cases spatially converge to some enstrophy curve that is not far away from the reference enstrophy as spatial discretization increases. Figure 3.25 shows the distance of the enstrophy obtained with the present method to the reference data. Without any surprises the higher order method with $N = 512^3$ is closer to the reference enstrophy curve than the lower order method (even at higher discretizations $N = 896^3$). Isocontours of the norm of the vorticity are compared on figure 3.26. The discrepancies at $t = 9$ suggest to reduce the timestep around the enstrophy peak, and this can be done by introducing a variable timestep that is driven by a lagrangian CFL. Hence we introduce a third case that corresponds to `Case-II` with variable timestepping.



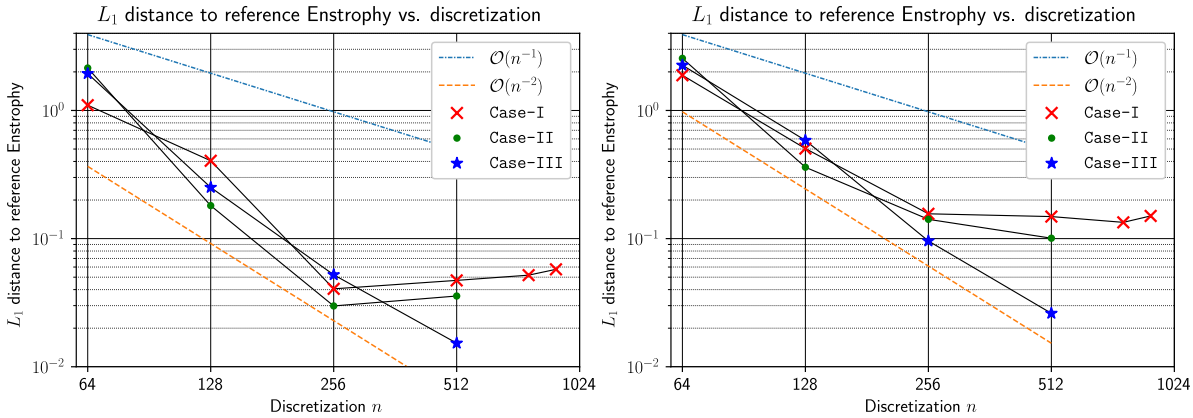
(a) Case I - $t \in [0, 20]$

(b) Case II - $t \in [0, 20]$

(c) Case I - $t \in [7.5, 10]$

(d) Case II - $t \in [7.5, 10]$

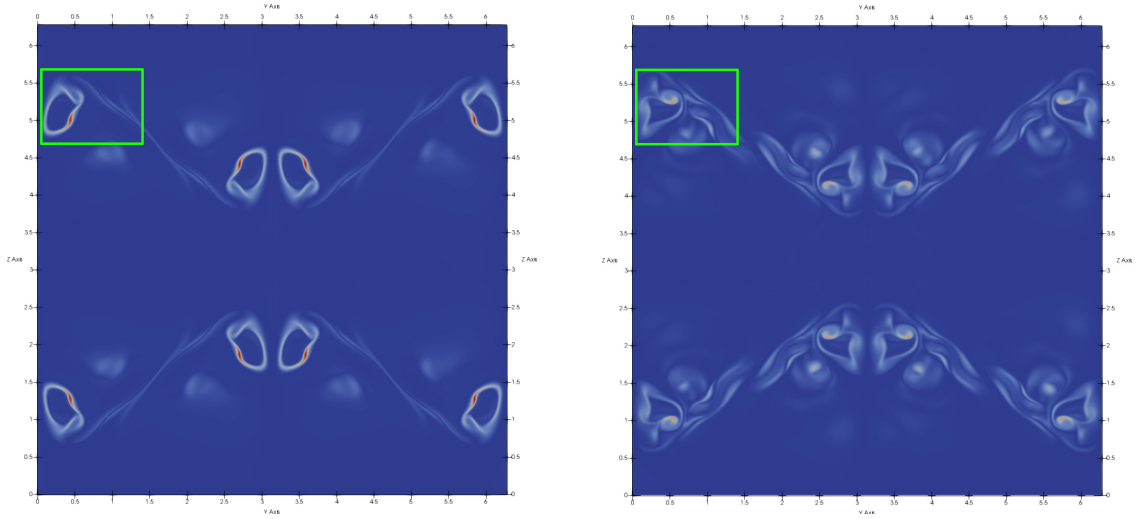
Figure 3.24 – Evolution of enstrophy with respect to time for various configurations and comparison with reference [Hillewaert 2012]. Simulation at fixed timestep $dt = 10^{-3}$.



(a) Comparison with [Van Rees et al. 2011]

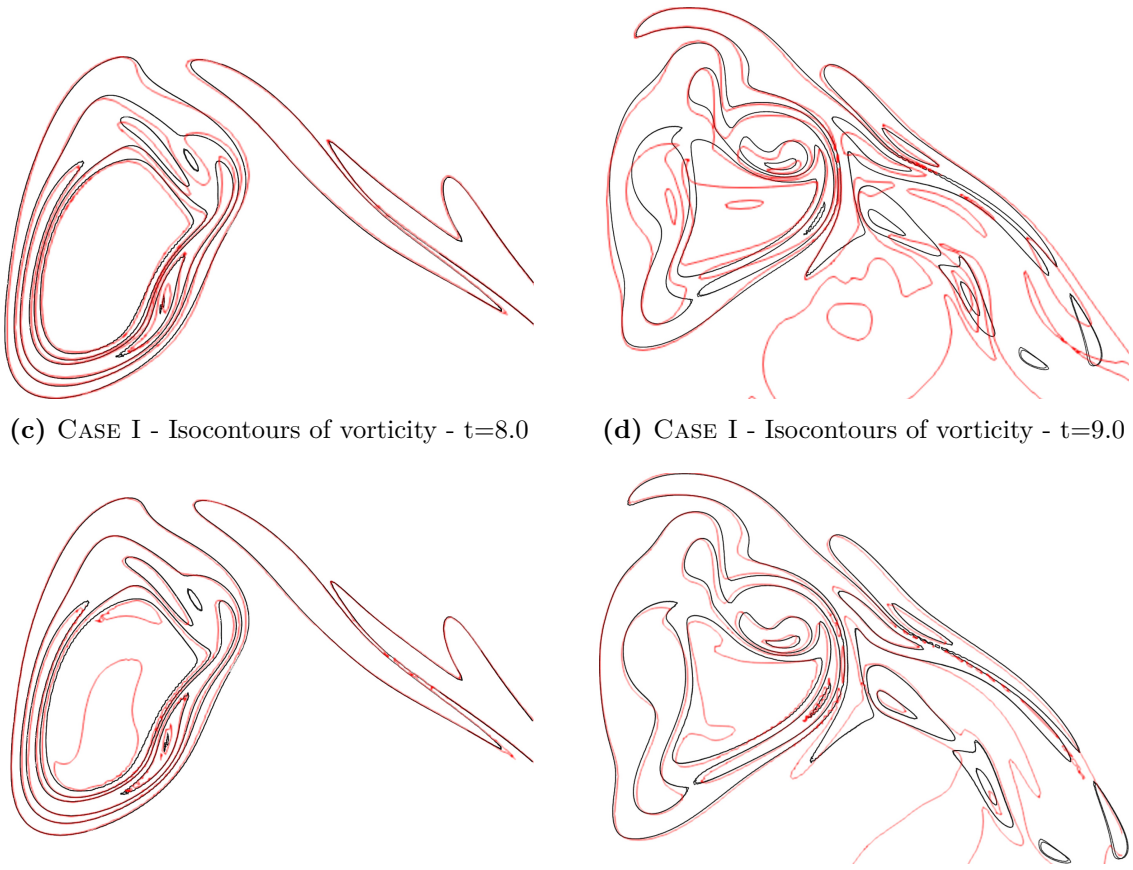
(b) Comparison with [Hillewaert 2012]

Figure 3.25 – Comparison of obtained enstrophy to reference enstrophies obtained with pseudo-spectral DNS at $N = 512^3$. `Case-III` is run with a smaller average timestep.



(a) Case I - $\|\boldsymbol{\omega}\|_2\,(x=0)$ - t=8.0

(b) Case I - $\|\boldsymbol{\omega}\|_2\,(x=0)$ - t=9.0



(c) Case I - Isocontours of vorticity - t=8.0

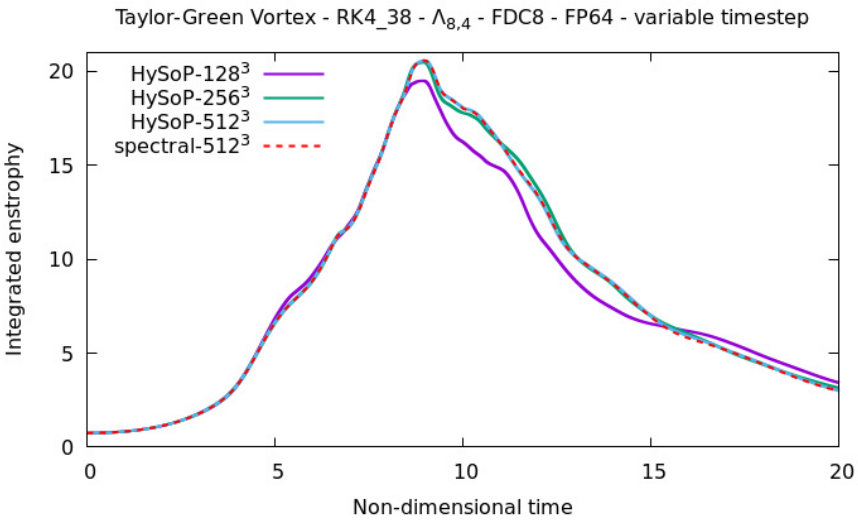
(d) Case I - Isocontours of vorticity - t=9.0

(e) Case II - Isocontours of vorticity - t=8.0

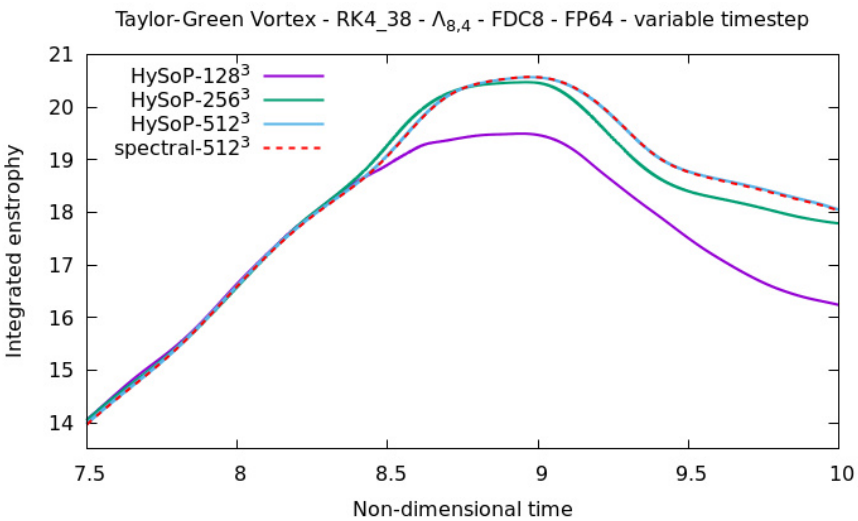
(f) Case II - Isocontours of vorticity - t=9.0

Figure 3.26 – Slice of vorticity at x=0 and comparison of isocontoursfor $\|\boldsymbol{\omega}\|_2\,(x=0) \in \{1, 5, 10, 20, 30\}$ in the top-left part of the slice (represented by a green rectangle). Solution is obtained with fixed timestep $dt = 10^{-3}$ (red) are compared with the 768^3 results of [Van Rees et al. 2011] (black). This corresponds to iterations 8000 and 9000 of the numerical simulation.

In order to fully converge in time, the third case introduces variable timestepping with drastic timestep constraints: a `CFL` condition of 1/10 along with a lagrangian `CFL` of 1/32. The resulting timestep lies between 6.7×10^{-5} and 1.3×10^{-3}, the mean timestep being around 2.3×10^{-4} (4.3 times more iterations than `Case-I` and `Case-II`). The simulation is run with the same parameters as case two (high order methods with double-precision up to 512^3 discretization). At 512^3, the resulting simulation runs on a single `GPU` and spans over 85982 iterations at 1.09s per iteration (49% performance hit due to the computation timestep criterias) which represent approximatively one day and two hours. As seen on figure 3.27, the resulting enstrophy curve at $N = 512^3$ is really close to the reference curve, even after $t = 10$ when compared to the same simulation at fixed timestep $dt = 10^{-3}$. Without variable time stepping, the total number of iterations would be around 3×10^6 (two days and 18 hours).

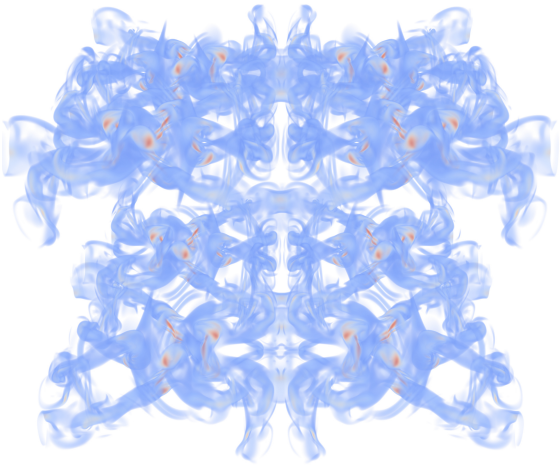


(a) Case III - $t \in [0, 20]$

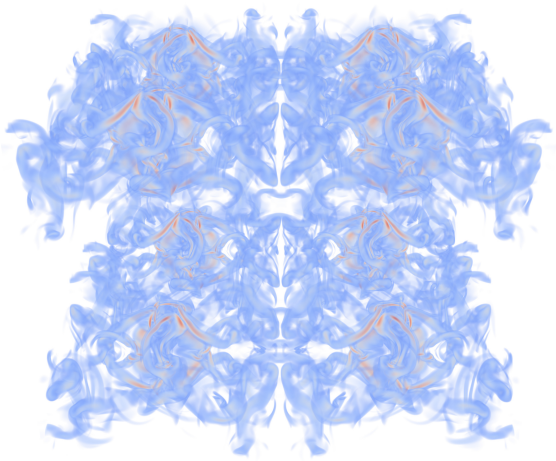


(b) Case III - $t \in [7.5, 10]$

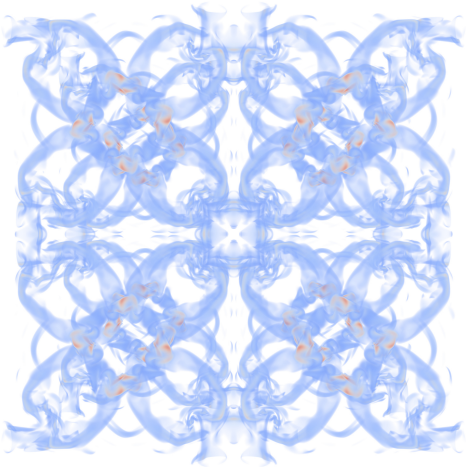
Figure 3.27 – Evolution of enstrophy with respect to time - variable timestep



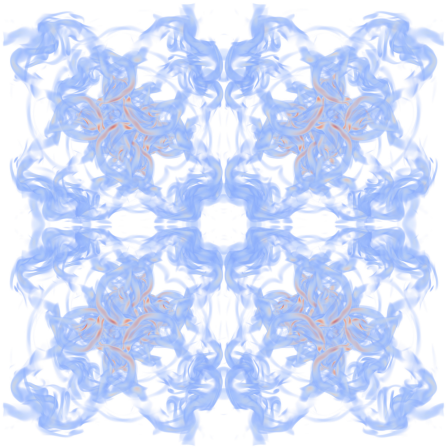
(a) Norm of vorticity - t=8.0



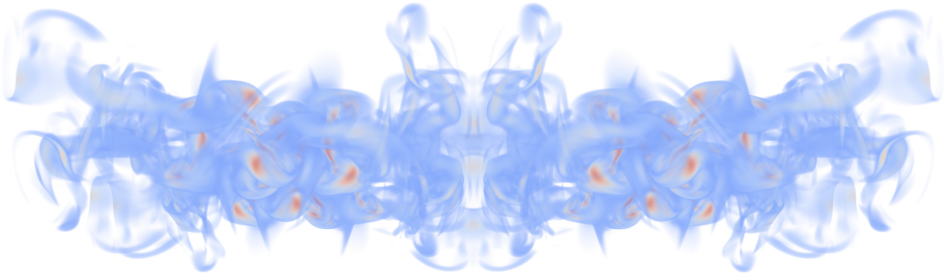
(b) Norm of vorticity - t=9.0



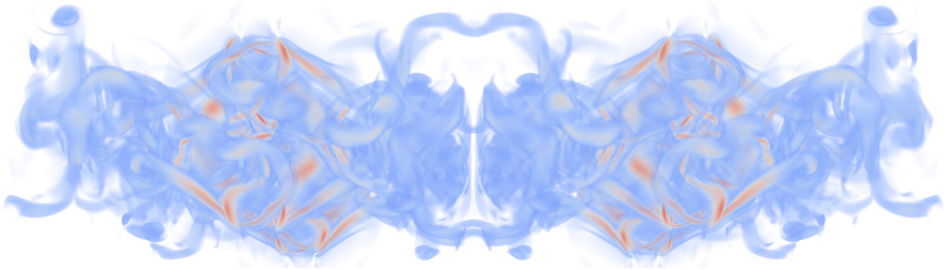
(c) Top view - t=8.0



(d) Top view - t=9.0



(e) Side view of the upper part of the compute domain - t=8.0



(f) Side view of the upper part of the compute domain - t=9.0

Figure 3.28 – Volume rendering of the norm of the vorticity $\|\boldsymbol{\omega}\| \geq 10$

3.3 Distributed computing

This section focus on the last kind of parallelism provided by the `HySoP` library, namely data-parallelism. The regular grid is split into multiple subgrids to perform domain decomposition. Each process is then responsible to handle one subdomain by using one or more compute backends. The second part focuses on the different strategies that can be adopted concerning ghost layers data exchanges with neighboring processes such as cross and diagonal ghost exchanges. In the last part we will see that the domain decomposition process is constrained by the Fourier based spectral solver.

3.3.1 Domain decomposition

The large cuboid computational domain $\mathcal{B}$ is divided into smaller cuboid subdomains. Its discretization, consisting in a rectilinear global grid of $\boldsymbol{\mathcal{N}}^v$ vertices, can be split in K local subgrids each labelled by their local cuboid subdomain position $\boldsymbol{p} = (p_1, \cdots, p_n) \in [\![0, P_1[\![\times \cdots [\![0, P_n[\![$. Every subdomain is affected to a specific compute process denoted $\mathtt{P}_{\mathtt{p}} = \mathtt{P}_{(\mathtt{p}_1,\ \ldots,\ \mathtt{p}_n)}$. The $K = \Pi_{j=1}^{n} P_j$ subdomains and processes are defined by $\boldsymbol{P} = (P_1, \cdots, P_n)$ splittings of the box, one for each direction of the space, and are indexed in a row-major ordering similar to the vertices. The local vertex grid belonging to process $\mathtt{P}_{(\mathtt{p}_1,\ \ldots,\ \mathtt{p}_n)}$ has size $N^{v,\boldsymbol{p}} = (N_1^{v,p_1}, \cdots, N_n^{v,p_n})$ and is allocated locally to the process it belongs in the main memory or in some dedicated accelerator memory by using the `OpenCL` standard. An example of uniform domain decomposition is shown on figure 3.29. However, the domain decomposition is not required to be uniform, the only condition for a splitting to be valid being, at the discrete level, the following:

$$\sum_{p_i=0}^{P_i-1} N_i^{v,p_i} = \mathcal{N}_i^v \quad \forall i \in [\![1, n]\!] \tag{3.5}$$

For the rest of the notations, we adopt the following convention: calligraphic letters such as $\mathcal{N}^v$ represent global parameters while their uppercase counterpart N^v represent local parameters. All local parameters are associated to a given subdomain indexed by $\boldsymbol{p}$ but the multi-index can be dropped when there is no ambiguity. Similarly to the global grid, the vertices belonging to the subdomain discretization are indexed using row-major ordering such that vertex $v_{\boldsymbol{j}}$ has index $I_{\boldsymbol{j}} = ((((\cdots + j_{n-3})N_{n-2}^v + j_{n-2})N_{n-1}^v + j_{n-1})N_n^v + i_n) = \boldsymbol{j} \cdot \boldsymbol{S}$ where $\boldsymbol{j}$ represent the local index and the local stride $\boldsymbol{S} = (S_1, \cdots, S_n)$ is defined by $S_i = \Pi_{j=i+1}^{n} N_j^v$. Passing from the global index $\boldsymbol{i}$ to the local index $\boldsymbol{j}$ is made possible by the knowledge for each processes of their coordinates and the exclusive prefix sum of the local grid sizes as illustrated on figure 3.29. More specifically, the process of coordinates $\boldsymbol{p} = (p_1, \cdots, p_n)$ has its local grid offset by $\boldsymbol{O^p} = (O_1^{p_1}, \cdots, O_n^{p_n})$ vertices such that $O_i^{p_k} = \sum_{p=1}^{p_k} N_i^{v,p} \quad \forall i \in [\![1, n]\!] \quad \forall p_k \in [\![0, P_i[\![$.

This leads to the following relations that links the global index $\mathcal{I}_{\boldsymbol{i}}$ and local index $I_{\boldsymbol{j}}$ for a given subdomain of coordinate $\boldsymbol{p}$, global vertex multi-index $\boldsymbol{i}$ and local vertex multi-index $\boldsymbol{j}$:

$$\mathcal{I}_{\boldsymbol{i}} = \boldsymbol{i} \cdot \boldsymbol{\mathcal{S}} \tag{3.6a}$$

$$I_{\boldsymbol{j}} = \boldsymbol{j} \cdot \boldsymbol{S} \tag{3.6b}$$

$$i_k = O_k^{p_k} + j_k \tag{3.6c}$$

with

$$i_k = \begin{cases} \left\lfloor \dfrac{\mathcal{I}_{\boldsymbol{i}}}{\mathcal{S}_k} \right\rfloor & \textbf{if } k = 1 \\ \left\lfloor \dfrac{\mathcal{I}_{\boldsymbol{i}} \texttt{ mod } \mathcal{S}_{k-1}}{\mathcal{S}_k} \right\rfloor & \textbf{else} \end{cases} \tag{3.7}$$

and

$$j_k = \begin{cases} \left\lfloor \dfrac{I_{\boldsymbol{j}}}{S_k} \right\rfloor & \textbf{if } k = 1 \\ \left\lfloor \dfrac{I_{\boldsymbol{j}} \texttt{ mod } S_{k-1}}{S_k} \right\rfloor & \textbf{else} \end{cases} \tag{3.8}$$

In practice each subdomain is extended on the boundaries such that the discretization contains additional ghosts vertices, useful for stencil based numerical methods such as finite differences [Micikevicius 2009]. The number of required ghost nodes $G = (G_1, \cdots, G_n)$ depends on the spatial order of the numerical method and the frequency of interprocess boundary data exchanges. Those ghost layers are required to compute values close to the local subdomain boundaries. Once a scalar field has been written to by an operator, its ghosts layers are invalidated and will become valid again after a ghost exchange step where all processes exchange their inner data with their neighbors as illustrated by the blue arrows on figure 3.30. In 2D, each process may exchange data with up to 4 direct neighbors and 4 diagonal neighbors. Subdomains that are on domain boundary $\overline{\mathcal{B}}$ may exchange less data if the boundary conditions are not periodic, in which case the ghost node values can be computed locally, directly from the knowledge of the boundary conditions and inner node values.

3.3.2 Interprocess communications

Each subdomain is of coordinate $\boldsymbol{p}$ is affected to a specific process $\texttt{P}_{\boldsymbol{p}}$. All processes can communicate with each other via `MPI` by using a `MPI_Cart` Cartesian topology. With such a configuration, calculations, such as a global sum, require very small amounts of data to be globally communicated between the K processes ($\mathcal{O}(K)$ communications). When possible, it is of interest to overlap communications with regular computations, so that the interprocess data exchanges happen while something is being computed, taking no extra communication time when compared with sequential execution [Khajeh-Saeed et al. 2013]. This is an important optimization because `MPI` interconnect hardware has a bandwidth $\mathcal{O}$(`10-200GB/s`) that is generally up to an order of magnitude less than local to process hardware memory bandwidth $\mathcal{O}$(`50-1000GB/s`) [Bode et al. 2004].

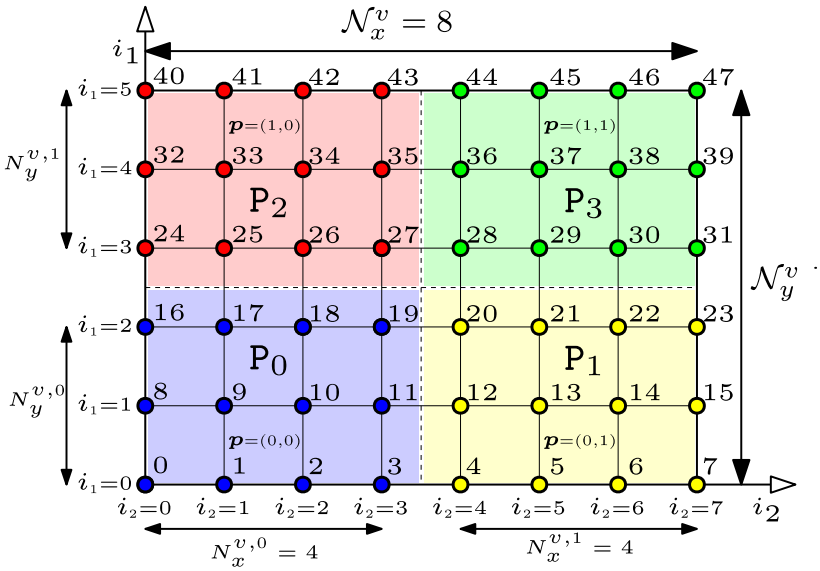


Figure 3.29 – Example of uniform grid distributionbetween 4 processes $\mathtt{P}_{(0,0)}$, $\mathtt{P}_{(0,1)}$, $\mathtt{P}_{(1,0)}$ and $\mathtt{P}_{(1,1)}$ aliased by $\mathtt{P}_0$, $\mathtt{P}_1$, $\mathtt{P}_2$ and $\mathtt{P}_3$ corresponding to the example presented in figure 2.1. The global grid of size $\boldsymbol{\mathcal{N}}_v = (6, 8)$ is split by $\boldsymbol{P} = (2, 2)$ processes in each direction such that the local grid size handled by process of coordinates $\boldsymbol{p}$, namely $\mathtt{P}_{(\mathtt{p}_y,\mathtt{p}_x)} = \mathtt{P}_{2\mathtt{p}_y+\mathtt{p}_x}$, contains $\boldsymbol{N}^{v,(p_y,p_x)} = (N_y^{v,p_y}, N_x^{v,p_x}) = (3, 4)$ vertices. The index present above each node correspond to the global node index $\mathcal{I}_{\boldsymbol{i}} = i_1\mathcal{N}_x^v + i_2 = \boldsymbol{i} \cdot \boldsymbol{\mathcal{S}}$ with global stride $\boldsymbol{\mathcal{S}} = (\mathcal{N}_x^v, 1)$.

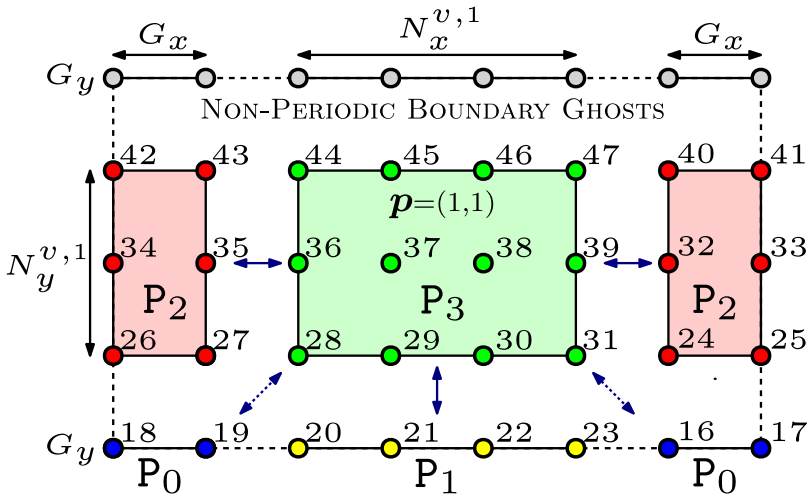


Figure 3.30 – Extension of the subdomain with ghosts nodes: The subdomain of coordinates $\boldsymbol{p} = (p_y, p_x) = (1, 1)$, discretized on a local grid of size $\boldsymbol{N}^{v,\boldsymbol{p}} = (N_y^{v,p_y}, N_x^{v,p_x}) = (3, 4)$, is extended with $\boldsymbol{G} = (G_y, G_x) = (1, 2)$ ghost nodes that overlaps the neighbor processes subdomains. Process $\mathtt{P}_3 = \mathtt{P}_{(1,1)}$ sends and receives data from and to its neighboring processes in the topology presented on figure 3.29. In this case the periodicity condition on the x-axis is such that process $\mathtt{P}_2$ is at the same time the left and right neighbor. Upper ghost nodes represented in grey accounts for a specific non-periodic boundary condition on the y-axis. The local extended subgrid has size $\boldsymbol{N}^{v,\boldsymbol{p}} + 2\boldsymbol{G} = (5, 8)$ and is represented with a dashed line.

All stencil based operators that require $\boldsymbol{G}$ ghosts to operates need to perform a ghost layer exchange for each of the scalar discrete fields they are writing to. Although the boundary data is typically small (less than 10%) compared to the internal data of the partitions, the `MPI` communications are also typically very slow (more than 10 times slower) than the internal operations. In the case of `nD` stencils that are the result of a tensor product of `1D` stencils, it is possible for each process to exchange data only with its $2n$ direct neighbors, connected by an hyperrectangle of dimension $n-1$ that we will call a `(n-1)`-rectangle or simply a `(n-1)`-face. However when the stencil is dense, a process has to communicate with all its 3^n-1 neighbors, including all the diagonals as represented on figure 3.30 with dashed blue arrows.

This splits the ghost exchange methods into two categories:

- **Cross exchange**: Each process exchange data with its direct left and right neighbors in each direction. A given process sends, for a given direction i, a `n`-rectangle ghost layer of size $(N_1, \cdots, N_{i-1}, G_i, N_{i+1}, \cdots, N_n)$ to the left and to the right, such that the size of the communication is $2(\prod_{i=1}^n N_i^{v,\boldsymbol{p}})(\sum_{i=1}^n G_i/N_i^{v,\boldsymbol{p}})$ elements per process. If the grid is uniformly distributed between processes, such that $\mathcal{N}_i^v = P_i N_i^v$ the total data sent by all the $K = \Pi_{i=1}^n P_i$ processes becomes:

$$E_{send} = 2K\left(\prod_{i=1}^n N_i^v\right)\sum_{i=1}^n \frac{G_i}{N_i^v} = \left(\prod_{i=1}^n \mathcal{N}_i^v\right)\sum_{i=1}^n \frac{2G_i}{N_i^v} \tag{3.9}$$

  We can compare this quantity with the total grid data in terms of elements $E_{tot} = (\prod_{i=1}^n \mathcal{N}_i^v)$ constituting the global grid and compute the ratio:

$$\alpha = \frac{E_{send}}{E_{tot}} = \sum_{i=1}^n \frac{2G_i}{N_i^v} = \sum_{i=1}^n \frac{2P_iG_i}{\mathcal{N}_i^v} \tag{3.10}$$

  Equation (3.10) tells us that to minimize the communications we should choose a number of processes P_i and a number of ghosts G_i that are small enough compared to the global grid size $\mathcal{N}_i^v$:

$$2P_iG_i \ll \mathcal{N}_i^v \quad \forall i \in [\![1,n]\!] \tag{3.11a}$$

$$2G_i \ll N_i^{v,p_i} \quad \forall i \in [\![1,n]\!] \quad \forall \boldsymbol{p} \in [\![0,P_1[\!] \times \cdots \times [\![0,P_n[\!] \tag{3.11b}$$

- **Full exchange**: In this case each process exchange data with all its 3^n neighbors. There exists exactly $2^{n-k}\binom{n}{k}$ k-rectangles on the boundary of a n-rectangle. The cross exchange corresponds to the case where each neighbors pair share a common k-face where $k = n-1$. For a given process and a given k, the size of the sent data belongs to $\{G_1, N_1\} \times \cdots \times \{G_n, N_n\}$ where we select k components of $\boldsymbol{N}$ and $n-k$ components of $\boldsymbol{G}$ depending on the chosen direction $\boldsymbol{d} \in \{-1,0,1\}^n \setminus \{\boldsymbol{0}\}$ where $k = n - \boldsymbol{d}\cdot\boldsymbol{d} \in [\![1,n]\!]$. If the grid is uniformly distributed between processes, such that $\mathcal{N}_i^v = P_i N_i^v$ the total data sent by all the $K = \Pi_{i=1}^n P_i$ processes becomes:

$$E_{send} = K \sum_{\substack{\boldsymbol{d}\in\{-1,0,1\}^n \\ \boldsymbol{d}\neq\boldsymbol{0}}} \left[\prod_{i=1}^n (G_i)^{|d_i|}(N_i^v)^{1-|d_i|}\right] = K\left[\prod_{i=1}^n (N_i^v + 2G_i) - \prod_{i=1}^n N_i^v\right] \tag{3.12}$$

As for the cross exchange case we can compare this quantity with the total data constituting the global grid and compute the ratio of elements to send:

$$\alpha = \frac{\prod_{i=1}^{n}(N_i^v + 2G_i) - \prod_{i=1}^{n} N_i^v}{\prod_{i=1}^{n} N_i^v} = \prod_{i=1}^{n}\left(1 + \frac{2G_i}{N_i^v}\right) - 1 = \prod_{i=1}^{n}\left(1 + \frac{2P_iG_i}{\mathcal{N}_i^v}\right) - 1 \tag{3.13}$$

As one could expect, equation (3.13) gives exactly the same condition on the number of processes and ghosts as (3.11a) and (3.11b) in order to minimize the communications.

- **Full exchange with only $2n$ neighbors:** It is also possible to exchange diagonal data by exchanging a bit more elements but with only $2n$ messages instead of 3^n. This can be achieved by performing successfully for each direction i, to the left and to the right, a cross exchange of size $(N_1+2G_1, \cdots, N_{i-1}+2G_{i-1}, G_i, N_{i+1}+2G_{i+1}, \cdots, N_n+2G_n)$ per process. If the grid is uniformly distributed between processes, such that $\mathcal{N}_i^v = P_iN_i^v$, this gives the following numbers of elements to send for all processes:

$$E_{send} = 2K\left(\prod_{i=1}^{n}(N_i^v + 2G_i)\right)\sum_{i=1}^{n}\frac{G_i}{N_i^v + 2G_i} \tag{3.14}$$

which gives a send ratio of

$$\alpha = \left(\prod_{i=1}^{n}\left(1 + \frac{2G_i}{N_i^v}\right)\right)\sum_{i=1}^{n}\frac{2G_i}{N_i^v + 2G_i} \tag{3.15}$$

In case of a uniform Cartesian grid this can be rewritten $\alpha = nx(1+x)^{n-1}$ with $x = 2\dfrac{G}{N}$.

If we want to keep the communication ratio under α for a Cartesian grid decomposed in hypercubes ($N = N_1 = \cdots = N_n$) with the same number of ghosts in any direction ($G = G_1 = \cdots = G_n$), this implies that $G < \dfrac{N}{2}\dfrac{\alpha}{n}$ for the cross exchange and $G < \dfrac{N}{2}\left[(1+\alpha)^{1/n} - 1\right] = \dfrac{N}{2}\dfrac{\alpha}{n} + \mathcal{O}\left(\alpha^2\right)$ when $\alpha \to 0$ for the full exchange. The full exchange with limited number of neighbors does not give a nice expression to describe the number of ghosts G in function of the ratio α. For all the three cases, the expression is without any surprises the same for one-dimensional problems ($n = 1$).

As expected, the number of ghosts at fixed α decreases by decreasing local grid size N. For example if each subdomain is discretized by using local grids of size 512^n and we want to keep the communications bellow $\alpha = 10\%$ this gives a maximum of 12 ghosts in 2D and 8 ghosts in 3D independently of the type of exchange. In 4D, the cross and full exchange can be made with up to 6 ghosts, but the full exchange with limited number of neighbors only allows a maximum of 5 ghosts. Although the number of ghosts may not be constrained by the exchange method for reasonable values of N and n, we must not forget that with cross exchange a process only performs communication with up to $2n$ neighbors whereas

with the full exchange a process communicates with up to $3^n - 1$ other processes (6 vs 26 in communications 3D). The higher number of communications is likely to saturate the `MPI` interconnect system and is not free if we take into account communication latencies. This is why we will use the full exchange with only $2n$ neighbors when full ghost exchange is required, at the cost of more data to communicate.

Some operations require full interprocess global communications ($\alpha \geq 100\%$). This is especially the case for the Fast Fourier Transform (FFT), extensively used by spectral operators. For each axis i, the FFT algorithm requires the knowledge of all the lines in the current axis, imposing $P_i = 1$ at the ith step such that $N_i^v = \mathcal{N}_i^v$. A solution is to set P_n to 1, yielding slab or pencil like `MPI` topologies, and to perform global data permutations in between each step [Pekurovsky 2012]. This is described in the next subsection.

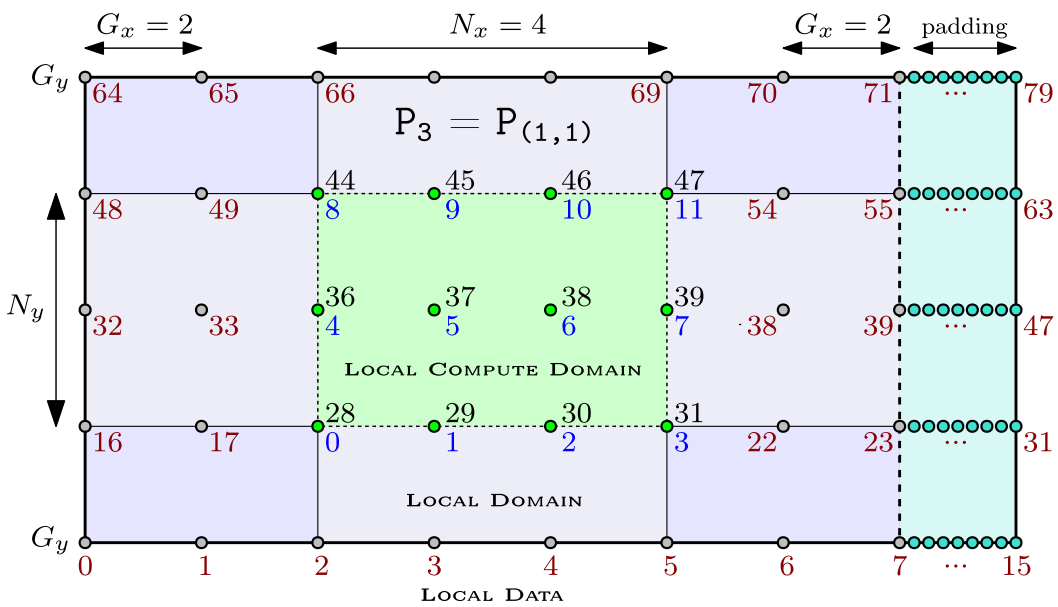


Figure 3.31 – Local grid data layout owned by process $\mathtt{P}_3 = \mathtt{P}_{(1,1)}$ as shown on figure 3.30. The full domain split by using a uniform domain decomposition with underlying local discrete grid size $\boldsymbol{N}^v = (N_y^v, N_x^v) = (3, 4)$ forming what we will call the local compute domain, represented by the green vertices. The discretization is extended on the boundaries by $\boldsymbol{G} = (G_y, G_x) = (1, 2)$ additional ghosts vertices represented in grey, forming the local domain. The local grid is allocated as as single block of memory, consisting into the local domain grid of size $\boldsymbol{N}^v + 2\boldsymbol{G}$ which rows are eventually padded by some virtual vertices, represented in turquoise, to enforce eventual memory alignement requirements. The local to process grid data is stored contiguously using a row-major ordering as shown in red. Each local compute node v is indexed by its global index $\mathcal{I}_i$, represented in black, and its local index I_j, represented in blue. Local and global vertex indices are linked by equations (3.6). In this case, if the elements stored in the grid would be double precision floating point numbers of 8 bytes each, the padding would ensure that each line is aligned on a `16x8=128B` boundary.

3.3.3 Field discretization

Each operator, as listed in algorithm (1), generally takes multiple scalar fields $R_i \in \mathcal{R}$ as inputs and writes to multiple scalar fields $W_i \in \mathcal{W}$ as outputs, some scalar fields being at the same time inputs and outputs in which case they belong to $\mathcal{R} \cap \mathcal{W}$. For example the 3D stretching operator defined by equation (2.6) takes $\mathcal{R} = \{\boldsymbol{\omega}_x, \boldsymbol{\omega}_y, \boldsymbol{\omega}_z, \boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_z\}$ as inputs and outputs the following scalar fields $\mathcal{W} = \{\boldsymbol{\omega}_x, \boldsymbol{\omega}_y, \boldsymbol{\omega}_z\}$. Each scalar field is discretized on one or more topology. A topology $\mathcal{T}(\mathcal{B}, \boldsymbol{\mathcal{N}}^v, \mathcal{O}, \boldsymbol{G})$ is defined by a domain $\mathcal{B}$, a global grid resolution $\boldsymbol{\mathcal{N}}^v$ along with the local domain decomposition defined by the knowledge of the number of splittings defined by the subgrid offsets $\mathcal{O} \in M_{K,n}(\mathbb{N})$ for all subdomains and the number of ghosts $\boldsymbol{G}$ that are the same for all the K local grids. A process is responsible to compute the outputs from the local domain discretization of the inputs of size $\boldsymbol{N}^v + 2\boldsymbol{G}$, comprising the local compute domain of size $\boldsymbol{N}^v$ and the additional ghost layers (see figure 3.31).

3.3.4 Spectral solvers

In section 2.6 we saw that the Fast Fourier Transform of multidimensional data could be performed as a sequence of one-dimensional transforms along each dimension. An three-dimensional array of size (N_z, N_y, N_x) can be Fourier transformed by first performing $N_y\, N_z$ independent one-dimensional transforms of size N_x, followed by $N_x\, N_z$ transforms of size N_y, and finally $N_x\, N_y$ transforms of size N_z. However in a domain-decomposition context, the signal to be transformed is distributed in the memory of multiple different compute nodes. In such a case, there exists at least one direction (a splitting direction) where the required one-dimensional data is not available locally to the node. There exist two main approaches when the signal does not fit entirely on one node, the first approach consisting in global redistributions of data to ensure that the required one-dimensional signal data is locally available when needed. The second approach consists into implementing directly a distributed `FFT` algorithm. This is known as the binary exchange method (as opposed to the transpose or permutation method). Those two methods are reviewed in [Foster et al. 1997] and the implementation based on global permutations of data has often be proven to be superior for large problems and in the presence of many compute nodes [Gupta et al. 1993]. As the `HySoP` library provides its own abstraction for `FFT`-based transforms, it also has to provide a way to compute those transforms in a distributed context. The implementation of distributed spectral transforms is still a work in progress but several execution policies and data redistribution methods are to be explored.

Slab decomposition is the simplest way to decompose a domain (only one direction is distributed) and a full forward transform can be computed in performing batched 2D transforms in the slabs followed by a global permutation of data and batched 1D transforms on the last axis. After this forward transform, local data is contiguous on the first axis (the z-axis). The backward transform just follows the reversed operation order to recover a contiguous x-axis. Slab decompositions are very efficient but they are limited to a small number of processes (the maximum number of processes scales with $min(N_x, N_y, N_z)$). This decomposition method is the one provided by `FFTW` [Frigo et al. 2012].

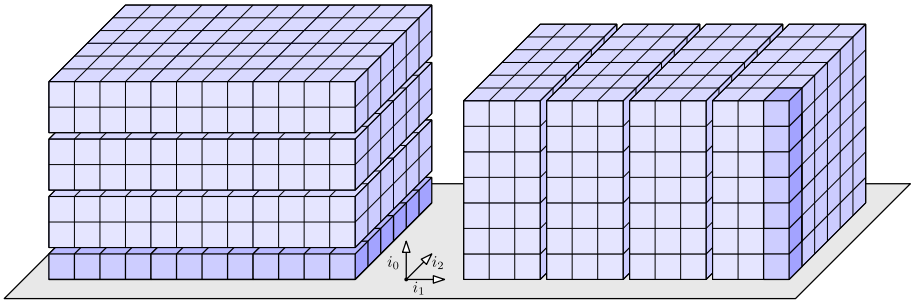


Figure 3.32 – Example of slab decomposition of a three-dimensional domain

Next levels of parallelism are achieved by splitting the domain in more directions and for a three-dimensional domain we obtain the so-called pencil decomposition. Here the maximum number of processes scales with $min(N_xN_y, N_yN_z, N_xN_z))$ but an additional global permutation is required. Many open-source implementations of pencil decomposition exist such as `P3DFFT` [Pekurovsky 2012] and `2DDECOMP&FFT` [Li et al. 2010]. `PFFT` is is a library that provides general n-dimensional pencil support on the top of `FFTW` [Pippig 2013].

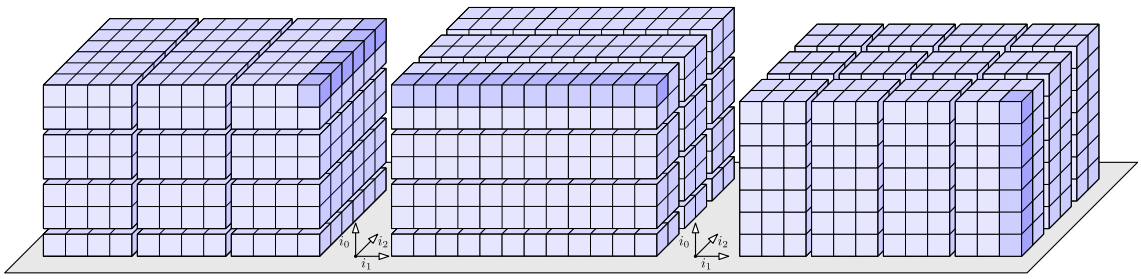


Figure 3.33 – Example of pencil decomposition of a three-dimensional domain

Apart from the differences in the way to distribute data, some methods exhibit different techniques to handle local data transpositions. Most of the methods perform local transposes prior to global data redistributions (as in the serial case, see [Frigo et al. 2005]) but permutation of data can also happen during data exchange through modern `MPI` features such as `MPI_ALLTOALLW` [Dalcin et al. 2019]. The best method will depends on many factors such as the latencies and bandwidth provided by the inter-node communication network and the ones involved in eventual host-device memory exchanges.

Conclusion

In this chapter we introduced `HySoP`, a high-performance library dedicated to fluid-related numerical simulations. The library is developed since 2012 and consists in a `Python` interface that sits on top of compiled languages such as `Fortran` and `C++`. It also relies on just-in-time compilation techniques through `OpenCL` and `Numba`. Within this framework, problems are described by building a directed acyclic graph of operators which execution order is deter-

mined by topological sort. Three different kind of parallelism are exposed, task parallelism that is directly extracted from the knowledge of the graph, accelerator parallelism that allows to use efficiently various kind of coprocessors by using the `OpenCL-1.2` standard and finally data parallelism through domain decomposition. The main contributions of this work to the `HySoP` library include the dynamic generation of certain numerical methods depending on user needs, the simplification of the user interface with the introduction of DAGs and an extensive command line interface, the generalization of directional splitting, the possibility to generate efficient accelerated elementwise and finite-differences based operators through a symbolic backend and code generation techniques, the introduction of kernel parameter autotuning, the support of non-periodic boundary conditions and finally the implementation of accelerated spectral operators.

A numpy like n-dimensional array interface targeting the `OpenCL` backend has been implemented. This interface extends `pyopencl.Array` and allows quick prototyping of `OpenCL` based operators. Advection and remeshing have been adapted to fit the new `OpenCL` code generation framework. The generated code is such that full lines of cache are not required anymore yielding some interesting performance improvements on `GPUs` and the possibility to run the solver on larger discretizations. Local data permutations kernels have been generalized to any dimension to satisfy graph builder and `FFT` planner requirements. An `OpenCL` code generator based on symbolic expressions has been implemented. All code generated kernels pass through a user-configurable autotuning process that strives to minimize kernel runtime by tweaking vectorization, global and local sizes and other kernel specific parameters. When not readily available, real-to-real transforms are computed with a classical `FFT` algorithm with Hermitian symmetry associated to some pre- or post-processing steps. Those transforms are required to handle homogeneous boundary conditions in spectral operators and are implemented on the top of many different external `FFT` libraries. An `Oclgrind` plugin has been implemented to collect `OpenCL` kernel instruction statistics and, apart from remeshing, all kernels are generally memory bound on the considered devices. For all kernels, achieved global memory bandwidth lies between 25 and 90% of maximal theoretical device bandwidth.

Each individual operator is validated within a continuous-integration development practice. The numerical method is validated on a three-dimensional Taylor-Green vortex benchmark that runs exclusively on accelerator with either fixed or variable timestep. It is shown that even if adaptive timestepping increases the mean runtime per iteration due to the computation of additional flow characteristics, the overall simulation duration decreases when compared to equivalent fixed timestep simulation. A three-dimensional incompressible Navier-Stokes problem discretized on a 512^3 grid with double-precision runs roughly at the rate of one iteration per second on a single server-grade `GPU`. There is still a lot of room for optimizations, ranging from the elimination of useless ghost exchanges to the improvement of achieved memory bandwidth in some permutation kernels. Current implementation efforts focus on `MPI` spectral transforms to allow fully distributed simulations. In the next chapter we use the `HySoP` library to perform high-performance simulations of sediment flows.

Chapter 4

Sediment-laden fresh water above salt water

Contents

4.1 Statement of the problem . . . 197
- 4.1.1 Governing equations . . . 198
- 4.1.2 Boussinesq approximation . . . 199
- 4.1.3 Boundary conditions . . . 200
- 4.1.4 Initial conditions . . . 200
- 4.1.5 Interfacial instabilities . . . 201

4.2 Setup and validation of the solver . . . 202
- 4.2.1 Outline of the method . . . 202
- 4.2.2 Timestep criterias . . . 203
- 4.2.3 Numerical setup . . . 204
- 4.2.4 Flow characteristics . . . 204
- 4.2.5 Comparison with two-dimensional reference solution . . . 206
- 4.2.6 Comparison with three-dimensional reference solution . . . 210

4.3 High performance computing for sediment flows . . . 213
- 4.3.1 Ensemble averaged two-dimensional simulations . . . 213
- 4.3.2 Three-dimensional simulations . . . 216
- 4.3.3 Achieving higher Schmidt numbers . . . 218

Introduction

Figure 4.1 – River delta of Irrawaddy, Myanmar (European Space Agency)

In this chapter we apply our numerical method to sediment flows that can occur in sediment-laden riverine outflows as depicted on figure 4.1. The processes that drive sediment settling are complex and still poorly known. Understanding the settling mechanisms is the key to be able to predict where sediments will settle. We consider a sedimentary process where small particles with negligible inertia, seen as a continuum, settle with a constant Stokes velocity above salt water. When such a layer of particle-laden fresh water flows above clear saline water, both Rayleigh-Taylor and double diffusive fingering instabilities may arise. Both of those instabilities increase the vertical transport of sediments. This simple model already raises numerical difficulties because of the high Schmidt numbers involved.

We first verify that our numerical method match the results obtained in the literature. The comparison is drawn in 2D and 3D for a large set of dimensionless parameters specific to this physical problem. We then show that within our framework, the performance delivered by `GPUs` allows us to compute large ensemble averages of parameters characterizing the flow. The ultimate goal being to understand vertical sediment transport with Schmidt numbers encountered in the nature, we finally develop strategies to increase the simulated Schmidt number.

4.1 Statement of the problem

This section follows the description of the problem originally introduced in [Burns et al. 2012]. We consider an initial stationary, unbounded single-phase fluid with a vertically stratified density profile. Within this carrier fluid, a layer of particle-laden fresh water is initially placed above denser clear, saline water. Sediment concentration C and salinity S are taken as a physical continuum and are coupled with incompressible Navier-Stokes equations. The fluid has a velocity $\boldsymbol{u}(\boldsymbol{x}, t)$ and an associated vorticity $\boldsymbol{\omega}(\boldsymbol{x}, t)$. Its density $\rho(\boldsymbol{x}, t)$ depends on the base density of clear water ρ_0, augmented by local sediment concentration $0 \leq C(\boldsymbol{x}, t) \leq 1$ and salinity $0 \leq S(\boldsymbol{x}, t) \leq 1$. The evolution of the species in the carrier fluid is governed by transport-diffusion equations, and the sediment concentration is transported relatively to some constant Stokes settling velocity V_{st} that is determined by particle properties. The presence of sediment and salt induces local gravity current which modifies the behavior of the fluid through the change of local density profiles (see figure 4.2).

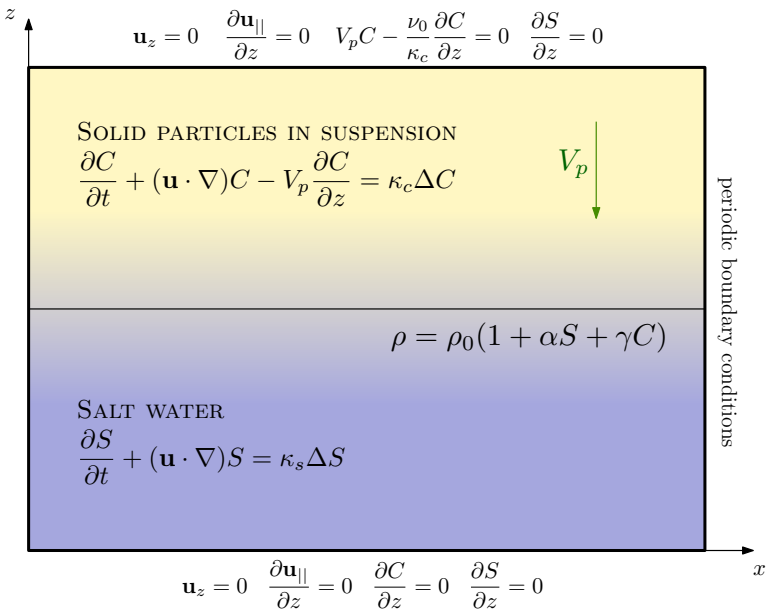


Figure 4.2 – Two-dimensional problem configuration. Here V_p corresponds to the dimensionless Stokes settling velocity of the particles while κ_* correspond to scalar diffusivities.

We do not take into account any particle-particle interactions such that the numerical model only exhibits dual-way coupling. As seen in section 1.2 this assumes that particles have negligible inertia, which is the case for silt and clay. The Stokes settling velocity of a given particle depends on its size and shape [Gibbs 1985]. Imposing the same constant settling velocity for all particles characterizes a dilute monodisperse suspension. The particles diffusion coefficient can be obtained from the Einstein-Stokes equation [Miller 1924], and is much smaller than the typical salinity diffusion coefficient. The small diffusion coefficient of the particles accounts for Brownian motion or the mixing that occurs in real polydisperse suspensions of particles, as a result of a distribution of particle shapes and sizes. The resulting dynamic of the problem should not vary significantly under $\kappa_c = 0$.

The density ρ is assumed to be a linear function in S and C:

$$\rho(\boldsymbol{x},t) = \rho_0[1 + \alpha S(\boldsymbol{x},t) + \gamma C(\boldsymbol{x},t)] \tag{4.1}$$

where (α, γ) are the density expansion coefficients of salinity and sediments. In practice, typical density expansion coefficients encountered for sediments and salt are of the order of 2 to 4%.

4.1.1 Governing equations

The resulting numerical model is very similar to the heat-salt system (1.43) used to model thermohaline convection introduced in section (1.3.2). It is extended by introducing an additional relative settling velocity in the transport term relating to the particles:

$$\rho = \rho_0 \left(1 + \alpha S + \gamma C\right) \tag{4.2a}$$

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \boldsymbol{u} = 0 \tag{4.2b}$$

$$\rho \left[\frac{\partial \boldsymbol{u}}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) \boldsymbol{u}\right] = \mu_0 \Delta \boldsymbol{u} - \nabla P + \rho \boldsymbol{g} \tag{4.2c}$$

$$\frac{\partial S}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) S = \kappa_s \Delta S \tag{4.2d}$$

$$\frac{\partial C}{\partial t} + \left(\boldsymbol{u} \cdot \nabla\right) C - V_p \frac{\partial C}{\partial z} = \kappa_c \Delta C \tag{4.2e}$$

This numerical model may exhibit double-diffusive behaviours depending on the settling velocity of the particles but also on the other constants of the problem.

The physical constants of the problem are:

ρ_0	Fresh water density
ν_0	Kinematic viscosity of fresh water
α, γ	Density expansion coefficients of salinity and sediments
κ_s, κ_c	Diffusion coefficients of salinity and sediments
V_{st}	Stokes settling velocity of the particles
$\boldsymbol{g} = -g\boldsymbol{e}_z$	Acceleration due to gravity

The nondimensionalization procedure is fully described in [Burns et al. 2012] and yields the following set of dimensionless parameters:

R_s	$=$	$\alpha S_{max}/\gamma C_{max}$	Stability ratio
S_c	$=$	ν_0/κ_s	Schmidt number between salt and water
τ	$=$	κ_s/κ_c	Diffusivity ratio between salt and sediments
V_p	$=$	$V_{st}/(\nu_0 g')^{\frac{1}{3}}$	Dimensionless settling velocity of the particles
g'	$=$	$\gamma C_{max} g$	Reduced gravity

where αS_{max} and γC_{max} refer to the maximum added density due to salinity and particles.

The average yearly sediment concentration of dirty rivers ranges from 10 to $40kg/m^3$ [Mulder et al. 1995]. The value $R_s = 2$ corresponds to a mass loading of roughly $20kg/m^3$ of sediments and a salinity of 3%. $V_p = 0.04$ corresponds to the dimensionless settling velocity of spherical particles of radius $10\mu m$ such as clays and silts. The Schmidt number between salt and water is approximately $S_c = 700$. Assuming a diffusivity ratio $\tau = 25$, this gives a Schmidt number between sediments and water of $\tau S_c = 17500$.

The physics tells us that the scalars may develop physical scales up to 27 and 133 times smaller than the finest velocity scales, for salt and sediment respectively (see section 1.1.10). Once the problem is grid converged, multiplying the grid resolution by two makes it possible to multiply the Schmidt number by a factor of four. It is thus estimated that passing from $S_c = 7$ to $S_c = 700$ on a 3D simulation will require to pass from a grid of size $1537 \times 512 \times 512$ as obtained in [Burns et al. 2015] to a grid of size $15370 \times 5120 \times 5120$ which is a problem that is 10^3 times larger then the current best simulation. The increased spatial resolution naturally comes with increased number of iterations because of timestep restrictions. While numerical considerations make it hard to achieve $S_c = 700$, the physical mechanisms to be explored are also relevant for smaller Schmidt numbers such as a warm river outflow into a cold lake ($S_c = 7$) or the settling of small water droplets in a temperature gradient in the air ($Sc = 0.7$).

4.1.2 Boussinesq approximation

As we consider only small local density perturbations, we place ourselves under the Boussinesq approximation where local variations of density are only taken into account in external forces (represented in blue in equations 4.2). This approximation has been introduced in section 1.1.6, and we perform the usual pressure shift to include the base density of water into the pressure term. In the end, when passing to the velocity-vorticity formulation the shifted pressure disappears and the resulting single-phase Boussinesq fluid is evolved with the following dimensionless set of equations:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{4.3a}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{4.3b}$$

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,\boldsymbol{\omega} - (\boldsymbol{\omega} \cdot \nabla)\boldsymbol{u} = \Delta\boldsymbol{\omega} - \nabla \times (R_s S + C)\,\boldsymbol{e_z} \tag{4.3c}$$

$$\frac{\partial S}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,S = \frac{1}{S_c}\Delta S \tag{4.3d}$$

$$\frac{\partial C}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,C - V_p \frac{\partial C}{\partial z} = \frac{1}{\tau S_c}\Delta C \tag{4.3e}$$

The velocity is recovered from the vorticity by solving the following Poisson problem:

$$\Delta\boldsymbol{\psi} = -\boldsymbol{\omega} \tag{4.4a}$$

$$\nabla \cdot \boldsymbol{\psi} = 0 \tag{4.4b}$$

$$\boldsymbol{u} = \nabla \times \boldsymbol{\psi} \tag{4.4c}$$

Note that from now on $\boldsymbol{u}$, $\boldsymbol{\omega}$, S and C refer to dimensionless fields.

4.1.3 Boundary conditions

As depicted on figure 4.2 for the 2D case, the boundary conditions in the horizontal directions are periodic. The 3D case adds an additional horizontal axis (x, y), the vertical direction being always considered as z. Top and bottom walls, denoted Γ_t and Γ_b respectively, are slip walls with no penetration. On those walls, we impose a no-flux boundary condition for the salinity field. For the sediment field we use a no-flux boundary condition formulated with respect to the Stokes settling velocity of the particles on Γ_t and the no-flux boundary condition on Γ_b.

$$\frac{\partial \boldsymbol{u}_{||}}{\partial z} = \mathbf{0} \text{ and } u_z = 0 \text{ on } \Gamma_t \cup \Gamma_b \tag{4.5a}$$

$$V_p C - \frac{1}{\tau S_c}\frac{\partial C}{\partial z} = 0 \text{ on } \Gamma_t \text{ et } \frac{\partial C}{\partial z} = 0 \text{ on } \Gamma_b \tag{4.5b}$$

$$\frac{\partial S}{\partial z} = 0 \text{ on } \Gamma_b \cup \Gamma_t \tag{4.5c}$$

The $\bullet_{||}$ notation will always denote horizontal components $\bullet_x$ in 2D and $(\bullet_x, \bullet_y)$ in 3D.

4.1.4 Initial conditions

As stated earlier the flow is initially quiescent and the initial sediment concentration field is given by a smoothed step profile. The initial salinity field is then obtained by computing $S = 1 - C$.

$$\boldsymbol{u}_0(x, y, z) = \mathbf{0} \text{ and } \omega_0(x, y, z) = 0 \tag{4.6a}$$

$$C_0(x, y, z) = \frac{1}{2}\left[1 + erf\left(\frac{z - \delta(x, y)}{l_0}\right)\right] \tag{4.6b}$$

$$S_0(x, y, z) = 1 - C_0(x, y, z) \tag{4.6c}$$

$\delta(x)$ is a small initial random perturbation that is uniformly distributed. The initial thickness of the profile l_0 represent less than 1% of the total domain height and for all this chapter we fix l_0 to 1.5. Initial sediment concentration profile $C_0(z)$ and salinity profile $S_0(z)$ are given for $\delta = 0$ on figure 4.3.

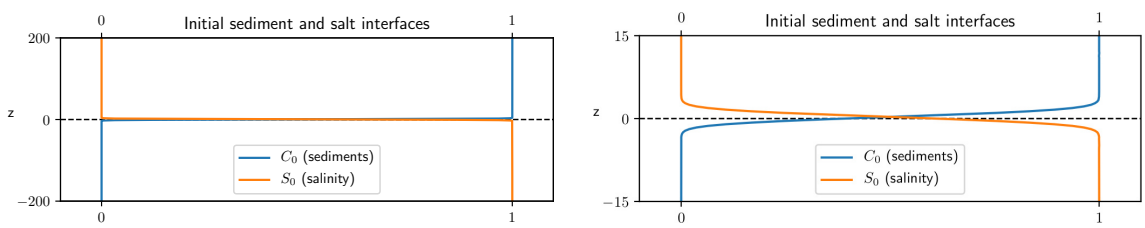


Figure 4.3 – Initial vertical sediment profile for $z \in [-200, 200]$ and $l_0 = 1.5$.

Unless otherwise specified we define $\delta(x, y)$ as $p_0\, l_0\, [\mathcal{U}(0, 1) - 0.5]$ with $p_0 = 0.1$ so that the initial perturbation stays within $\pm 5\%$ of initial profile thickness l_0. Here $\mathcal{U}(0, 1)$ represent random samples obtained from a uniform distribution over $[0, 1]$.

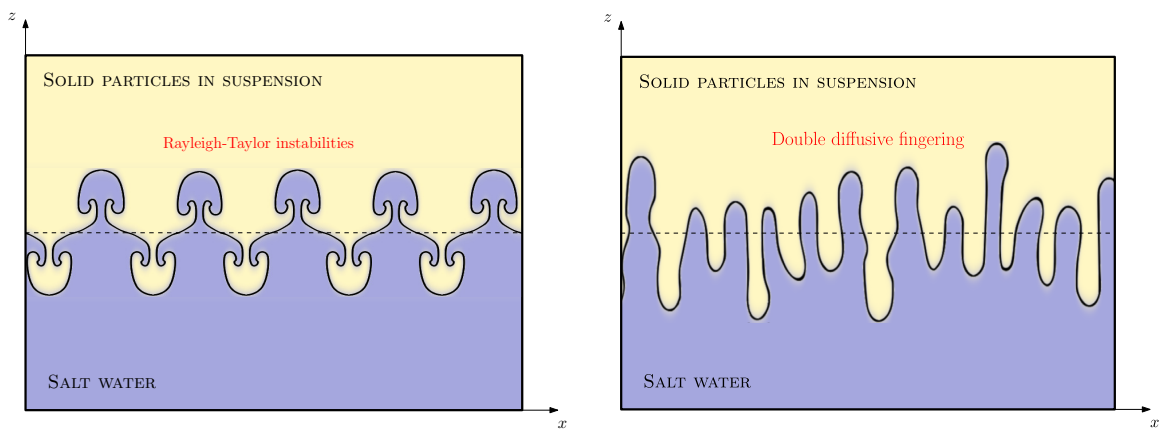
4.1.5 Interfacial instabilities

Similarly to the heat-salt system, the initial vertical density profile $\rho_0(z) = \rho(z, t = 0)$ is such that the system is initially gravitationally stable ($\frac{\partial \rho_0}{\partial z} < 0$). Here both the settling velocity of the particles and the difference in the scalar diffusivities may destabilize the initial interface.

If we first consider $V_p = 0$, the problem is purely double-diffusive and we know that the form of the resulting vertical motions depends on whether the driving energy comes from the component having the high or low diffusivity (section 1.3.2). We recall that in this case two basic type of convective instabilities may arrise: diffusive and fingering configurations. In our problem, a layer of particle-laden fresh water initially lies on top of a denser layer of saline water. Because the salinity S, which diffuses $\tau = 25$ times faster than sediments, provides the stable stratification, the resulting instabilities will be of finger type.

The presence of a non-zero settling velocity V_p can significantly increase the instability growth rate because particles settle downwards into the upper region of the saline layer, where they form an unstable layer of excess density which is gravitationally unstable. An important parameter is the ratio of the particle settling velocity to the diffusive spreading velocity of the salinity interface. When this ratio is kept below unity, instabilities are still mostly determined by double-diffusive effects. For ratios above unity, large settling velocities or equivalently large Schmidt numbers, the salinity does not diffuse fast enough to keep up with the particles that are settling. In this case, it is expected that Rayleigh-Taylor instabilities dominate the flow [Lord 1900].

Hence, when a layer of particle-laden fresh water flows above clear saline water, both Rayleigh-Taylor and double diffusive fingering instabilities may arise. Those two kind of instabilities are represented on figure 4.4. In both cases, the vertical fluxes can be much larger than the vertical transport in a single-component fluid because of the coupling between diffusive and convective processes [Turner 1985].



(a) Rayleigh-Taylor instabilities **(b)** Double-diffusive fingering

Figure 4.4 – Sketch of the different kind of instabilities the flow can develop. Figure adapted from [Burns et al. 2015]. The dotted line represent sediment interface location.

4.2 Setup and validation of the solver

In this section we describe how the algorithm to solve incompressible Navier-Stokes equations coupled with a passive scalar is extended to handle dual-way coupling with sediment and salinity fields. After defining all the timestep criterias required for this particular problem, the problem is described within the `HySoP` framework. The resulting solver is compared to state of the art numerical simulations for reference two- and three-dimensional flow configurations.

4.2.1 Outline of the method

In order to solve the dimensionless problem 4.3 we use the same approach as in algorithms 1 and 4. In those algorithm we replace the transport and diffusion of the passive scalar θ at fixed velocity $\boldsymbol{u}$, by the independent transport and diffusion of scalar field S and sediment concentration field C. The stretching and buoyancy term $\nabla \times (R_s\, S + C)$ are then computed and integrated into the vorticity $\boldsymbol{\omega}$ by using the directional splitting approach introduced in section 2.5.

Given an initial state $(\boldsymbol{u}^k, \boldsymbol{\omega}^k, C^k, S^k)$ at a given time t^k, we can compute $(\boldsymbol{u}^{k+1}, \boldsymbol{\omega}^{k+1}, C^{k+1}, S^{k+1})$ at time $t^{k+1} = t^k + dt^k$ by using the following algorithm:

1. Independently perform each transport-diffusion steps for the vorticity, salinity and sediment concentration fields at fixed velocity. Diffusion can be either spectral or finite-differences based depending on the timestep criterias and actual grid discretization.
   (a) Obtain $\boldsymbol{\omega}^{k,1}$ from $\boldsymbol{\omega}^k$, $\boldsymbol{u}^k$ and equation (4.3c).
   (b) Obtain S^{k+1} from S^k, $\boldsymbol{u}^k$ and equation (4.3d).
   (c) Obtain C^{k+1} from C^k, $\boldsymbol{u}^k - V_p \boldsymbol{e_z}$ and equation (4.3e).
2. Compute conservative stretching (1) and the Buoyancy term $\nabla \times (R_s\, S + C)$ directionally by using explicit finite differences. Integrate those terms into $\boldsymbol{\omega}^{k,1}$ with an explicit Runge-Kutta time integrator and a directional Strang splitting to obtain $\boldsymbol{\omega}^{k,2}$.
3. Correct the newly obtained vorticity $\boldsymbol{\omega}^{k,2}$ to obtain the divergence-free vorticity $\boldsymbol{\omega}^{k+1}$ and compute $\boldsymbol{u}^{k+1}$ spectrally with equations 4.3a, 4.3b, 4.4 and algorithm 11.
4. Compute timestep criterias and determine the new timestep by computing
$$dt^{k+1} = \min\left(dt^{k+1}_{\texttt{CFL}}, dt^{k+1}_{\texttt{LCFL}}, dt^{k+1}_{\texttt{STRETCH}}, dt^{k+1}_{\texttt{DIFF}}, dt_{\texttt{MAX}}\right) \tag{4.7}$$
   Advance in time by setting $t^{k+1} = t^k + dt^k$ and $k = k + 1$.

Within a multiscale approach where the sediments and salinity fields are discretized on a finer grid than the one of the velocity and the vorticity, this algorithm is modified such that velocity is interpolated to the fine grid prior to step 1 and scalar fields are restricted to the coarse grid in-between step 1 and 2. By default all fields are defined on the same grid and the fastest diffusing component being the vorticity (for S_c above unity) the vorticity is diffused spectrally while the scalars are diffused by using explicit finite differences split on the grid.

4.2.2 Timestep criterias

Let $\boldsymbol{dx} = (dz, dy, dx)$ denote the space step associated to the coarse grid where the flow is discretized ($\boldsymbol{u}$, $\boldsymbol{\omega}$) and $\boldsymbol{dx}^* = (dz^*, dy^*, dx^*)$ the one associated to the grid where the scalars are discretized (S, C). In a multiscale context we have $dx_i^* \leq dx_i$ for all $i \in [\![1, n]\!]$ else $\boldsymbol{dx}^* = \boldsymbol{dx}$. As for the Taylor-Green problem with variable timestep introduced in section 3.2.8 the timestep is determined by:

1. An arbitrary CFL to determine the maximum number of advection ghosts on the coarse grid. Here we have also to take into account directional splitting and additional settling velocity V_p:

$$CFL = \max\left(\frac{dt\,\|u_x\|_\infty}{dx}, \frac{dt\,\|u_y\|_\infty}{dy}, \frac{dt\,\|u_z\|_\infty}{dz}, \frac{dt\,\|u_z - V_p\|_\infty}{dz},\right) \leq C_{\texttt{CFL}} \tag{4.8}$$

2. A lagrangian CFL condition bellow unity that does not depend on any space step so that particles do not cross during the advection step. Because the settling velocity is constant, we recover the classical LCFL timestep criteria, here formulated with respect to vorticity instead of the gradient of velocity:

$$LCFL = \max\left(dt\,\|\boldsymbol{\omega}_x\|_\infty, dt\,\|\boldsymbol{\omega}_y\|_\infty, dt\,\|\boldsymbol{\omega}_z\|_\infty\right) \leq C_{\texttt{LCFL}} \leq 1 \tag{4.9}$$

3. A stability condition for the stretching term:

$$dt \max_{i\in[\![1,3]\!]}\left(\sum_{j\in[\![1,3]\!]}\left\|\frac{\partial u_i}{\partial x_j}\right\|_\infty\right) \leq C_{\texttt{STRETCH}} \tag{4.10}$$

   where $C_{\texttt{STRETCH}}$ is a constant depending on the time integration scheme used to discretize the stretching. For explicit Runge-Kutta schemes we have $C_{\texttt{STRETCH}} \geq 2$ (see [Mimeau 2015], appendix B).

4. A stability condition for the explicit finite-differences based diffusion terms:

$$\max\left(\frac{dt\,S_c^{-1}}{(dx^*)^2}, \frac{dt\,S_c^{-1}}{(dy^*)^2}, \frac{dt\,S_c^{-1}}{(dz^*)^2}, \frac{dt\,(\tau S_c)^{-1}}{(dx^*)^2}, \frac{dt\,(\tau S_c)^{-1}}{(dy^*)^2}, \frac{dt\,(\tau S_c)^{-1}}{(dz^*)^2}\right) \leq C_{\texttt{DIFF}} \tag{4.11}$$

   where $C_{\texttt{DIFF}} \geq 0.5$ is a constant depending on the chosen explicit Runge-Kutta scheme.

An additional constant timestep criteria $dt_{\texttt{MAX}}$ ensures the convergence of the coupling between the carrier fluid $(\boldsymbol{u}, \boldsymbol{\omega})$ and the transported scalars (S, C). Note that because the flow is initially quiescent, this constant timestep drives the simulation during the first timesteps. In practice because the flow is laminar ($R_e = 1$), this timestep criteria may constraint the timestep during the whole simulation under reasonable discretization and particle settling velocity V_p. Unless otherwise specified, we use $C_{\texttt{CFL}} = 1.0$, $C_{\texttt{LCFL}} = 0.95$, $C_{\texttt{STRETCH}} = 2$, $C_{\texttt{DIFF}} = 0.5$ and $dt_{\texttt{MAX}} = 0.1$.

4.2.3 Numerical setup

Most of the required numerical routines have already been introduced in chapter 3 and the problem is described within the `HySoP` framework by using existing operators. We use the $\Lambda_{4,2}$ remeshing kernel along with fourth-order Runge-Kutta scheme `RK4`, fourth-order centered finite-differences `FDC4` and second order Strang splitting. The fields are discretized by using single-precision floating point numbers. The resolution of this problem on the `OpenCL` computing backend requires minimal changes to the existing codebase: the buoyancy term is computed from the directional symbolic code-generation interface introduced in section 3.2.5 and advection routines are adapted to take into account an optional relative velocity. The library generates a graph containing 85 operators with up to nine independent execution queues. This graph includes operators that compute the timestep criterias as well as monitors that compute flow characteristics during the simulation.

The boundary conditions on Γ_b and Γ_t are homogeneous boundary conditions that are compatible with real-to-real spectral transforms introduced in section 2.6.4. In all the other directions the domain is periodic and we use classical Fourier transforms. The slip walls with no penetration boundary condition $\dfrac{\partial \boldsymbol{u}_{||}}{\partial z} = \mathbf{0}$ and $u_z = 0$ also yields real-to-real transform compatible vorticity boundary conditions. All top and bottom no-flux boundary conditions on the scalar fields are handled with homogeneous Neumann boundary conditions. The height of the domain is set to be sufficiently large such that the results are not affected by the lower or upper boundary planes.

4.2.4 Flow characteristics

In order to verify that our solver performs well when compared to state of the art numerical results we have to compute many different horizontally averaged flow characteristics. We use the same notations as in [Burns et al. 2015]. Horizontally averaged quantities are computed as:

$$\langle \bullet \rangle (z,t) = \int_{x_{min}}^{x_{max}} \int_{y_{min}}^{y_{max}} \bullet(x,y,z,t) \mathrm{d}x\mathrm{d}y \tag{4.12}$$

The quality of horizontally averaged statistics can be improved by using a large horizontal domains or by computing ensemble averages over multiple runs with the same parameters.

Scalar interfaces tracking: The error function is solution of the laminar diffusion equation with constant diffusion coefficient starting from a discontinuous initial condition [List et al. 1979] and serves to provide a good physical approximation of actual sediment and salt concentration profiles. We can thus track both the thickness of a given interfacial region as well as its position by computing the least-squares fit of the horizontally averaged scalar profiles to an error function:

1. We fit $\langle C \rangle (z,t)$ against $C_{fit}(z,t) = 0.5 \left[1 + \mathrm{erf}\left(\dfrac{z - z_c(t)}{l_c(t)} \right) \right]$ to obtain $z_c(t)$ and $l_c(t)$.

2. We fit $\langle S \rangle (z,t)$ against $S_{fit}(z,t) = 0.5 \left[1 - \mathrm{erf}\left(\dfrac{z - z_s(t)}{l_s(t)} \right) \right]$ to obtain $z_s(t)$ and $l_s(t)$.

$z_\bullet$ is the location at which $\langle \bullet \rangle_{fit} = 0.5$, location of the horizontally averaged interface, whereas $l_\bullet$ represents the thickness of the horizontally averaged profile. The optimization process is done every few timesteps and takes as input the last known interface location $z_\bullet^*$ and thickness $l_\bullet^*$, initial values being set to $z_c = z_s = 0$ and $l_c = l_s = l_0$. The fitting procedure is weighed more heavily around the last known position of the interface $z_\bullet^*$ than near the end points of the profile. The weights are chosen to have analytical formula $w(z) = \dfrac{1}{1 + 10^{-2}(z - z_\bullet^*)^2}$.

Height, thickness ratio and nose region ratio: After the fitting procedure we can compute the sediment to salinity thickness ratio as well as the offset between the interface locations. The thickness ratio is defined as $\xi(t) = l_c(t)/l_s(t)$ and the height is defined as the difference between the upward salinity interface and downward sediment interface locations: $H(t) = z_s(t) - z_c(t) \leq 0$. Within this nose region both salinity and sediment are present in high concentrations.

The nose region height to salinity profile thickness ratio is defined as $R_t(t) = H(t)/l_s(t)$. Note that both $H(t)$ and $l_s(t)$ diffuse with time but the nose region remains embedded within the larger region containing the salinity gradients: $H(t) < l_s(t)$. It has been observed in [Burns et al. 2012] that this ratio will dictate the dominant type of instability:

- $H/l_s \leq \mathcal{O}(0.1)$: double-diffusive fingering dominates.
- $H/l_s \geq \mathcal{O}(0.1)$: sediment and salinity interfaces become increasingly separated in space, the dominant instability mode becomes Rayleigh-Taylor like.

They show that the ratio R_t initially grows and then plateaus, at a value that is determined by the balance between the flux of sediment into the nose region coming from above, the sediment flux out of the nose region below, and the rate of sediment accumulation within the region.

Vertical interface velocities: Upwards salinity interface velocity v_{zs} is obtained by a linear fit of $z_s(t - t_0) = v_{zs}\, t$ where t_0 represents the delay before the salinity interface begins to move upwards. Downwards sediment interface velocity v_{cs} is obtained by a linear fit of $z_c(t - t_0) = v_{cs}\, t$. Before the initial delay t_0 it is expected that $z_c(t) = -V_p\, t$ and $z_s(t) \simeq 0$.

Turbulent interface diffusivities: Fitting the interfacial thicknesses $l_\bullet$ to diffusive $\sqrt{t}$ profiles allows us to compute the turbulent diffusivities of the sediment and salt interfaces. If we assume that the averaged profiles evolve as $\dfrac{\partial \langle S \rangle}{\partial t} = \dfrac{1}{\overline{S_c}} \dfrac{\partial^2 \langle S \rangle}{\partial z^2}$ and $\dfrac{\partial \langle C \rangle}{\partial t} = \dfrac{1}{\bar{\tau}\overline{S_c}} \dfrac{\partial^2 \langle C \rangle}{\partial z^2}$

we can fit

- $l_s(t)$ to $l_{s,fit}(t) = \dfrac{1}{\overline{S_c}} t^{\frac{1}{2}}$ to obtain $\overline{S_c}$.
- $l_c(t)$ to $l_{c,fit}(t) = \dfrac{1}{\bar{\tau}\ \overline{S_c}} t^{\frac{1}{2}}$ to obtain $\bar{\tau}$.

4.2.5 Comparison with two-dimensional reference solution

The first reference simulation is obtained from [Burns et al. 2015] with the following low Schmidt number configuration, discretized over a grid of size $\mathcal{N}^v = (3n+1, n)$ with $n = 1024$:

- $t \in [t_{start}, t_{end}] = [0, 500]$
- $\Omega = [z_{min}, z_{max}] \times [x_{min}, x_{max}[= [-600, +600] \times [0, 750[$
- $S_c = 0.7, \tau = 25, V_p = 0.04, R_s = 2$

Sediment and salt interfaces are fitted with error functions during the simulation every $\Delta_t = 1$ dimensionless time. This yields 500 estimations of interface locations (z_c, z_s) and interface thicknesses (l_c, l_s). Snapshots of horizontally averaged profiles of sediment concentration and salinity as well as their error function fit are plotted on figure 4.5.

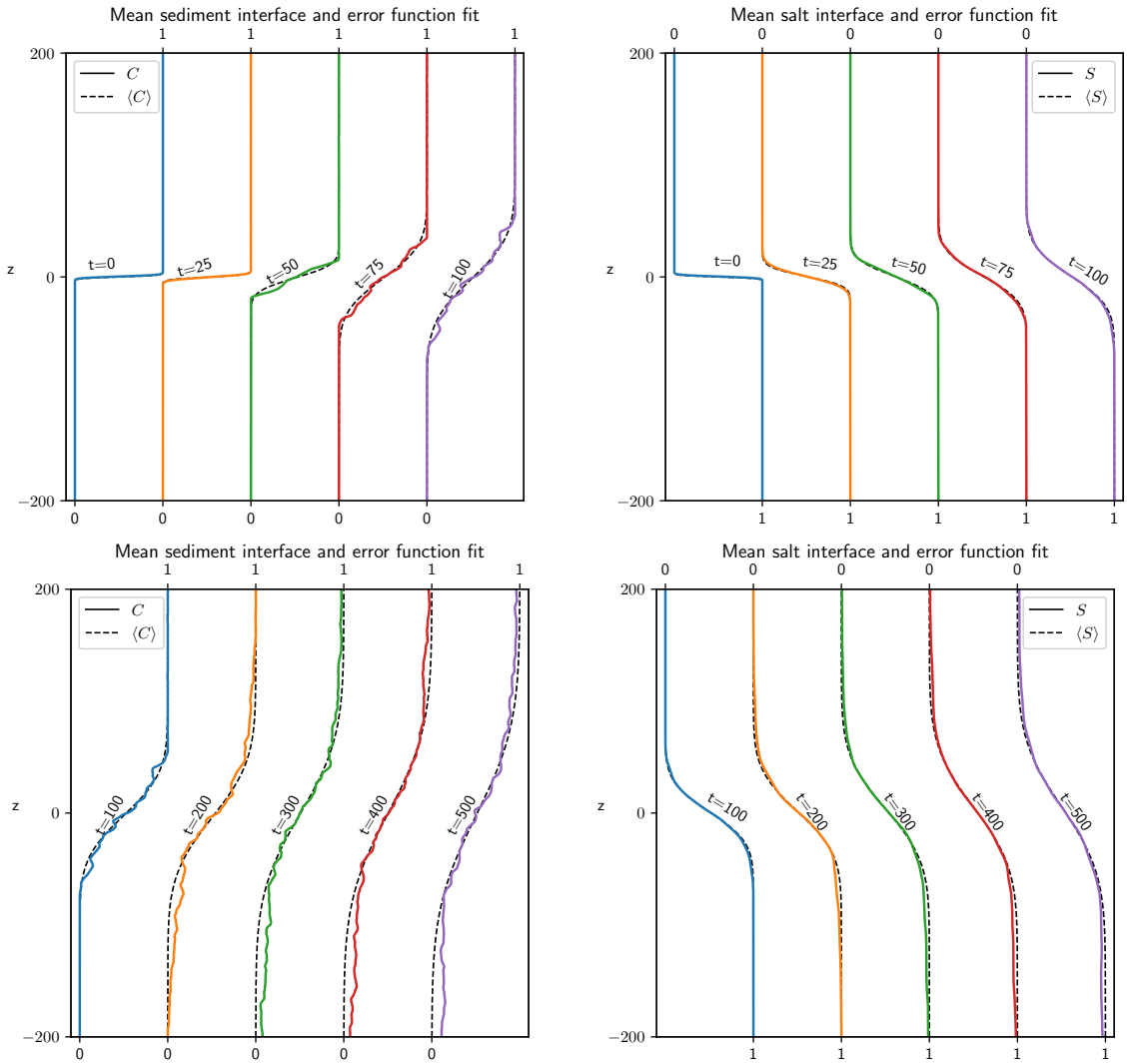
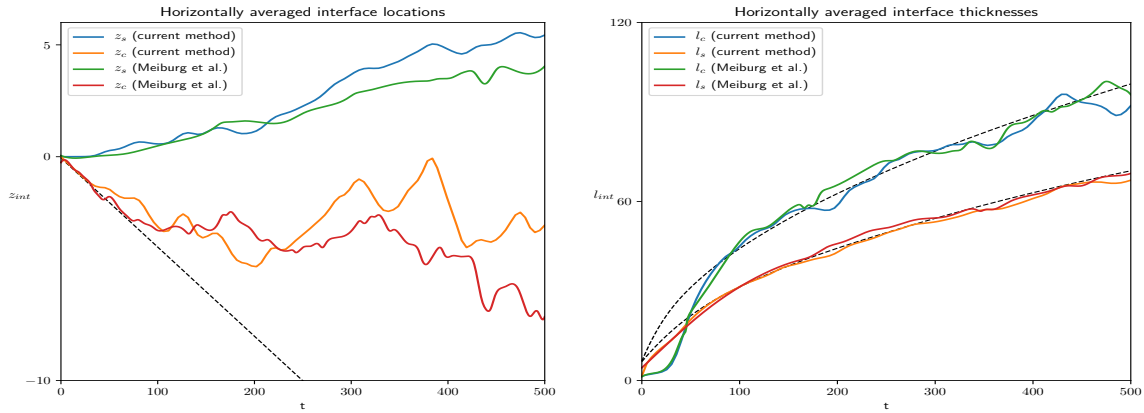


Figure 4.5 – Snapshots of horizontally averaged sediment and salt profiles

Quantitative comparison

Figure 4.5 shows that the horizontally averaged salinity profile shares many features with the horizontally averaged sediment profile. The salinity profile is much smoother due to the faster diffusion of salinity and the results of the error function fitting procedure are much closer to the original curves than the ones of obtained from averaged sediment concentrations. The evolution of fitted average interface locations $(z_c(t), z_s(t))$ and interface thicknesses $(l_c(t), l_s(t))$ is shown on figure 4.6 and compared to reference results. The fitting procedure used to estimate those parameters is not exactly the same in the reference data because the weighting procedure used in [Burns et al. 2015] is not described. The parameter that is the most sensible to the optimization procedure for a given scalar is the interface location and the method employed to obtain reference interface locations seems to be much more sensible than ours (data has been smoothed out in the data extraction process, see figure 8 of [Burns et al. 2015] for original curves). The two runs also differ in the random displacement of the initial interface determined by $\delta(x)$ in equation 4.6. As stated earlier, we use a uniform distribution of the form $\delta(x) = 10^{-1}\, l_0\, [\mathcal{U}(0,1) - 0.5]$ which may be different in amplitude than the one used to obtain reference data.



(a) Averaged interface locations **(b)** Averaged interface thicknesses

Figure 4.6 – Evolution of horizontally averaged interface locations and thicknesses

Qualitative comparison

Figures 4.7 and 4.8 show the sediment and salinity fields at four different times $t \in \{40, 100, 200, 300\}$, top-to-bottom. The domain has been truncated in the vertical direction $z \in [-300, 300]$ so that our results can be compared to reference data available from [Burns et al. 2015], figure 4. As it can be seen on the top frames, the interface is initially displaced by a random perturbation $\delta(x)$ but develops a definite preferred wavelength during the early development of the instability. Overall our solver seems to capture the same physical scales and interface displacements as the reference one. Here the instability is clearly double-diffusive with the emergence of fingers at the interface. As for the reference solution, we observe both positively and negatively buoyant plumes that detach from the interfaces at $t = 200$ and eventually collide around $t = 300$.

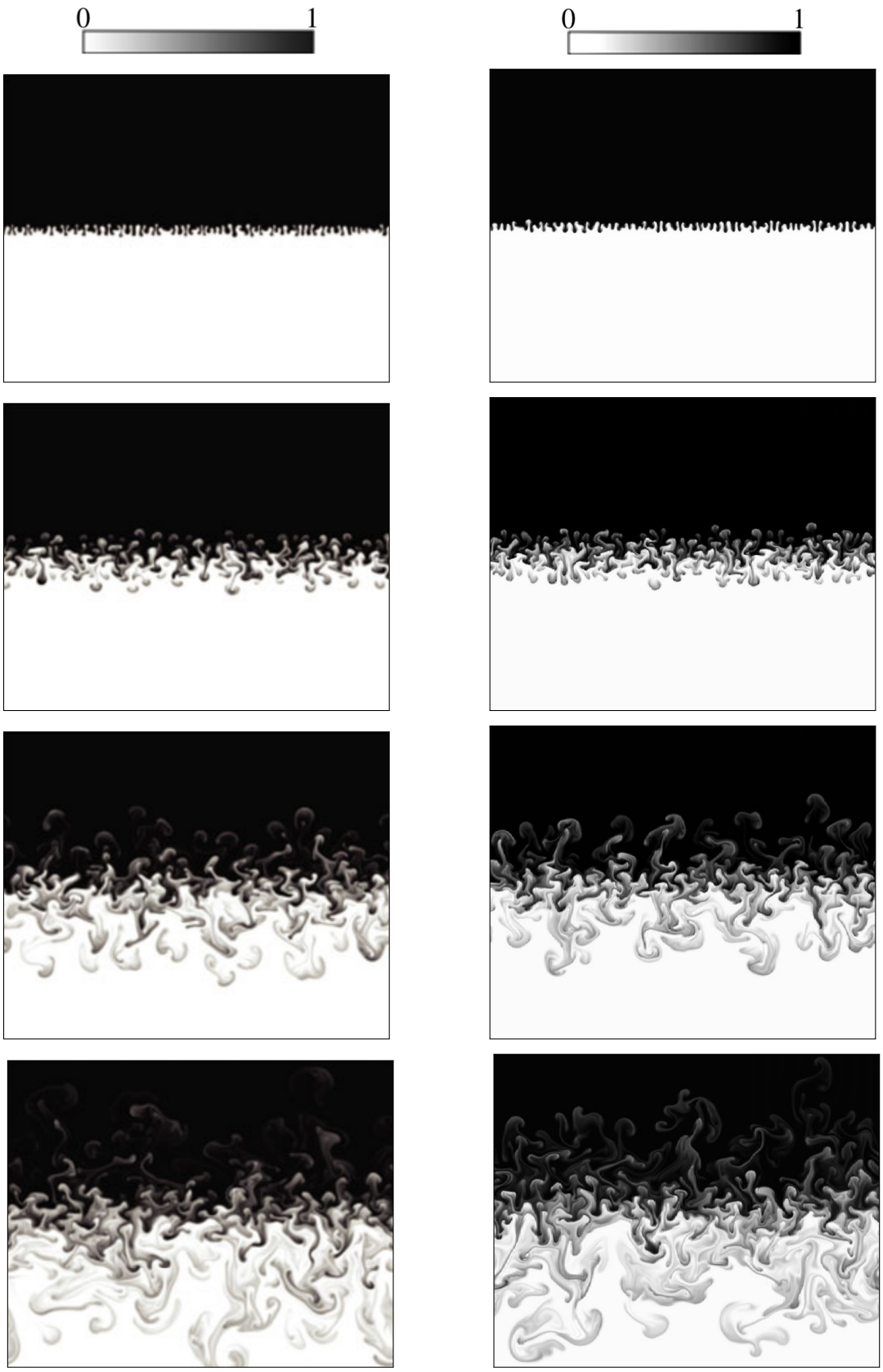


Figure 4.7 – Comparison of 2D sediment slices with reference data: $C(x, z, t)$ for $x \in [0, 750]$, $z \in [-300, 300]$ and $t \in \{40, 100, 200, 300\}$. Left slices corresponds to [Burns et al. 2015] and right slices to our results. Parameters: $V_p = 0.04$, $Sc = 0.7$, $R_s = 2.0$, $\tau = 25$.

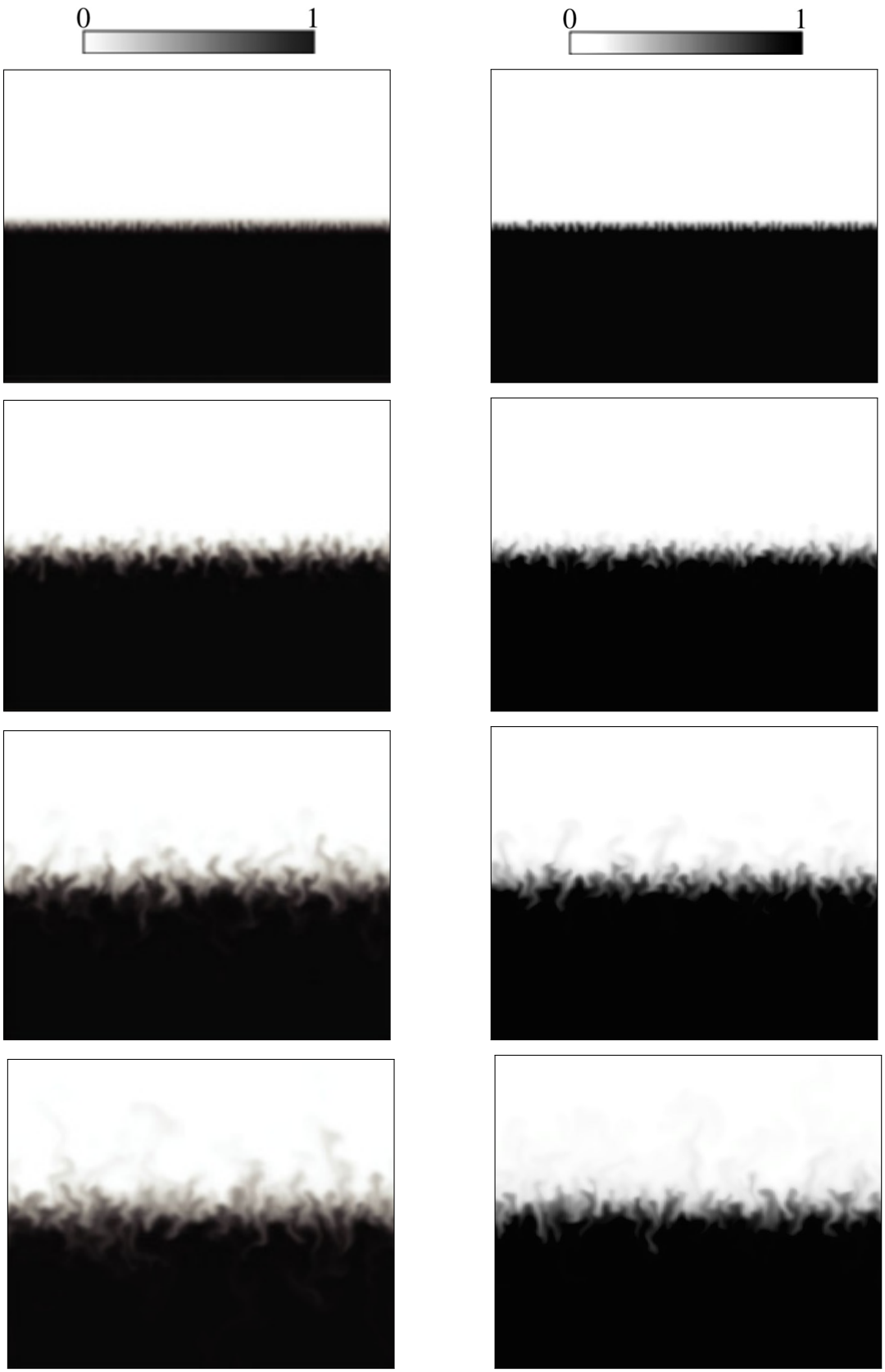


Figure 4.8 – Comparison of 2D salt slices with reference data: $S(x, z, t)$ for $x \in [0, 750]$, $z \in [-300, 300]$ and $t \in \{40, 100, 200, 300\}$. Left slices corresponds to [Burns et al. 2015] and right slices to our results. Parameters: $V_p = 0.04$, $Sc = 0.7$, $R_s = 2.0$, $\tau = 25$.

When the simulation starts, it can be clearly seen that the sediment interface begins to move downward with velocity V_p as a result of initial particle settling (orange and red curves on figure 4.6). Initial sediment interface location prediction $-V_p t$ is shown as a dashed line and after $t_0 = 40$ double-diffusive processes take over. Before t_0 we also observe that the initial salinity interface does not move much (it only diffuses upwards at rate $1/S_c$). After t_0, both the sediment and salinity interface thicknesses can be fit to $\sqrt{t}$ diffusive profiles (represented with dashed lines) and we obtain roughly the same profiles as in the reference data. In this configuration, the turbulent diffusion resulting from the double-diffusivity instability along with the settling velocity of sediments has modified the sediments evolution from diffusing $\tau = 25$ times more slowly than salinity, to diffusing close to twice as fast.

Finally we can compare the ratio of the nose region height $H(t) = z_s(t) - z_c(t)$ to the salinity interface thickness $l_s(t)$. It is expected from [Burns et al. 2012] that once is fully developed (after t_0), the ratio H/l_s remains roughly constant. This is effectively what we get in practice, with a ratio $R_t(t)$ that is in agreement to the one obtained in the reference data. The fluctuations are mainly due to the differences in the estimation of interface locations $z_c(t)$ and $z_s(t)$. Those fluctuations can be smoothed out by computing an ensemble average over multiple runs with different initial random perturbations (see section 4.3.1).

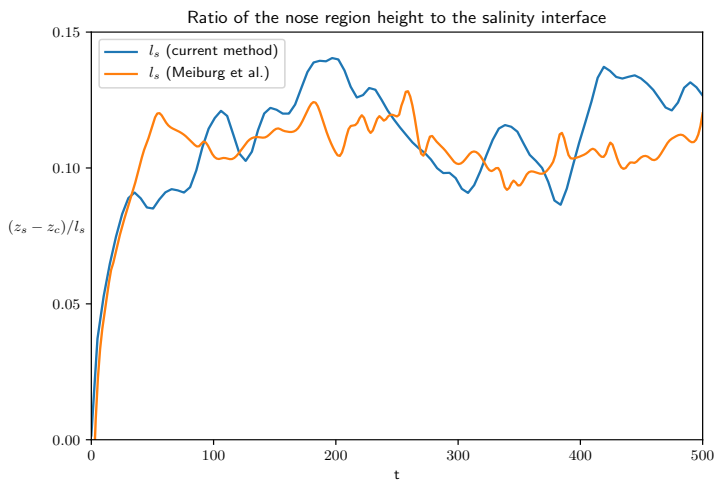


Figure 4.9 – Evolution of the ratio of the nose region height to the salinity interface thickness

4.2.6 Comparison with three-dimensional reference solution

The second reference simulation is also obtained from [Burns et al. 2015] and features the following configuration:

- $t \in [t_{start}, t_{end}] = [0, 150]$
- $\Omega = [z_{min}, z_{max}] \times [y_{min}, y_{max}[\times[x_{min}, x_{max}[= [-110, +65] \times [0, 100[\times[0, 100[$
- $S_c = 7, \tau = 25, V_p = 0.04, R_s = 2$

It is discretized over a grid of size $\mathcal{N}^v = (3n + 1, n, n)$ with $n = 512$.

This moderate Schmidt number configuration will highlight different flow features. Note that the domain is now much smaller so that the space step allows to simulate this ten-fold increase in the Schmidt number. Our simulation is carried out on a slightly bigger domain then the one of the reference simulation. We use $\Omega = [z_{min}, z_{max}] \times [y_{min}, y_{max}[\times [x_{min}, x_{max}[= [-128, +64] \times [0, 128[\times [0, 128[$ so that the space step is $\boldsymbol{dx} = (dz, dy, dx) = (0.125, 0.25, 0.25)$.

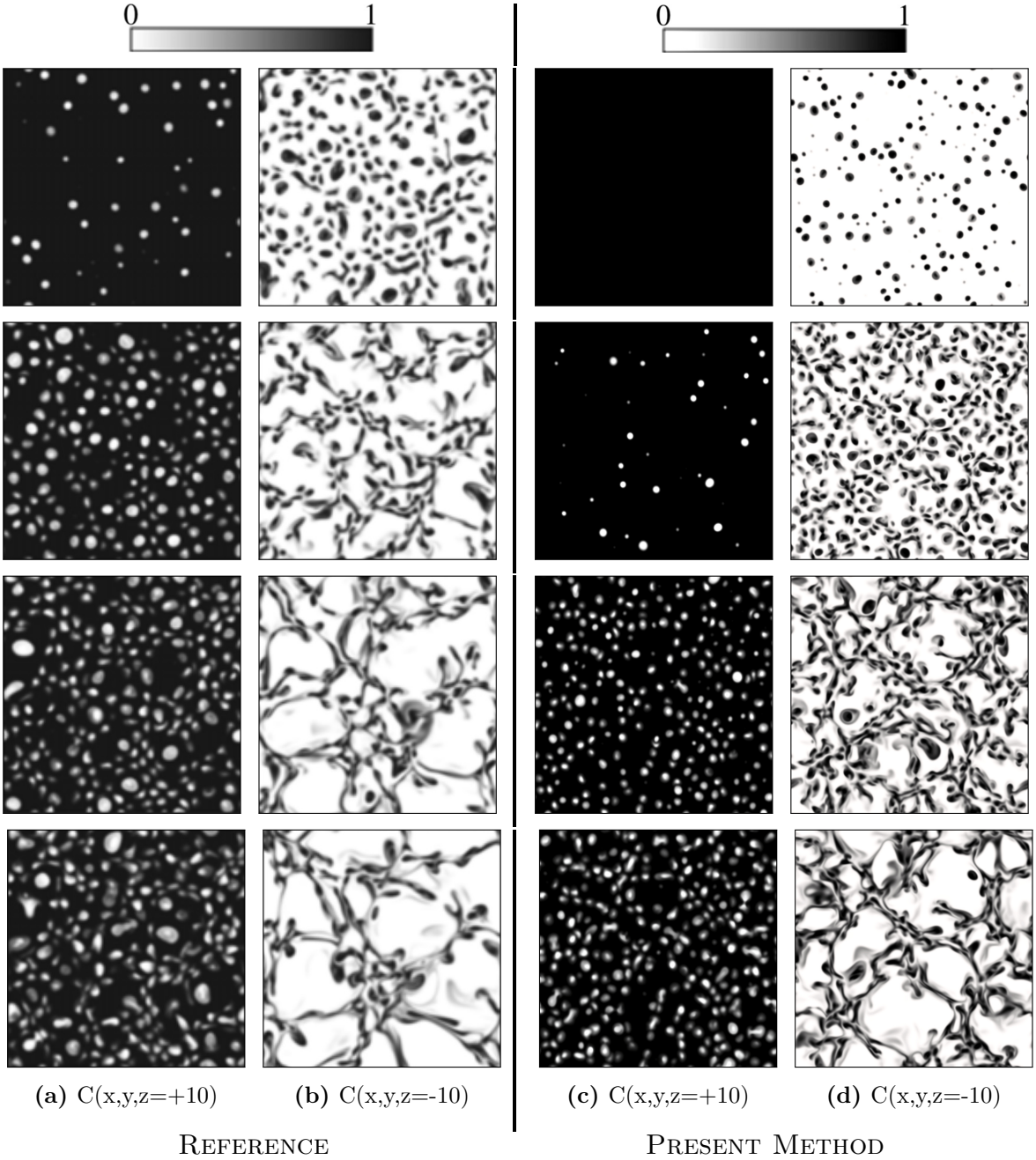
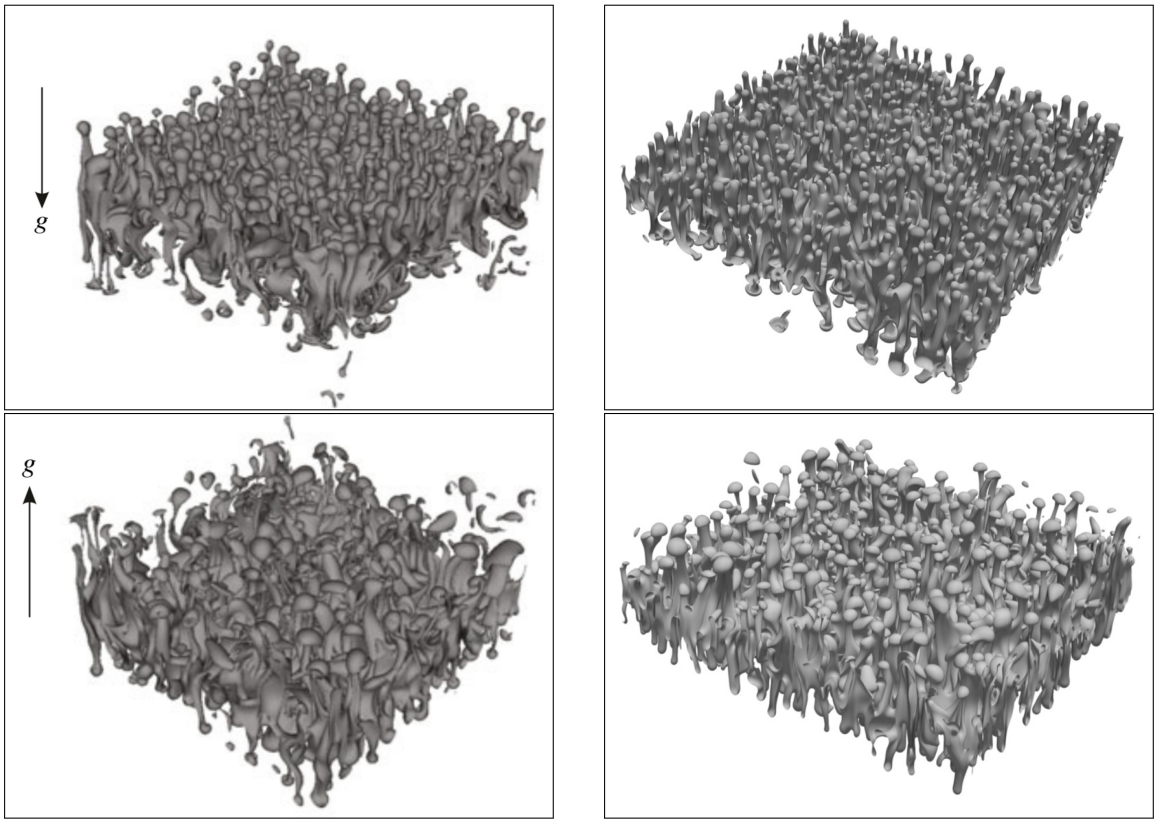


Figure 4.10 – Comparison of 3D sediment slices with reference data: $C(x, y, z, t)$ for $(x, y, z) \in [0, L_x[\times [0, L_y[\times \{-10, +10\}$, and $t \in \{75, 100, 125, 150\}$. Left slices corresponds to reference and right slices to our results. Parameters: $V_p = 0.04$, $Sc = 7$, $R_s = 2.0$, $\tau = 25$.

Figure 4.10 shows four different snapshots of horizontal slices of the sediment concentration, above an below the initial interface position at $z = \pm 10$. Left reference slices represent a dimensionless physical surface of size $[0, 100[^2$ while right slices corresponds to our results obtained on slightly larger surfaces of size $[0, 128[^2$. Above the interface at $z = +10$, the plots show circular spots representing the upward moving salt fingers. The only change in this structure is that more and more fingers appear with time. The first frame at $z = -10$ and $t = 75$ show downward moving sediment fingers that are similar in size, although larger in count, when compared to the upward moving salt fingers at the same period. After $t = 100$, sediment fingers are no longer circular but stretched by local convection zones. This is suggests that Rayleigh-Taylor instabilities dominate in the lower region of the flow. It can be better seen on the lower isocontours of the sediment concentration $C = -1.5$ at $t = 100$ that are shown on figure 4.11. Exactly as for [Burns et al. 2015] we observe that the convection zones become larger with time, yielding polygon-like shapes. Overall we observe the same structures as in the reference data, but our results seem to be lagging behind with a delay of $\Delta_t = 15$ (dimensionless time). It is believed that the amplitude of the initial perturbation $\delta(x, y)$ is the cause of such a difference. Here we use the same initialization as for the 2D case, $\delta(x, y)$ as $p_0\, l_0\, [\mathcal{U}(0, 1) - 0.5]$ with $p_0 = 0.1$ whereas for the reference data δ has not been specified.



(a) Reference (b) Present Method

Figure 4.11 – Comparison of 3D sediment isocontours with reference data: contours $C(x, y, z) = 0.5$ at $t = 100$ showing upward (top) and downward (down) moving fingers.

4.3 High performance computing for sediment flows

In this section we explore the performance and limitations of our solver with respect to the sediment-laden fresh water above salt water problem.

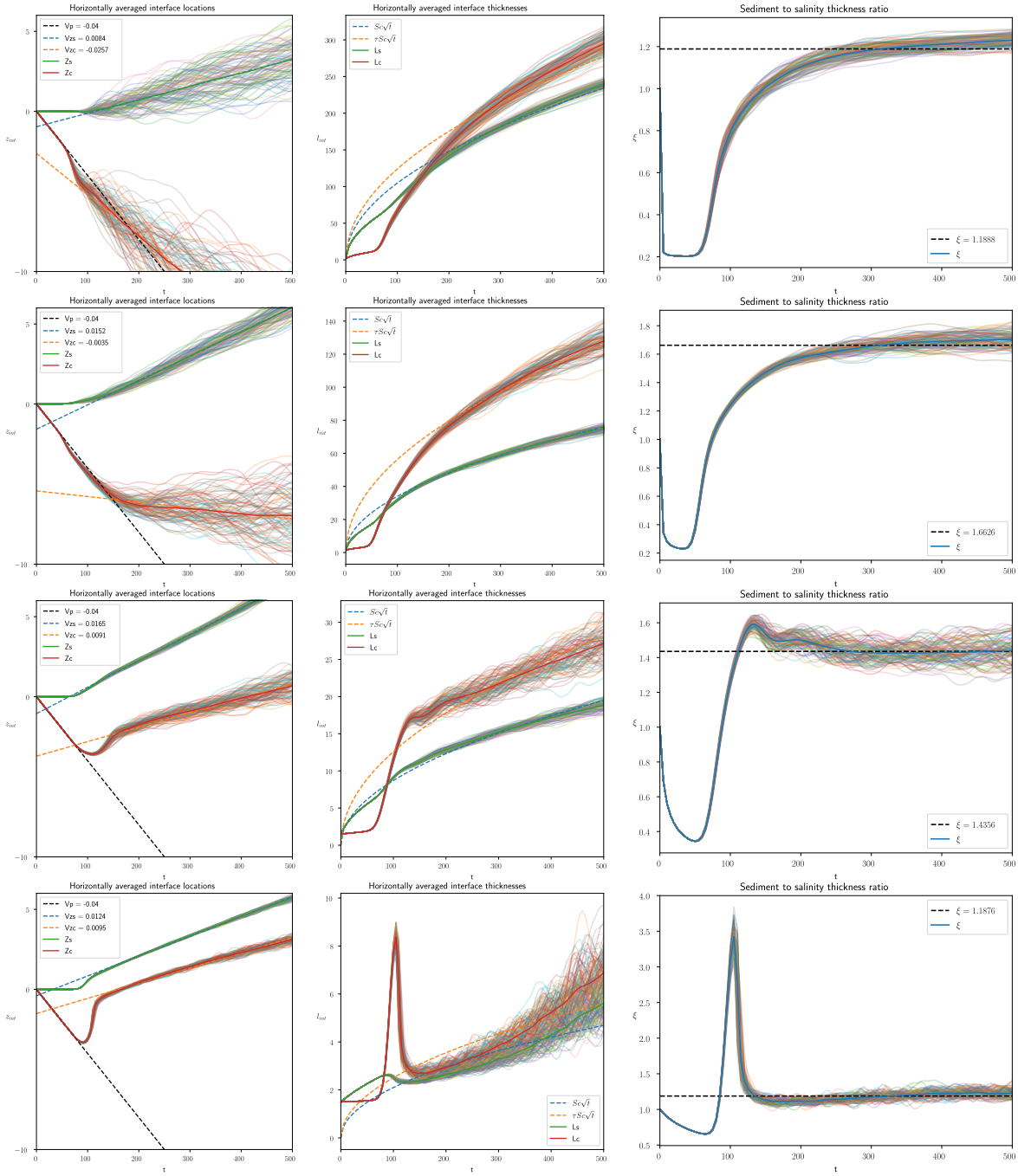
4.3.1 Ensemble averaged two-dimensional simulations

The two reference simulations with $S_c = 0.7$ and $Sc = 7$ have already shown the influence of the Schmidt number on instabilities. For small Schmidt numbers, molecular diffusion is high and the settling velocity is too low to have a real effect on the evolution of the interfaces in which case double diffusion dominates. Larger Schmidt number values result in a sharper salinity interface and the double-diffusive mode is replaced by Rayleigh-Taylor instabilities. We cannot however draw any conclusions from a single numerical simulation. The idea of [Burns et al. 2015] is to perform a parametric study over 170 simulations with various configurations of the dimensionless parameters of this problem (V_p, R_s, τ, S_c). Those simulations each yield horizontally averaged flows quantities that are ensemble-averaged over 10 runs to see the effect of the parameters on the H/l_s ratio that determines the type of instabilities. This represents an embarrassingly parallel problem that is a first application to our high performance solver.

To illustrate this, we run four different two-dimensional configurations from $Sc = 0.07$ to $Sc = 70.0$ and compute ensemble-averaged parameters every $\Delta_t = 5$ over 100 runs. It takes $227s$ to compute a case from $t = 0$ to 500 with $dt_{\texttt{MAX}} = 1$, discretized on $(n+1, n)$ points with $n = 4096$ on a single `Nvidia Tesla V100 GPU`. Those four different ensemble-averaged cases thus run in approximately 6 hours and 20 minutes on a single compute node containing four of such `GPUs`. Samples of obtained statistics are shown on figure 4.12 where the thick lines represent ensemble averaged statistics. Estimated interface velocities and diffusivities as well as median thickness ratios and nose region height to salinity thickness ratios are shown in table 4.1 for $n = 2048$ and $n = 4096$. It can be seen that the cases $Sc = 7$ and $Sc = 70$ do no converge to the expected solution because of insufficient spatial discretization.

	$n = 2048$					
S_c	V_{zs}	V_{zc}	$\overline{S_c}$	$\bar{\tau}$	$\bar{\xi}$	$\overline{R_t}$
0.07	9.27×10^{-3}	-2.45×10^{-2}	1.07×10^{1}	1.15×10^{0}	1.16×10^{0}	6.71×10^{-2}
0.7	1.57×10^{-2}	-3.24×10^{-3}	3.39×10^{0}	1.65×10^{0}	1.67×10^{0}	1.63×10^{-1}
7.0	1.71×10^{-2}	9.40×10^{-3}	8.64×10^{-1}	1.45×10^{0}	1.43×10^{0}	3.24×10^{-1}
70.0	1.22×10^{-2}	9.09×10^{-3}	2.17×10^{-1}	1.21×10^{0}	1.21×10^{0}	5.68×10^{-1}
S_c	$n = 4096$					
0.07	8.84×10^{-3}	-2.57×10^{-2}	1.05×10^{1}	1.18×10^{0}	1.19×10^{0}	6.66×10^{-2}
0.7	1.52×10^{-2}	-3.52×10^{-3}	3.40×10^{0}	1.66×10^{0}	1.66×10^{0}	1.62×10^{-1}
7.0	1.65×10^{-2}	9.08×10^{-3}	8.70×10^{-1}	1.45×10^{0}	1.44×10^{0}	3.22×10^{-1}
70.0	1.24×10^{-2}	9.48×10^{-3}	2.10×10^{-1}	1.19×10^{0}	1.19×10^{0}	5.88×10^{-1}

Table 4.1 – Ensemble averaged statistics for different Schmidt numbers. Parameters are averaged over 100 runs for $V_p = 0.04, R_s = 2, \tau = 25$ and $S_c \in \{0.07, 0.07, 7, 70\}$.



(a) Interface locations **(b)** Interface thicknesses **(c)** Thickness ratio

Figure 4.12 – Ensemble averaged quantities at varying Schmidt number of 2D flows, averaged over 100 runs with $\mathcal{N}^v = (4097, 4096)$ from $Sc = 0.07$ (top) to $Sc = 70$ (bottom) on domain $\Omega = [-1024, 1024]^2$. With this discretization, the two highest Schmidt configurations are under-resolved, $\boldsymbol{dx} = (0.5, 0.5)$, and result in positive sediment interface velocities. Parameters: $V_p = 0.04$, $Sc = \{0.07, 0.7, 7, 70\}$, $R_s = 2.0$, $\tau = 25$.

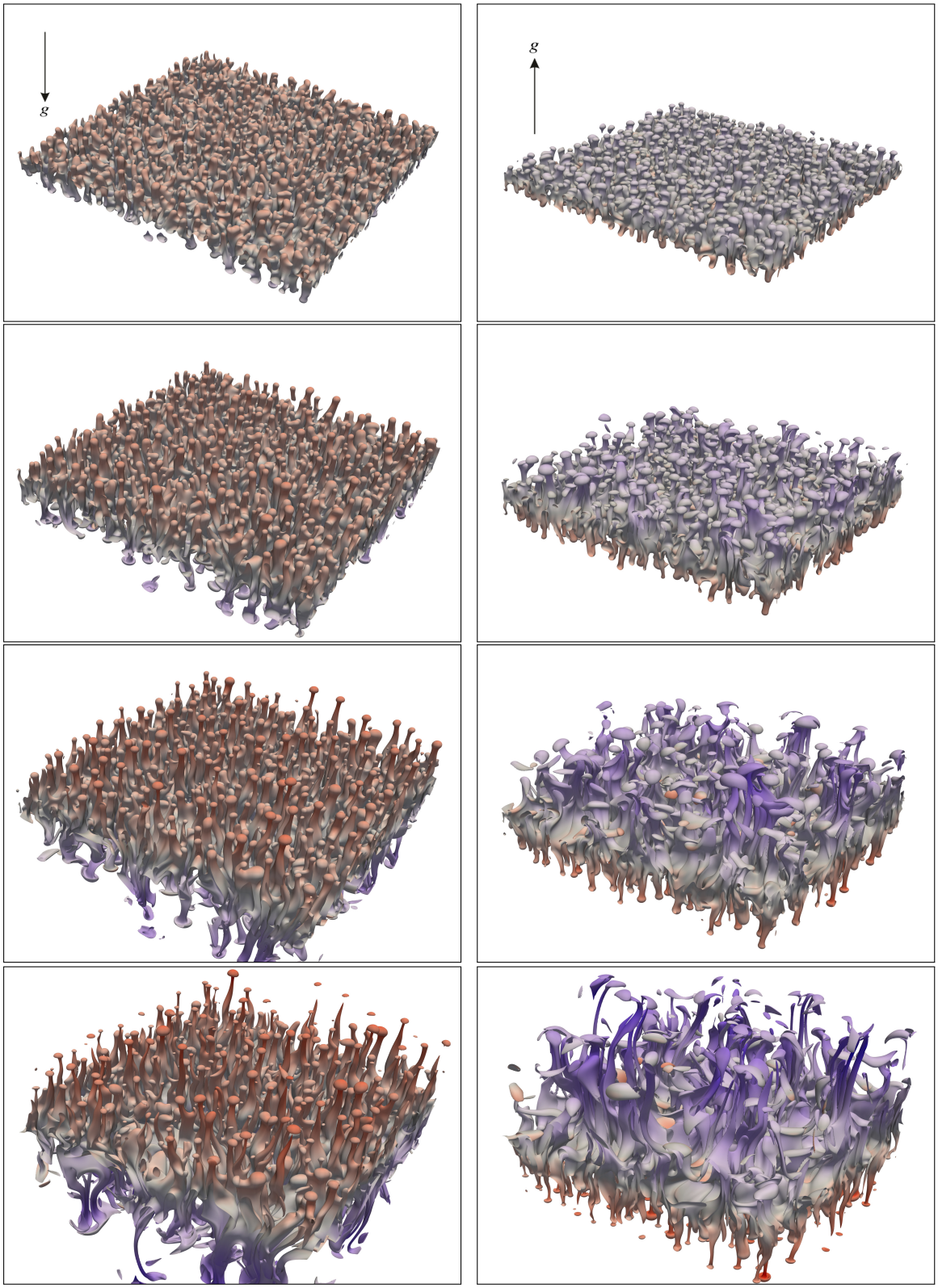


Figure 4.13 – 3D sediment isocontours at moderate Schmidt number $S_c = 7$: Plot of the contours $C(x, y, z) = 0.5$ for $t \in \{75, 100, 125, 150\}$ (top-to-bottom) showing upward (left) and downward (right) moving sediment fingers. Color represent vertical velocity. Parameters: $V_p = 0.04$, $Sc = 7$, $R_s = 2.0$, $\tau = 25$.

4.3.2 Three-dimensional simulations

On a GPU with 32GB of memory, our implementation is able to solve single-precision two-dimensional problems with discretizations up to $(32768, 16384)$ and three-dimensional configurations up to $(1537, 512, 512)$. On an Nvidia Tesla V100, the three-dimensional problem is memory-bound and can be solved at a rate of $10.9s$ per iteration, including the computation of the statistics and timestep criterias. A compute-node with 256GB of memory and a sufficient number of CPUs can handle a simulation of size up to $(3073, 1024, 1024)$. In this case the problem is compute-bound and it takes around 24 minutes per iteration on a quad socket Intel Xeon E7-8860 v4 platform when using the Intel MKL-FFT library to perform the spectral transforms. The total number of iterations required to solve a typical three-dimensional problem from $t = 0$ to 150 ranges from 300 to 2000 depending on the setting of $dt_{\mathtt{MAX}}$ and imposed CFL constant. With $dt_{\mathtt{MAX}} = 0.1$ and $C_{\mathtt{CFL}} = 1$, the $1537 \times 512 \times 512$ simulation takes under 6 hours on the aforementioned GPU device and can be used for Schmidt numbers up to $Sc = 28$ on the three-dimensional domain $\Omega = [-128, 64] \times [0, 128] \times [0, 128]$.

The resulting isocontours of sediment concentration obtained with the GPU setup and $S_c = 7$ are shown on figure 4.13. This specific configuration is at the interface between the two kind of instabilities we may encounter: double-diffusion and Rayleigh-Taylor instabilities. Before $t = 100$ we can clearly see sediment fingers moving both upwards and downwards, result of a double-diffusive process. After $t = 100$, the instabilities become more and more Rayleigh-Taylor like. As introduced in section 1.3.2 the dominant type of instability can be deduced from the knowledge of the particle settling velocity and the diffusive spreading velocity of the salinity layer. More precisely, the key parameter that determines whether the settling process is dominated by double-diffusive or Rayleigh-Taylor instabilities is the ratio of the nose region height to the salinity thickness $R_t = H/l_s$. By performing a parametric study over 15 different two-dimensional configurations, [Burns et al. 2015] showed that the value of the nose thickness ratio R_t could be predicted a priori based on the knowledge of the dimensionless grouping $R_s V_p \sqrt{S_c}$. Because there currently exist no three-dimensional reference data for Schmidt numbers higher than 7, we propose to validate our implementation to higher Schmidt numbers by checking the fit between the predicted values of H/l_s and obtained results. Table 4.2 (A-B-C-D) shows the two-dimensional configurations used by Burns and its collaborators to perform their parametric study. We introduce a new dimensionless group (E) that characterizes our three-dimensional test cases. We explore Schmidt numbers ranging from $S_c = 3.5$ to $S_c = 28.0$ while other dimensionless parameters are kept constant.

Case	τ	S_c	R_s	V_p	L_x	L_y	L_z	N_x	N_y	N_z
A	25	0.7	2.0	(0.02, 0.04, 0.08, 0.16)	750	–	600	1536	–	4097
B	25	7.0	2.0	(0.01, 0.02, 0.04, 0.08)	300	–	250	2048	–	4097
C	25	70.0	2.0	(0.01, 0.02)	125	–	100	2048	–	4097
D	25	0.7	(1.1, 1.5, 2.0, 4.0, 8.0)	0.04	750	–	600	1536	–	4097
E	25	(3.5, 7.0, 14.0, 28.0)	2.0	0.04	128	128	256	512	512	1537

Table 4.2 – Dimensionless group values and grid properties for the parametric study. The first four classes of parameters (A,B,C,D) corresponds to the 15 two-dimensional setups of [Burns et al. 2015] that have non-vanishing settling velocities. The last class (E) corresponds to our three-dimensional cases with increasing Schmidt numbers up to $Sc = 28$.

For small values of $R_s V_p \sqrt{S_c}$ (small settling velocity or big salinity diffusivity) the sediment inflow into the nose region from above is small and double-diffusive fingering compensate with an equal sediment outflow from within the nose region (H/l_s remains small). For large values of $R_s V_p \sqrt{S_c}$ (large settling velocity or small salinity diffusivity), the sediment inflow into the nose region from above is high and the nose region height H grows faster than the salinity layer thickness l_s. As sediments accumulate in the nose region, H/l_s grows until achieving a critical value where Rayleigh-Taylor instabilities kicks in, releasing downward sediments plumes. The transition between small and large values has been identified as $R_s V_p \sqrt{S_c} = \mathcal{O}(0.1)$.

Our four three-dimensional test cases (E) are run simultaneously on a single compute node containing four `GPUs` and give the following results:

τ	R_s	V_p	S_c	$H(t=150)$	$l_s(t=150)$	$R_s V_p \sqrt{S_c}$	$R_t = H/l_s$
25	2.0	0.04	3.5	4.57	18.8	0.150	0.243
25	2.0	0.04	7.0	3.99	12.8	0.212	0.312
25	2.0	0.04	14.0	3.66	7.34	0.299	0.499
25	2.0	0.04	28.0	2.14	3.63	0.423	0.590

Table 4.3 – Obtained nose region ratio for three-dimensional test cases (E).

We compare those results with the *a priori* prediction of the nose thickness ratio from [Burns et al. 2015]. As seen on figure 4.14, all the four predictions are in agreement with our results.

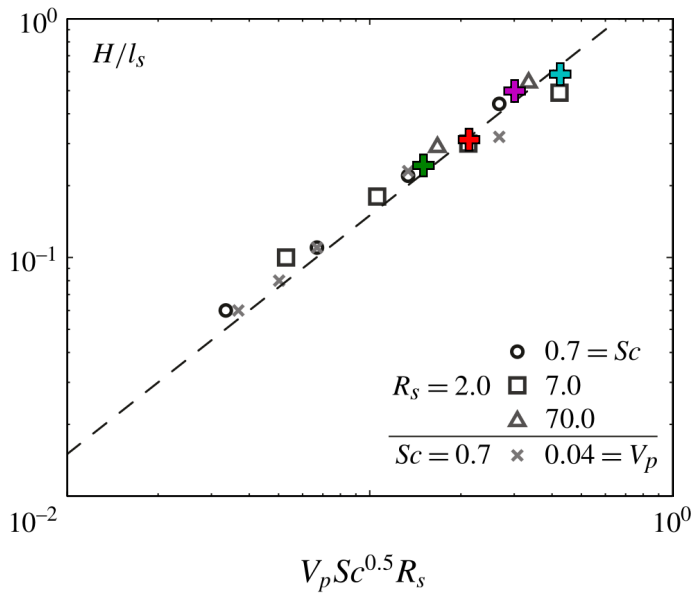


Figure 4.14 – Prediction of the nose thickness ratio H/l_s versus obtained results. Dots, squares, triangles and crosses correspond to the two-dimensional results obtained by [Burns et al. 2015] with parameters groups (A,B,C,D) defined in table 4.2. The superimposed plus signs correspond to our three-dimensional results obtained for parameter group (E), from smallest Schmidt number (left, $Sc = 3.5$) to highest Schmidt number (right, $Sc = 28.0$).

The sediment concentration isocontours corresponding to averaged quantities from table 4.3 are shown on figure 4.16. As expected, with increasing Schmidt number, we see the transition from a double-diffusive fingering behaviour ($R_s V_p \sqrt{S_c} = 0.15$) to a Rayleigh-Taylor dominated configuration ($R_s V_p \sqrt{S_c} = 0.42$). The two lowest Schmidt number configurations feature salt fingers initially present on both sides of the interface that are eventually suppressed by Rayleigh-Taylor instabilities after $t = 100$. For the two highest Schmidt number configurations no more sediment fingers can be seen. Instead, we can see downward moving sediment plumes that have a mushroom like structure characteristic of Rayleigh-Taylor instabilities.

4.3.3 Achieving higher Schmidt numbers

Achieving a Schmidt number of 100, an order of magnitude above the grid-converged simulation at $Sc = 24$, will require a discretization of at least $(3073, 1024, 1024)$. This would be too costly for a single compute-node and too big too fit into a single `GPU` memory. Some memory could be saved by using the multiscale approach where only the scalars are discretized on a fine grid of size $(3073, 1024, 1024)$ while velocity and vorticity are discretized on a coarser grid of size $(1537, 512, 512)$. It is however not possible to perform this simulation on a `GPU` with only `32GB` of `RAM` because a single field discretized on the fine grid with single-precision would take up to `13GB` of memory.

We recall that when using a multiscale approach, the ratio between the scalar and velocity grids can be initially computed as the square root of the respective Schmidt numbers (see section 1.1.10). For $\tau = 25$ and $Sc = 100$ we have $\sqrt{\tau S_c} \simeq 50$. This indicates that we could use grid ratios up to 50 in each direction between the velocity and sediment concentration grid (and 10 for the salinity grid). To illustrate this point, the different physical scales obtained for a problem with $Sc = 7$ and $\tau = 25$ are shown on figure 4.15

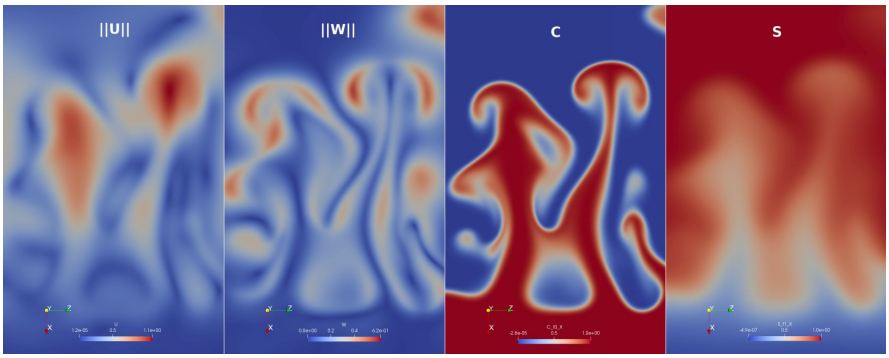
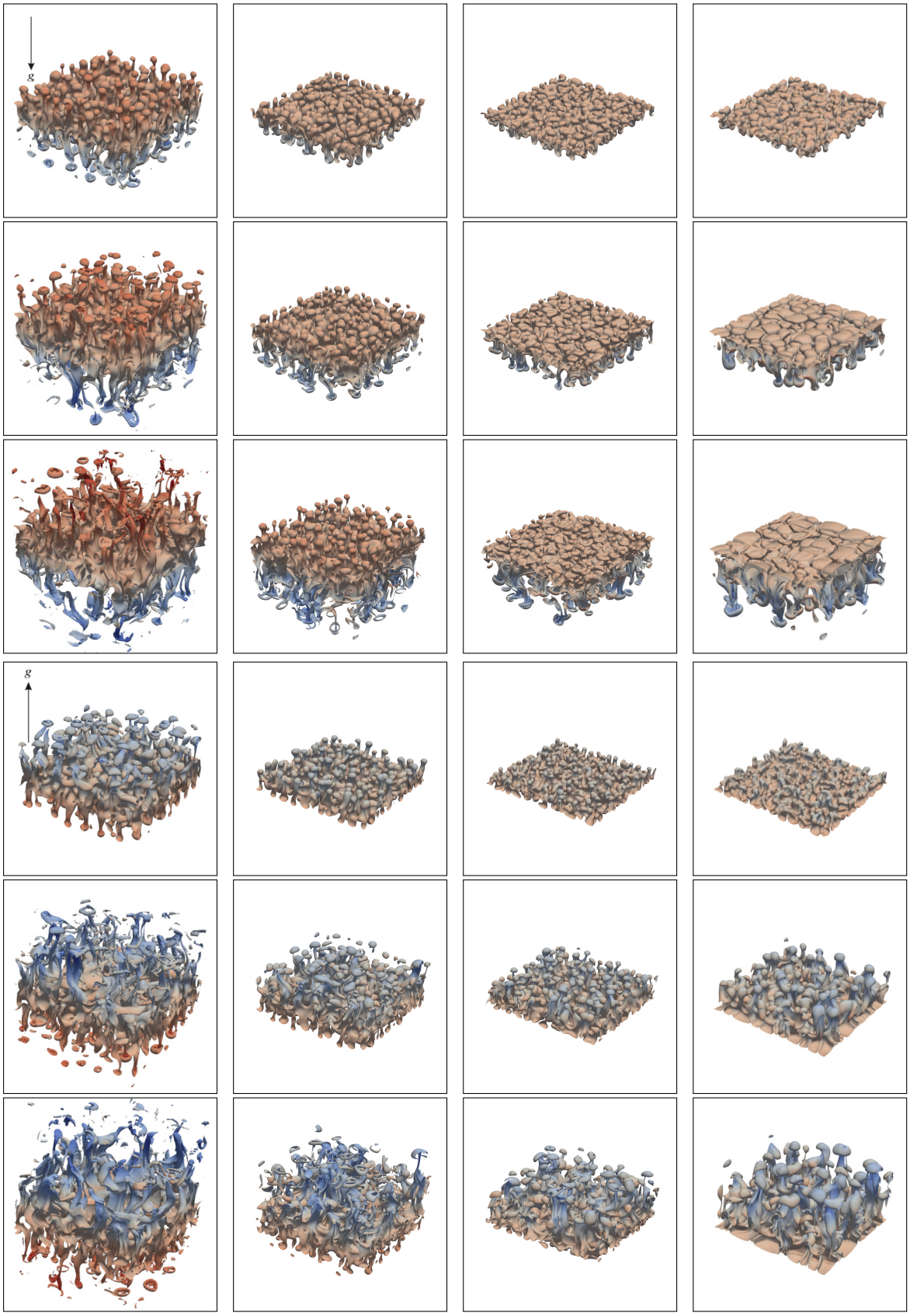


Figure 4.15 – Example of the physical scales obtained with $Sc = 7$ and $\tau = 25$.

In practice to simplify the interpolation and restriction steps, we restrict ourselves to power of two grid ratios $(2, 4, 8, 16, \cdots)$. A two- and three- dimensional convergence study with fixed initial displacement δ has shown that, when the flow is sufficiently resolved, such a multiscale approach worked well with linear interpolation up to a grid ratio of 2. Higher grid ratios required at least cubic polynomial interpolation.



(a) $Sc = 3.5$ **(b)** $Sc = 7.0$ **(c)** $Sc = 14.0$ **(d)** $Sc = 28.0$

Figure 4.16 – 3D sediment isocontours for varying Schmidt number: Plot of the contours $C(x, y, z) = 0.5$ for $t \in \{100, 125, 150\}$ showing upward (top) and downward (bottom) moving fingers at $n = 512$. Parameters: $V_p = 0.04$, $Sc \in \{3.5, 7, 14, 28\}$, $R_s = 2.0$, $\tau = 25$.

Currently the only limitation of our implementation relies in the fact that we perform in-core computations on single GPUs with rather limited memory (32GB per GPU). This means that the problem has to fit in the embedded GPU memory (VRAM). On the contrary CPU devices can directly use the main memory (RAM) which is usually larger (currently, 128GB to 512GB per compute node is rather common) but offer far lower floating-point compute capabilities (see table 1.2).

Hence, a solution to achieve higher Schmidt numbers on accelerators having limited memory capabilities is to perform out-of-core simulations by storing the whole problem in the host memory instead of device memory. The idea is then to overlap compute kernels, device-to-host and host-to-device memory transfers by using multiple OpenCL command queues [Van Werkhoven et al. 2014]. The second solution is to distribute the computation across multiple GPUs by using domain decomposition. Ongoing developments efforts are thus focused on distributed spectral operators and support for out-of-core computations.

Conclusion

This chapter was dedicated to the simulation of particle-laden clear water above salt-water. The full physical model has first been introduced, followed by the specification of initial and boundary conditions. The computational domain consists in a box, compatible with Cartesian grid discretization. The use of the remeshed particle method make it possible to discretize the sediment concentration and salinity on a finer grid than that of the flow variables, without impacting the timestep. An algorithm to solve this problem within our framework has been proposed along with associated timestep criterias. Homogeneous boundary conditions are handled by using the spectral solver introduced in section 3.2.6. The horizontally averaged flow characteristics of interest in this kind of problem have been detailed. In the end, the whole simulation, including the computation of averaged quantities, can run exclusively on the OpenCL compute backend.

The proposed implementation was run for two- and three-dimensional configurations and compared to reference solutions available from state of the art numerical simulations. Obtained numerical results were in agreement both qualitatively and quantitatively. We then showed that within our framework, the performance delivered by GPUs allowed us to compute large ensemble averages of parameters characterizing the flow. Subsequent three-dimensional runs for for Schmidt numbers comprised between $S_c = 3.5$ and $S_c = 24$ showed the transition from a double-diffusive fingering behaviour to a Rayleigh-Taylor dominated configuration. Because all other dimensionless parameters were kept constant between the different cases, this was the expected behaviour and our results all fit the prediction of the nose thickness ratio. We thus demonstrated that it was possible to simulate flows at Schmidt numbers while keeping reasonable compute times on a single GPU. The ultimate goal being to understand vertical sediment transport with Schmidt numbers encountered in the nature ($Sc = 1750$), we finally developed strategies to increase the simulated Schmidt number. While the results obtained on a single GPU looks very promising, going even higher in Schmidt number will require to distribute the computations on multiple GPUs.

Conclusion and perspectives

General conclusion

The main objective of this work was to achieve high Schmidt number sediment-laden flow simulations by using adapted numerical methods in a high performance computing context.

The physics of this kind of problem is characterized by the presence of several phenomena at different physical scales that is due to the large difference in the diffusivities of the transported components. Those type of flows are more commonly referred to as high Schmidt number flows. The need for high performance computing and adapted numerical methods was motivated by the fact that such flows are very demanding in terms of computing ressources. Achieving high Schmidt numbers is the key to be able to numerically investigate the settling of fine particles arising from sediment-laden riverine outflows. The underlying numerical simulations are designed to identify the dominant sediment settling mechanisms which in turn can help to determine the location of sediment deposits on the sea floor. Based on the state of the art, this work proposes an efficient numerical method that is able to capture the dynamics of high Schmidt number flows in which there exist a two-way coupling between the transported quantities and the fluid.

A first chapter has been dedicated to a review of the underlying physical model and its derivation. The mathematical models required for the modeling of sediment flows have first been introduced, leading to the modelization of dilute monodisperse distribution of small particles as a continuum, that is, a particle concentration. The corresponding model was reduced to a coupling between the particles and the fluid, in an eulerian-eulerian framework. This model served as a base model for the physics of sediment-laden fresh water above salt water and was then further simplified by using a Boussinesq approximation. The final model consisted in a coupling between two scalars, representing salinity and sediment concentration, with the carrier fluid. The small diffusivity of fine sediments such as clay or fine silts was estimated by a semi-empirical model and led to Schmidt numbers between sediment and water as high as 17500. We saw that in such a flow, the slowly diffusive particles could develop smaller physical scales, up to a ratio depending on the square root of the Schmidt number, when compared to the smallest velocity eddies.

The second chapter was focused on the development the numerical method. This numerical method has been designed by starting from a state of the art numerical method adapted to the resolution of incompressible Navier-Stokes equations passively coupled with the transport of a scalar, at high Schmidt number. The key idea behind the proposed numerical method relies in the use of a semi-lagrangian (remeshed) particle method which benefits are twofold. Firstly, this method do not suffer from a timestep restriction depending on the grid size (CFL) but rather on velocity gradients (LCFL), reducing the timestep constraints associated to high Schmidt number flows. Secondly, the presence of an underlying grid allows to the use of

efficient eulerian solvers. The final proposed method consisted in an hybrid method based on semi-lagrangian particles, spectral methods and finite differences based methods. By relying heavily on operator splitting techniques, the method has been further adapted in order to become accelerator-friendly.

In the third chapter, the high performance implementation of the method has been described. Details about the development of associated numerical routines were given. Required numerical methods were implemented once for multiple different architectures by relying on the `OpenCL` standard. Performance-portability has been achieved by the way of code generation techniques associated to automatic runtime kernel performance tuning. Within the solver, the description of a numerical algorithm has been handled by building a graph of operators. The translation of finite differences based methods to `GPU` accelerated operators has been simplified by an `OpenCL` code generator based on symbolic expressions. An implementation of efficient `OpenCL` real-to-real transforms, required to handle homogeneous boundary conditions in spectral operators, has been proposed. A tool dedicated to the collection of `OpenCL` kernel instruction statistics has been implemented to evaluate the performance of the proposed numerical routines. For all kernels, achieved global memory bandwidth lied between 25 and 90% of peak device memory bandwidth. Two other levels of parallelism were also described: task parallelism that is automatically extracted from a directed acyclic graph of operators and domain decomposition. This implementation has been validated on a three-dimensional Taylor-Green vortex benchmark running exclusively on single `GPU` accelerator.

Chapter four has been devoted to the simulation of particle-laden clear water above saltwater. The problem to be solved as well as target averaged flow statistics were introduced. Reference solutions were obtained from state of the art two- and three-dimensional numerical simulations. Similar configurations were run with the proposed implementation by using a single `GPU`. Obtained numerical results were in agreement with the reference data both qualitatively and quantitatively. Even though no reference data exists for Schmidt numbers higher than 7, subsequent run at higher Schmidt numbers showed that the resulting flow was dominated by the expected type of instabilities, slowly passing from a double-diffusive fingering behaviour to Rayleigh-Taylor dominated settling as the Schmidt number increased. We thus demonstrated that it was possible to simulate Schmidt numbers four times higher than current state of the art simulations by using a single `GPU` accelerator. Distributing the computations on multiple `GPUs` will allow us to reach even higher Schmidt numbers while keeping reasonable compute times.

Perspectives

Many perspectives emerge from the numerical and applicative aspects of this work. First of all, since the spectral solvers have been rewritten from scratch to implement the support of homogeneous boundary conditions on accelerators, the distributed `MPI` implementation introduced in section 3.3.4 has not been implemented yet. This is the current blocking point for distributed multi-`GPU` simulations and constitute an essential step to develop the full potential of the proposed numerical method. Moreover, two performance bottlenecks have been identified in chapter 3.2: ghost exchanges and permutations kernels.

The roadmap to achieve higher Schmidt numbers within the `HySoP` library is the following:

1. Short term perspectives to achieve Schmidt numbers up to $Sc \simeq 100$:
   (a) Implement `CPU−GPU` data streaming to perform numerical simulations up to 8 `GPUs` on a single compute node. The problem will have to fit in the compute node memory while being sent back and forth through the `PCI` bus. While this will reduce the performance of memory-bound kernels, it will allow to compute bigger problems in terms of spatial resolution without having to wait for the next generation of `GPU` cards. Here the basic idea is to overlap kernels, device-to-host and host-to-device copies by using multiple command queues [Van Werkhoven et al. 2014].
   (b) This will also be the opportunity to implement hybrid `OpenCL-OpenCL` computing by using `CPU` and `GPU OpenCL` platforms at the same time. A simple microbenchmark could be used to determine the optimal compute granularity and the workload between `CPU` and `GPU` [Henry et al. 2014] [Aji et al. 2015].
2. Mid-term perspectives to achieve Schmidt numbers $Sc \simeq 400$:
   (a) Implement distributed spectral solvers and perform numerical simulations up to 64 `GPUs`. Here an approach similar to [Aji et al. 2012] and [Dalcin et al. 2019] should be preferred. This would yield three levels of parallelism by using hybrid `MPI-OpenCL-OpenCL` programming.
   (b) If possible, introduce better permutations kernels, based on open-source `CUDA` tensor-transpose libraries [Hynninen et al. 2017][Vedurada et al. 2018]. Determine preferred permutation axes by microbenchmarking prior to graph building.
   (c) Reduce the number of required ghost exchanges by integrating ghosts exchange to the graph analysis step presented in section 3.1.4.
3. Long-term perspectives to achieve Schmidt numbers $Sc \simeq 1600$ (512 `GPUs`):
   (a) Implement time sub-stepping for finite-difference based diffusion operators. Here the numerical method of choice would be `RKC`, a family of Runge-Kutta-Chebyshev formulas with a stability bound that is quadratic in the number of stages [Sommeijer et al. 1998].
   (b) Depending on the performance of the distributed spectral solver, implement an alternative solver based on an Aitken-Schwarz algorithm [Garbey et al. 2002][Baranger et al. 2003]. This algorithm trades the global permutations required by the spectral transforms with simple ghost exchanges, at the price of increased complexity.
   (c) Use the task parallelism provided by the graph analysis by using asynchronous capabilities of the `Python` language [Hunt 2019]. At this point, this would yield four levels of parallelism (`Task-MPI-OpenCL-OpenCL`). Simulations containing blocking `MPI` communications and disk `I/O` should largely benefit from this feature.

The `HySoP` library is planed to be open-sourced by the end of the year (2019). The short term perspectives listed here are currently a work in progress within the library.

Numerically speaking many interesting tracks could be explored, ranging from the comparison of the different interpolation and restriction methods in multiscale approaches to the use of higher order directional splitting schemes. When considering the applicative aspects of this work, many things come to mind because of the great flexibility of the numerical code developed in this work. Indeed, the `HySoP` library can be used as a high performance numerical toolbox for any fluid-related problem which resolution algorithm can be expressed in terms of operators. The library is modular and can be extended by implementing additional `Python`, `C++`, `Fortran`, or `OpenCL` operators.

If we stick to the original aim of this work, the next logical step is to extend the work of [Burns et al. 2015] by performing ensemble averaged three-dimensional simulations of increasing Schmidt numbers ($Sc > 28$). Those numerical results could eventually be used to identify the dominant settling mechanism of fine particles for Schmidt numbers encountered in the nature ($Sc = 1750$). While this application was focused on vertical convection, the study of horizontal convection at high Schmidt numbers is also of importance [Cohen et al. 2018][Konopliv et al. 2018]. The developed method could also be used to explore many other physical problems where transported quantities, diffusing at different rates, would be coupled. This is especially the case for chemical reactions happening in turbulent flows [Schwertfirm et al. 2010][Watanabe et al. 2015].

Acknowledgments

The author thanks Professor Eckart Meiburg and Peter Burns for their valuable inputs about the numerical aspects of their model. The author would also like to thank Jean-Matthieu Etancelin, Chloé Mimeau and Franck Pérignon for numerous helpful discussions around the `HySoP` library and its development. The author also thanks Delphine Depeyras, Matthieu Puyo, Quentin Desbonnets, Florent Braure and more generally the Ingeliance company (https://www.ingeliance.com) for their interest in this work and for providing additional financial support through the establishment of consulting duties (dispositif de doctorat-conseil).

This work has been supported by the `ANR MPARME` project funded by grant `ANR-17-CE23-0024-01` of the programme Investissements d'Avenir supervised by the Agence Nationale pour la Recherche. Some of the computations presented in this work have been performed using the `GRICAD` infrastructure (https://gricad.univ-grenoble-alpes.fr), which is partly supported by the `Equip@Meso` project funded by grant `ANR-10-EQPX-29-01`.

Appendix A

Mathematical notations

A.1 Function spaces

- $C^0(\Omega, \mathbb{K})$ is the spaces of continuous functions $f : \Omega \to \mathbb{K}$.
- $C^0_I(\Omega, \mathbb{K})$ is the space of piecewise continuous functions $f : \Omega \to \mathbb{K}$.
- **Spaces of differentiable functions:** Let $k \in \mathbb{N}^*$, we define the space of functions $f : \Omega \to \mathbb{K}$ that are k-th time differentiable as $D^k(\Omega, \mathbb{K})$. The class of infinitely differentiable functions $D^\infty(\Omega, \mathbb{K})$ s defined the intersection of the sets $D^k(\Omega, \mathbb{K})$ as k varies over the non-negative integers.
- **Spaces of piecewise continously differentiable functions:** Let $k \in \mathbb{N}^*$, we define the space consisting of functions $f \in D^k(\Omega, \mathbb{K})$ whose k-th derivative is piecewise continuous as $C^k_I(\Omega, \mathbb{K})$. The class of infinitely piecewise continuously differentiable functions $C^\infty_I(\Omega, \mathbb{K})$ s defined the intersection of the sets $C^k_I(\Omega, \mathbb{K})$ as k varies over the non-negative integers.
- **Spaces of continuously differentiable functions:** Let $k \in \mathbb{N}^*$, we define the space consisting of functions $f \in C^k_I(\Omega, \mathbb{K})$ whose k-th derivative is continuous as $C^k(\Omega, \mathbb{K})$. The class of infinitely continuously differentiable functions $C^\infty(\Omega, \mathbb{K})$ is defined the intersection of the sets $C^k(\Omega, \mathbb{K})$ as k varies over the non-negative integers.
- **Space of analytic functions:** The space of analytic function $C^\omega(\Omega, \mathbb{K})$ is the space of inifinitely differentiable functions that are locally given by a convergent power series:

$$f \in C^\omega(\Omega, \mathbb{K}) \;\Leftrightarrow\; f \in C^\infty(\Omega, \mathbb{K}) \text{ and } \forall x_0 \in \Omega \;\lim_{x \to x_0} \sum_{n=0}^{+\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n = f(x_0)$$

- **Relation between spaces C^k, C^k_I, D^k and C^ω:**

$$C^0_I \supset C^0 \supset D^1 \supset C^1_I \supset C^1 \supset \cdots \supset D^k \supset C^k_I \supset C^k \supset \cdots \supset D^\infty = C^\infty_I = C^\infty \supset C^\omega$$

- **Lebesgue spaces:** Let $1 \leq p < \infty$ and $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$, we define the space $L^p(\Omega)$ of functions for which the p-th power of the absolute value is Lebesgue integrable:

$$f : \Omega \to \mathbb{K} \in L^p(\Omega) \quad \Leftrightarrow \quad \|f\|_p = \left(\int_\Omega |f(\boldsymbol{x})|^p d\boldsymbol{x} \right)^{1/p} < +\infty$$

A.2 Functions and symbols

- **Set of integers:** Let $(a,b) \in \mathbb{Z}^2$ such that $a < b$, we define the following integer sets:

$$\begin{aligned} [\![a,b]\!] &= [a,b] \bigcap \mathbb{Z} & [\![a,b[\![&= [a,b[\bigcap \mathbb{Z} \\ ]\!]a,b]\!] &=]a,b] \bigcap \mathbb{Z} &]\!]a,b[\![&=]a,b[\bigcap \mathbb{Z} \end{aligned}$$

- **Iverson bracket**: The Iverson converts any logical proposition into a number that is one if the proposition is satisfied, and zero otherwise: $\big[P\big] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$

- **Kronecker delta:** Let $(i,j) \in \mathbb{R}^2$, we define $\delta_{i,j} = \big[i = j\big] = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i! = j \end{cases}$

- **Indicator function:** The indicator function of a set $\mathbb{K}$ is defined as:

$$\mathbb{1}_{\mathbb{K}}(x) = \big[x \in \mathbb{K}\big] = \begin{cases} 1 & \text{if } x \in \mathbb{K} \\ 0 & \text{if } x \notin \mathbb{K} \end{cases}$$

A.3 Vector operations

Let $n \in \mathbb{N}$, $\boldsymbol{a} \in \mathbb{R}^n$ and $\boldsymbol{b} \in \mathbb{R}^n$ we use the following notations:

- Dot product: $\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{k=1}^{n} a_k b_k \in \mathbb{R}$

- Cross product: $\boldsymbol{a} \times \boldsymbol{b} = \left(\sum_{i=1}^{3} \sum_{j=1}^{3} \mathcal{E}_{kij} a_i b_j \right)_{k \in [\![1,3]\!]} \in \mathbb{R}^3$ (where $\mathcal{E}$ is the Levi-Civita symbol)

- Elementwise sum: $\boldsymbol{a} + \boldsymbol{b} = (a_k + b_k)_{k \in [\![1,n]\!]} \in \mathbb{R}^n$

- Elementwise subtraction: $\boldsymbol{a} - \boldsymbol{b} = (a_k - b_k)_{k \in [\![1,n]\!]} \in \mathbb{R}^n$

- Elementwise multiplication: $\boldsymbol{a} \odot \boldsymbol{b} = (a_k b_k)_{k \in [\![1,n]\!]} \in \mathbb{R}^n$

- Elementwise division: $\boldsymbol{a} \oslash \boldsymbol{b} = (a_k / b_k)_{k \in [\![1,n]\!]} \in \mathbb{R}^n$

- Elementwise flooring: $\lfloor \boldsymbol{a} \rfloor = (\lfloor a_k \rfloor)_{k \in [\![1,n]\!]} \in \mathbb{R}^n$

- Elementwise ceiling: $\lceil \boldsymbol{a} \rceil = (\lceil a_k \rceil)_{k \in [\![1,n]\!]} \in \mathbb{R}^n$

- Tensor product: $\boldsymbol{a} \otimes \boldsymbol{b} = (a_i b_j)_{(i,j) \in [\![1,n]\!]^2} \in M_n(\mathbb{R})$

A.4 Vector calculus identities

The following vector calculus identities are required to derive the Navier-Stokes equations. In those identities φ represent a scalar field, $\boldsymbol{u}$ and $\boldsymbol{v}$ are vector fields and A is a tensor field.

Divergence formulas

$$\nabla \cdot (\varphi \boldsymbol{v}) = \varphi(\nabla \cdot \boldsymbol{v}) + \nabla\varphi \cdot \boldsymbol{v} \tag{A.1a}$$

$$\nabla \cdot \left(\varphi \overline{\overline{A}}\right) = \varphi \left(\nabla \cdot \overline{\overline{A}}\right) + \overline{\overline{A}} \nabla\varphi \tag{A.1b}$$

$$\nabla \cdot \left(\overline{\overline{A}} \boldsymbol{v}\right) = \left(\nabla \cdot \overline{\overline{A}}\right) \cdot \boldsymbol{v} + \overline{\overline{A}} : \overline{\overline{\nabla \boldsymbol{v}}} \tag{A.1c}$$

$$\nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{v}) = \overline{\overline{\nabla \boldsymbol{u}}} \boldsymbol{v} + \boldsymbol{u}(\nabla \cdot \boldsymbol{v}) \tag{A.1d}$$

Divergence theorem (Green-Ostrogradski)

The divergence theorem states that the outward flux of a tensor field through a closed surface is equal to the volume integral of the divergence over the region inside the surface. It relates the flow of a vector field through a surface to the behavior of the tensor field inside the surface:

$$\int_{\Omega} \nabla\varphi \ \mathrm{dv} = \oint_{\partial\Omega} \varphi \ \boldsymbol{n} \ \mathrm{ds} \tag{A.2a}$$

$$\int_{\Omega} \nabla \cdot \boldsymbol{v} \ \mathrm{dv} = \oint_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{n} \ \mathrm{ds} \tag{A.2b}$$

$$\int_{\Omega} \nabla \cdot \overline{\overline{T}} \ \mathrm{dv} = \oint_{\partial\Omega} \overline{\overline{T}} \boldsymbol{n} \ \mathrm{ds} \tag{A.2c}$$

Appendix B

HySoP dependencies and tools

This appendix provides some information and references about the dependencies the `HySoP` library uses. Il also covers basic tools that are usefull to build the library and analyze code metrics. It is better used with the electronic version of this work through hyperlinks.

B.1 FFT libraries

- **FFTPACK:** FFTPACK is a package of `Fortran` and `C` subroutines for the fast Fourier transform (FFT). It includes complex, real, sine, cosine, and quarter-wave single and double precision one-dimensional transforms [Swarztrauber 1982].
  Link: https://www.netlib.org/fftpack
- **FFTW:** The Fastest Fourier Transform in the West is a `C` subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data [Frigo et al. 1998]. It can also computes discrete cosine and sine transforms (DCT/DST). The library supports single and double precision floating point numbers as well as extended precision `long double` and non standard `__float128` quadruple-precision floating point types. It provides two different multithreaded implementations using `OpenMP` and `POSIX` threads (`pthread`) in addition to distributed-memory transforms through `MPI`. Benchmarks comparing the performance of `FFTW` to other FFT implementations can be found in [Frigo et al. 2012].
  Link: http://www.fftw.org
- **MKL-FFT:** Intel specific FFT implementation for multidimensional complex-to-complex, real-to-complex, and real-to-real transforms of arbitrary length. The library supports single and double precision floating point numbers and proposes `FFTW` interfaces for compatibility. It provides a multithreaded implementation through user suplied `OpenMP` or `pthread` threads in addition to distributed-memory transforms through `MPI`.
  Link: https://software.intel.com/en-us/mkl/features/fft
- **clFFT:** clFFT is a `C++` library containing FFT functions written in `OpenCL`. In addition to `GPU` devices, the library also supports running on `CPU` devices to facilitate debugging and heterogeneous programming. It supports one-dimensional, two-dimensional and three-dimensional single and double precision batched transforms whose dimension lengths can be a combination of powers of 2, 3, 5, 7, 11 and 13.
  Link: https://github.com/clMathLibraries/clFFT

B.2 Python modules

- **SciPy:** SciPy is a Python based ecosystem of open-source software for mathematics, science, and engineering [Oliphant 2007b]. The SciPy library is one of the core packages that make up the SciPy stack. It provides many user-friendly and efficient numerical routines such as routines for numerical integration, interpolation, optimization and linear algebra. It also provide a pythonic wrapper to the Fortran FFT routines of FFTPACK.
  Link: https://www.scipy.org

- **NumPy:** NumPy is the fundamental package for scientific computing with Python. It contains among other things a powerful n-dimensional array object supporting sophisticated broadcasting functions [Van Der Walt et al. 2011]. NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined to mimic C structures. It contains tools to integrate C++ and Fortran code, see F2PY. It also provide a pythonic wrapper to the C FFT routines of FFTPACK.
  Link: https://www.numpy.org

- **Sympy:** Sympy is a Python library for symbolic mathematics [Meurer et al. 2017]. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. Beyond use as an interactive tool, Sympy can be embedded in other applications and extended with custom functions.
  Link: https://www.sympy.org

- **Python-FLINT:** Python-FLINT is a module wrapping FLINT (Fast Library for Number Theory) and Arb (Arbitrary-precision ball arithmetic). It features integer, rationals, real and complex numbers with arbitrary precision as well as polynomials and matrices over all those types [Hart 2010][Johansson 2013]. This module is used mainly for its linear solver when solving for a matrix containing rationals. It is much more performant than the equivalent solver proposed by Sympy with sympy.Rational elements.
  Link: http://fredrikj.net/python-flint

- **PyOpenCL:** PyOpenCL lets you access GPUs and other massively parallel compute devices from Python trough the OpenCL API [Klöckner et al. 2012]. It tries to offer computing goodness in the spirit of its sister project PyCUDA. PyOpenCL offers broad support and works with with Intel, Nvidia, AMD and POCL OpenCL implementations.
  Link: https://documen.tician.de/pyopencl

- **MPI4py:** MPI4py provides bindings of the Message Passing Interface standard for the Python programming language, allowing any Python program to exploit multiple processors [Dalcín et al. 2005]. This package is constructed on top of the MPI-1/2/3 specifications and provides an object oriented interface. It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communications of any picklable object, as well as optimized communications of Python objects exposing the single-segment buffer interface such as NumPy arrays.
  Link: https://mpi4py.readthedocs.io

- **pyFFTW:** pyFFTW is a pythonic wrapper around FFTW. The ultimate aim is to present a unified interface for all the possible transforms that FFTW can perform. Both the complex DFT and the real DFT are supported. Real to real transforms are not supported in the master branch yet (but are available as a merge request). This wrapper does not support the MPI capabilities of FFTW.
  Link: https://github.com/pyFFTW/pyFFTW

- **gpyfft:** gpyfft is a Python wrapper for the OpenCL FFT library clFFT. It is designed to tightly integrate with PyOpenCL and consists of a low-level Cython based wrapper with an interface similar to the underlying C library. On top of that it offers a high-level interface designed to work on data contained in instances of pyopencl.array.Array. The high-level interface takes some inspiration from pyFFTW.
  Link: https://github.com/geggo/gpyfft

- **mkl_fft:** The mkl_fft Python module wraps the MKL-FFT library and provides numpy.fft and scipy.fftpack compatible FFT interfaces to NumPy and SciPy.
  Link: https://github.com/IntelPython/mkl_fft

- **graph-tool:** graph-tool is is an efficient Python module for manipulation and statistical analysis of graphs. The core data structures and algorithms are implemented in C++, making extensive use of template metaprogramming, based heavily on the Boost Graph Library. An extensive array of features is included, such as support for arbitrary vertex, edge or graph properties and topological aglorithms.
  Link: https://graph-tool.skewed.de

B.3 Build tools

- **Numba:** Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library [Lam et al. 2015]. Numba compiled numerical algorithms in Python can approach the speeds of C or Fortran. Numba is designed to be used with NumPy arrays and functions, the source code remains pure Python while Numba handles the compilation at runtime just in time (JIT). Numba generates specialized code for different array data types and layouts to optimize performance. It supports vector instructions and adapts to the CPU capabilities (SSE, AVX, AVX-512) and enable GPU acceleration through CUDA and ROCm.
  Link: https://numba.pydata.org

- **F2PY:** The purpose of F2PY, the Fortran to Python interface generator, is to provide a connection between Python and Fortran languages [Peterson 2009]. F2PY is part of NumPy and is also available as a standalone command line tool. It makes it possible to call Fortran 77/90/95 external subroutines and Fortran 90/95 module subroutines as well as C functions and allows access to Fortran 77 COMMON blocks and Fortran 90/95 module data, including allocatable arrays.
  Link: https://docs.scipy.org/doc/numpy/f2py

- **SWIG:** The Simplified Wrapper and Interface Generator is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python [Beazley 1996]. SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software.
  Link: http://www.swig.org

- **LLVM:** The LLVM Project is a collection of modular and reusable compiler and toolchain technologies [Lattner et al. 2004]. The LLVM Core libraries provide a modern source and target-independent optimizer, along with code generation support for many popular CPUs. These libraries are built around a well specified code representation known as the LLVM intermediate representation (LLVM IR).
  Link: https://llvm.org

B.4 Benchmarking and debugging tools

- **clpeak:** clpeak is a synthetic benchmarking tool to measure peak capabilities of OpenCL devices [Bhat 2017]. It measures peak metrics that can be achieved using vector operations such as global memory bandwidth, host-to-device and device-to-host memory bandwidth, peak compute for single and double precision floating point numbers and kernel launch latencies.
  Link: https://github.com/krrishnarraj/clpeak

- **mixbench:** The purpose of mixbench is to evaluate performance bounds of GPUs on mixed operational intensity kernels [Konstantinidis et al. 2015]. The executed kernels are customized on a range of different operational intensity values. Modern GPUs are able to hide memory latency by switching execution to threads able to perform compute operations. Using this tool one can assess the practical optimum balance in both types of operations for GPUs with CUDA, OpenCL or HIP.
  Link: https://github.com/ekondis/mixbench

- **gearshifft:** This is a simple and easy extensible benchmark system to answer the question, which FFT library performs best under which conditions [Steinbach et al. 2017]. Conditions are given by compute architecture, inplace or outplace as well as real or complex transforms, data precision, and so on. This project supports FFTW, MKL-FFT, clFFT, rocFFT and cuFFT libraries.
  Link: https://github.com/mpicbg-scicomp/gearshifft

- **Oclgrind:** Oclgrind implements a virtual OpenCL device simulator [Price et al. 2015]. and aims to provide a platform for creating tools to aid OpenCL development. This project implements utilities for debugging memory access errors, detecting data-races and barrier divergence and for interactive OpenCL kernel debugging. The simulator is built on an interpreter for LLVM IR. Oclgrind provides a simple plugin interface that allows third-party developers to extend its functionality. This interface allows a plugin to be notified when various events occur within the simulator, such as an instruction being executed, memory being accessed, or work-group synchronisation constructs occurring.
  Link: https://github.com/jrprice/Oclgrind

Appendix C

Oclgrind plugin and kernel statistics

This plugin aims to output execution statistics about OpenCL kernels run in isolation in Oclgrind. It is heavily inspired from the InstructionCounter plugin already provided by the device simulator and is thus subject to the same 3-clause BSD license (https://github.com/jrprice/Oclgrind/blob/master/LICENSE). This plugin has been tested with LLVM 6.0.1 and Oclgrind 18.3.

C.1 Requirements, build and usage

The plugin consists into a single source file with associated header (InstructionDumper.h and InstructionDumper.cpp) which are compiled to a shared library (libOclgrindInstructionDumper.so). To build the plugin, it is required to copy oclgrind core directory (core/*.h and core/*.cpp) in the current directory. It also requires LLVM to be present at LLVM_DIR, a c++17 compatible compiler defined as CXX and boost progress and property_tree headers to be present in include path. Please refer to the online documentation to obtain more information about the plugin build process (https://github.com/jrprice/Oclgrind/wiki/Creating-Plugins).

The simplest way to get the source code and a working plugin is to clone the repository:

```
#!/usr/bin/env bash
export CXX=...
export LLVM_DIR=...
git clone https://gitlab.com/keckj/oclgrind-instruction-dumper
make
```

Oclgrind can be built from the cloned submodule or directly installed from the system package manager (sudo apt-get install oclgrind). A basic kernel simulation file is present in the repository in the test subdirectory. After build the test can be run with make test (see https://gitlab.com/keckj/oclgrind-instruction-dumper). The general command to run the plugin on an arbitrary kernel is the following:

```
oclgrind-kernel --plugins $(pwd)/libOclgrindInstructionDumper.so kernel.sim
```

Although the plugin supports multithreaded execution, kernel isolation remains a slow process for large kernels (its runs exclusively on CPU). When execution is too long, instruction statistics can be extrapolated from the first work-group by defining the OCLGRIND_ONLY_FIRST_WORKGROUP environment variable (its value is not checked).

C.2 Output sample and obtained kernel statistics

Here is a sample of statistics that are output by the plugin for the three dimensional directional remeshing kernel with $\Lambda_{8,4}$ on a 512^3 single-precision array. The values are printed to the standard output stream correspond only to the first work-group (out of 512^2 work-groups). The output has been formatted to fit the page:


```
{
    "op_counts": {
        "Add": "9604", "Sub": "3072", "Mul": "3072",
        "FSub": "512", "FMul": "8192",
        "Alloca": "18944", "GetElementPtr": "10988",
        "Load": "101400", "Store": "39288",
        "Trunc": "10928", "BitCast": "3072",
        "ZExt": "4608", "SExt": "3692",
        "ICmp": "3900", "FCmp": "5120",
        "PHI": "512", "Call": "48256",
        "ExtractElement": "11068", "InsertElement": "384",
        "Ret": "5760", "Br": "28820"
    },
    "function_calls": {
        "lambda_8_4__0_86b2": "512", "lambda_8_4__1_86b2": "512",
        "lambda_8_4__2_86b2": "512", "lambda_8_4__3_86b2": "512",
        "lambda_8_4__4_86b2": "512", "lambda_8_4__5_86b2": "512",
        "lambda_8_4__6_86b2": "512", "lambda_8_4__7_86b2": "512",
        "lambda_8_4__8_86b2": "512", "lambda_8_4__9_86b2": "512",
        "barrier(unsigned int)": "2560",  "convert_int_rtn(float)": "512"
    },
    "memory_op_bytes": {
        "load":  { "private":  "483344", "global": "4096",
                   "constant": "0",       "local": "2240" },
        "store": { "private":  "164080", "global": "2096",
                   "constant": "0",       "local": "2240" },
        "total": { "private": "647424",  "global": "6192",
                   "constant": "0",       "local": "4480" }
    },
    "integer_ops": { "32": "15748" },
    "floating_point_ops": { "32": "86016" },
    "work_group": { "executed": "1", "total": "262144" }
}
```


If we consider that all work group will perform an equivalent number of instructions and memory transactions, this kernel has an arithmetic intensity of `86016/6192=13.89 FLOP/B`. By knowing the runtime of this kernel on a considered target device, one can deduce achieved bandwidth (`GB/s`) and compute (`GFLOP/s`).

Configuration		Runtime (us)			Mean Memory Bandwidth			Mean Achieved Performance			Kernel Configuration			
integrator	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP32)	FLOP/B	global_size	local_size	P	S
Euler	(2048, 2048)	55.81	54.27	64.51	52.4TiB/s	841.0GiB/s	561.0GiB/s	66.9%	1202.50 (8.1%)	2.0	(1024, 2048)	(1024, 1)	1	✓
Euler	(4096, 4096)	206.98	204.80	210.94	44.3TiB/s	908.3GiB/s	606.3GiB/s	72.3%	1175.35 (7.9%)	1.8	(512, 4096)	(512, 1)	1	✓
Euler	(8192, 8192)	775.49	773.12	780.29	45.1TiB/s	969.7GiB/s	647.3GiB/s	77.2%	1233.16 (8.3%)	1.8	(512, 8192)	(512, 1)	1	✓
Euler	(16384, 16384)	3100.51	3095.55	3110.91	44.0TiB/s	970.1GiB/s	647.6GiB/s	77.3%	1222.91 (8.2%)	1.8	(512, 16384)	(512, 1)	1	✓
RK2	(2048, 2048)	58.37	56.32	68.61	55.4TiB/s	1.3TiB/s	537.5GiB/s	64.1%	1724.63 (11.6%)	3.0	(512, 2048)	(512, 1)	1	✓
RK2	(4096, 4096)	213.34	211.97	218.11	54.7TiB/s	1.4TiB/s	590.5GiB/s	70.4%	1828.36 (12.3%)	2.9	(256, 4096)	(256, 1)	1	✓
RK2	(8192, 8192)	802.05	795.65	807.94	58.2TiB/s	1.5TiB/s	625.8GiB/s	74.7%	1945.37 (13.1%)	2.9	(512, 8192)	(512, 1)	1	✓
RK2	(16384, 16384)	3159.23	3154.94	3166.21	58.0TiB/s	1.5TiB/s	635.5GiB/s	75.8%	1964.90 (13.2%)	2.9	(512, 16384)	(512, 1)	1	✓
RK3	(2048, 2048)	62.53	61.44	64.51	71.0TiB/s	1.7TiB/s	500.8GiB/s	59.7%	2683.20 (18.0%)	5.0	(1024, 2048)	(1024, 1)	1	✓
RK3	(4096, 4096)	223.20	221.18	225.28	66.2TiB/s	1.9TiB/s	564.4GiB/s	67.3%	2875.13 (19.3%)	4.7	(256, 4096)	(256, 1)	1	✓
RK3	(8192, 8192)	828.48	818.18	838.66	71.4TiB/s	2.1TiB/s	605.9GiB/s	72.3%	3098.34 (20.8%)	4.8	(512, 8192)	(512, 1)	1	✓
RK3	(16384, 16384)	3258.66	3254.27	3269.63	71.5TiB/s	2.1TiB/s	616.1GiB/s	73.5%	3140.59 (21.1%)	4.7	(512, 16384)	(512, 1)	1	✓
RK4	(2048, 2048)	66.75	64.51	77.82	77.4TiB/s	2.1TiB/s	469.1GiB/s	56.0%	3078.87 (20.7%)	6.1	(1024, 2048)	(1024, 1)	1	✓
RK4	(4096, 4096)	233.66	231.42	236.54	75.8TiB/s	2.4TiB/s	539.1GiB/s	64.3%	3392.58 (22.8%)	5.9	(256, 4096)	(256, 1)	1	✓
RK4	(8192, 8192)	857.70	845.82	868.35	81.6TiB/s	2.6TiB/s	587.5GiB/s	70.1%	3687.21 (24.7%)	5.8	(256, 8192)	(256, 1)	1	✓
RK4	(16384, 16384)	3363.20	3355.65	3369.98	82.8TiB/s	2.6TiB/s	599.3GiB/s	71.5%	3756.32 (25.2%)	5.8	(256, 16384)	(256, 1)	1	✓
RK4_38	(2048, 2048)	66.53	65.54	69.63	78.4TiB/s	2.1TiB/s	470.6GiB/s	56.2%	3467.51 (23.3%)	6.9	(1024, 2048)	(1024, 1)	1	✓
RK4_38	(4096, 4096)	238.18	234.50	241.66	75.1TiB/s	2.3TiB/s	528.9GiB/s	63.1%	3750.95 (25.2%)	6.6	(256, 4096)	(256, 1)	1	✓
RK4_38	(8192, 8192)	856.80	852.99	864.26	82.6TiB/s	2.6TiB/s	588.1GiB/s	70.2%	4161.02 (27.9%)	6.6	(256, 8192)	(256, 1)	1	✓
RK4_38	(16384, 16384)	3390.78	3383.30	3400.70	82.9TiB/s	2.6TiB/s	594.4GiB/s	70.9%	4200.76 (28.2%)	6.6	(256, 16384)	(256, 1)	1	✓
integrator	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP32)	FLOP/B	global_size	local_size	P	S
Euler	(64, 64, 64)	9.25	8.19	12.29	26.5TiB/s	323.4GiB/s	217.8GiB/s	26.0%	538.57 (3.6%)	2.3	(64, 16, 64)	(64, 1, 1)	1	✓
Euler	(128, 128, 128)	30.91	29.70	33.79	73.6TiB/s	766.1GiB/s	513.4GiB/s	61.2%	1356.85 (9.1%)	2.5	(128, 64, 128)	(128, 1, 1)	1	✓
Euler	(256, 256, 256)	195.58	193.54	197.63	87.0TiB/s	968.7GiB/s	649.1GiB/s	77.4%	1544.04 (10.4%)	2.2	(128, 256, 256)	(128, 1, 1)	1	✓
Euler	(512, 512, 512)	1512.77	1507.33	1520.64	69.9TiB/s	996.7GiB/s	666.2GiB/s	79.5%	1463.93 (9.8%)	2.0	(256, 128, 512)	(256, 1, 1)	1	✓
RK2	(64, 64, 64)	10.40	9.22	16.38	27.9TiB/s	475.4GiB/s	193.7GiB/s	23.1%	705.77 (4.7%)	3.4	(64, 16, 64)	(64, 1, 1)	1	✓
RK2	(128, 128, 128)	32.13	30.72	37.89	82.2TiB/s	1.2TiB/s	493.9GiB/s	58.9%	1892.97 (12.7%)	3.6	(128, 64, 128)	(128, 1, 1)	1	✓
RK2	(256, 256, 256)	202.02	198.66	206.85	98.7TiB/s	1.5TiB/s	628.4GiB/s	75.0%	2242.32 (15.0%)	3.3	(128, 256, 256)	(128, 1, 1)	1	✓
RK2	(512, 512, 512)	1535.62	1531.90	1543.17	96.9TiB/s	1.6TiB/s	653.7GiB/s	78.0%	2447.29 (16.4%)	3.5	(512, 128, 512)	(512, 1, 1)	1	✓
RK3	(64, 64, 64)	10.59	10.24	15.36	36.7TiB/s	668.4GiB/s	190.2GiB/s	22.7%	1214.26 (8.1%)	5.9	(64, 32, 64)	(64, 1, 1)	2	✓
RK3	(128, 128, 128)	33.38	32.77	36.86	81.3TiB/s	1.6TiB/s	475.5GiB/s	56.7%	2701.87 (18.1%)	5.3	(128, 32, 128)	(128, 1, 1)	1	✓
RK3	(256, 256, 256)	215.20	212.99	219.14	107.1TiB/s	2.0TiB/s	589.9GiB/s	70.4%	3274.36 (22.0%)	5.2	(128, 256, 256)	(128, 1, 1)	1	✓
RK3	(512, 512, 512)	1577.66	1571.84	1583.10	97.6TiB/s	2.2TiB/s	638.8GiB/s	76.2%	3445.49 (23.1%)	5.0	(256, 128, 512)	(256, 1, 1)	1	✓
RK4	(64, 64, 64)	10.08	9.22	14.34	38.2TiB/s	878.0GiB/s	199.8GiB/s	23.8%	1352.33 (9.1%)	6.3	(64, 16, 64)	(64, 1, 1)	1	✓
RK4	(128, 128, 128)	35.04	33.79	39.94	87.9TiB/s	2.0TiB/s	452.9GiB/s	54.0%	3112.21 (20.9%)	6.4	(128, 32, 128)	(128, 1, 1)	1	✓
RK4	(256, 256, 256)	222.40	219.14	230.40	105.5TiB/s	2.5TiB/s	570.8GiB/s	68.1%	3771.86 (25.3%)	6.2	(128, 128, 256)	(128, 1, 1)	1	✓
RK4	(512, 512, 512)	1613.60	1606.66	1620.99	110.0TiB/s	2.7TiB/s	624.6GiB/s	74.5%	4117.36 (27.6%)	6.1	(256, 128, 512)	(256, 1, 1)	1	✓
RK4_38	(64, 64, 64)	11.42	10.24	15.36	32.3TiB/s	780.0GiB/s	181.7GiB/s	21.7%	1285.02 (8.6%)	6.6	(32, 32, 64)	(32, 1, 1)	1	✓
RK4_38	(128, 128, 128)	34.91	33.79	46.08	88.9TiB/s	2.0TiB/s	454.5GiB/s	54.2%	3484.04 (23.4%)	7.1	(128, 32, 128)	(128, 1, 1)	1	✓
RK4_38	(256, 256, 256)	214.66	211.97	218.11	110.1TiB/s	2.6TiB/s	591.4GiB/s	70.6%	4376.88 (29.4%)	6.9	(128, 128, 256)	(128, 1, 1)	1	✓
RK4_38	(512, 512, 512)	1620.90	1615.87	1627.14	110.4TiB/s	2.7TiB/s	621.8GiB/s	74.2%	4595.66 (30.8%)	6.9	(256, 128, 512)	(256, 1, 1)	1	✓
integrator	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP32)	FLOP/B	global_size	local_size	P	S

Table C.1 – 2D and 3D single-precision directional advection kernel statistics. Benchmark of advection kernels on the `NVidia Tesla V100-SXM2 GPU`. Kernel parameter `P` represent the number of simultaneously advected particles (vectorization) and `S` determines whether the shared memory version of the kernel was chosen by the autotuner or not.

Configuration		Runtime (us)			Mean Memory Bandwidth			Mean Achieved Performance			Kernel Configuration			
integrator	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	P	S
Euler	(2048, 2048)	97.09	95.23	104.45	37.7TiB/s	966.9GiB/s	645.0GiB/s	77.0%	691.22 (9.3%)	1.0	(1024, 2048)	(1024, 1)	1	✓
Euler	(4096, 4096)	355.36	353.28	359.42	35.7TiB/s	1.0TiB/s	704.9GiB/s	84.1%	708.18 (9.5%)	0.9	(1024, 4096)	(1024, 1)	1	✓
Euler	(8192, 8192)	1402.59	1400.83	1409.02	33.4TiB/s	1.0TiB/s	714.4GiB/s	85.2%	693.77 (9.3%)	0.9	(1024, 8192)	(1024, 1)	1	✓
Euler	(16384, 16384)	5684.51	5675.01	5701.63	31.1TiB/s	1.0TiB/s	705.0GiB/s	84.1%	672.92 (9.0%)	0.9	(1024, 8192)	(1024, 1)	1	✓
RK2	(2048, 2048)	98.72	97.28	101.38	43.4TiB/s	1.5TiB/s	635.6GiB/s	75.8%	1019.68 (13.7%)	1.5	(512, 2048)	(512, 1)	1	✓
RK2	(4096, 4096)	367.10	365.57	372.74	46.7TiB/s	1.7TiB/s	682.3GiB/s	81.4%	1096.84 (14.7%)	1.5	(1024, 4096)	(1024, 1)	1	✓
RK2	(8192, 8192)	1423.01	1419.26	1426.43	45.5TiB/s	1.7TiB/s	704.1GiB/s	84.0%	1108.26 (14.9%)	1.5	(1024, 8192)	(1024, 1)	1	✓
RK2	(16384, 16384)	5756.74	5743.62	5769.22	43.1TiB/s	1.7TiB/s	696.2GiB/s	83.1%	1084.14 (14.6%)	1.5	(1024, 8192)	(1024, 1)	1	✓
RK3	(2048, 2048)	103.52	102.40	106.50	53.1TiB/s	2.1TiB/s	606.1GiB/s	72.3%	1580.16 (21.2%)	2.4	(512, 2048)	(512, 1)	1	✓
RK3	(4096, 4096)	377.31	373.76	380.93	55.7TiB/s	2.3TiB/s	665.2GiB/s	79.4%	1711.91 (23.0%)	2.4	(512, 4096)	(512, 1)	1	✓
RK3	(8192, 8192)	1459.81	1457.15	1464.32	56.2TiB/s	2.3TiB/s	687.7GiB/s	82.0%	1758.39 (23.6%)	2.4	(512, 8192)	(512, 1)	1	✓
RK3	(16384, 16384)	5914.34	5904.38	5925.89	55.0TiB/s	2.3TiB/s	677.6GiB/s	80.8%	1736.06 (23.3%)	2.4	(1024, 8192)	(1024, 1)	1	✓
RK4	(2048, 2048)	104.80	102.40	107.52	63.0TiB/s	2.6TiB/s	598.7GiB/s	71.4%	1921.06 (25.8%)	3.0	(512, 2048)	(512, 1)	1	✓
RK4	(4096, 4096)	389.25	385.02	392.19	65.4TiB/s	2.8TiB/s	644.8GiB/s	76.9%	2047.33 (27.5%)	3.0	(512, 4096)	(512, 1)	1	✓
RK4	(8192, 8192)	1481.73	1477.63	1491.97	67.4TiB/s	3.0TiB/s	677.5GiB/s	80.8%	2140.00 (28.7%)	2.9	(512, 8192)	(512, 1)	1	✓
RK4	(16384, 16384)	6030.94	6017.02	6047.74	65.3TiB/s	2.9TiB/s	665.8GiB/s	79.4%	2097.52 (28.2%)	2.9	(512, 4096)	(512, 1)	1	✓
RK4_38	(2048, 2048)	105.38	102.40	107.52	63.6TiB/s	2.6TiB/s	595.4GiB/s	71.0%	2149.37 (28.9%)	3.4	(512, 2048)	(512, 1)	1	✓
RK4_38	(4096, 4096)	381.06	378.88	388.10	67.8TiB/s	2.9TiB/s	658.6GiB/s	78.6%	2355.51 (31.6%)	3.3	(512, 4096)	(512, 1)	1	✓
RK4_38	(8192, 8192)	1486.62	1480.70	1497.09	68.2TiB/s	3.0TiB/s	675.3GiB/s	80.6%	2403.80 (32.3%)	3.3	(512, 8192)	(512, 1)	1	✓
RK4_38	(16384, 16384)	6049.15	6037.50	6064.13	66.0TiB/s	2.9TiB/s	663.8GiB/s	79.2%	2357.46 (31.6%)	3.3	(512, 4096)	(512, 1)	1	✓
integrator	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	P	S
Euler	(64, 64, 64)	9.79	9.22	12.29	30.1TiB/s	610.9GiB/s	411.4GiB/s	49.1%	508.65 (6.8%)	1.2	(64, 16, 64)	(64, 1, 1)	1	✓
Euler	(128, 128, 128)	51.30	50.18	55.30	50.1TiB/s	932.9GiB/s	628.3GiB/s	75.0%	735.91 (9.9%)	1.1	(64, 128, 128)	(64, 1, 1)	1	✓
Euler	(256, 256, 256)	354.56	353.28	360.45	58.0TiB/s	1.0TiB/s	716.1GiB/s	85.4%	851.73 (11.4%)	1.1	(128, 256, 256)	(128, 1, 1)	1	✓
Euler	(512, 512, 512)	2756.00	2753.54	2764.80	63.3TiB/s	1.1TiB/s	728.5GiB/s	86.9%	974.00 (13.1%)	1.2	(512, 256, 512)	(512, 1, 1)	1	✓
RK2	(64, 64, 64)	10.11	9.22	13.31	40.6TiB/s	977.8GiB/s	398.4GiB/s	47.5%	751.80 (10.1%)	1.8	(64, 32, 64)	(64, 1, 1)	1	✓
RK2	(128, 128, 128)	58.69	56.32	70.66	47.0TiB/s	1.3TiB/s	549.1GiB/s	65.5%	929.08 (12.5%)	1.6	(64, 64, 128)	(64, 1, 1)	1	✓
RK2	(256, 256, 256)	365.41	362.50	370.69	60.4TiB/s	1.7TiB/s	694.9GiB/s	82.9%	1193.75 (16.0%)	1.6	(128, 128, 256)	(128, 1, 1)	1	✓
RK2	(512, 512, 512)	2781.54	2778.11	2786.30	75.5TiB/s	1.8TiB/s	721.8GiB/s	86.1%	1399.34 (18.8%)	1.8	(512, 256, 512)	(512, 1, 1)	1	✓
RK3	(64, 64, 64)	10.66	9.22	14.34	41.3TiB/s	1.3TiB/s	378.0GiB/s	45.1%	1057.83 (14.2%)	2.6	(64, 16, 64)	(64, 1, 1)	1	✓
RK3	(128, 128, 128)	53.86	52.22	57.34	85.7TiB/s	2.0TiB/s	589.3GiB/s	70.3%	1791.24 (24.0%)	2.8	(128, 128, 128)	(128, 1, 1)	1	✓
RK3	(256, 256, 256)	373.02	370.69	378.88	99.0TiB/s	2.3TiB/s	675.4GiB/s	80.6%	2068.91 (27.8%)	2.9	(256, 256, 256)	(256, 1, 1)	1	✓
RK3	(512, 512, 512)	2842.11	2836.48	2851.84	79.2TiB/s	2.4TiB/s	706.5GiB/s	84.3%	2030.66 (27.3%)	2.7	(512, 128, 512)	(512, 1, 1)	1	✓
RK4	(64, 64, 64)	10.65	10.24	13.31	52.1TiB/s	1.6TiB/s	378.1GiB/s	45.1%	1303.95 (17.5%)	3.2	(64, 32, 64)	(64, 1, 1)	1	✓
RK4	(128, 128, 128)	53.28	52.22	58.37	97.1TiB/s	2.6TiB/s	595.7GiB/s	71.1%	2164.81 (29.1%)	3.4	(128, 128, 128)	(128, 1, 1)	1	✓
RK4	(256, 256, 256)	364.93	363.52	367.62	113.4TiB/s	3.0TiB/s	690.4GiB/s	82.4%	2528.57 (33.9%)	3.4	(256, 256, 256)	(256, 1, 1)	1	✓
RK4	(512, 512, 512)	2876.64	2857.98	2889.73	115.1TiB/s	3.1TiB/s	698.0GiB/s	83.3%	2566.18 (34.4%)	3.4	(512, 512, 512)	(512, 1, 1)	1	✓
RK4_38	(64, 64, 64)	10.75	10.24	14.34	52.2TiB/s	1.6TiB/s	374.7GiB/s	44.7%	1438.48 (19.3%)	3.6	(64, 32, 64)	(64, 1, 1)	1	✓
RK4_38	(128, 128, 128)	52.86	51.20	55.30	98.8TiB/s	2.6TiB/s	600.4GiB/s	71.6%	2419.91 (32.5%)	3.8	(128, 128, 128)	(128, 1, 1)	1	✓
RK4_38	(256, 256, 256)	363.14	360.45	371.71	115.0TiB/s	3.0TiB/s	693.8GiB/s	82.8%	2818.26 (37.8%)	3.8	(256, 256, 256)	(256, 1, 1)	1	✓
RK4_38	(512, 512, 512)	2855.20	2844.67	2864.13	117.0TiB/s	3.1TiB/s	703.2GiB/s	83.9%	2867.50 (38.5%)	3.8	(512, 512, 512)	(512, 1, 1)	1	✓
integrator	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	P	S

Table C.2 – 2D and 3D double-precision directional advection kernel statistics. Benchmark of advection kernels on the `NVidia Tesla V100-SXM2 GPU`. Kernel parameter `P` represent the number of simultaneously advected particles (vectorization) and `S` determines whether the shared memory version of the kernel was chosen by the autotuner or not.

Configuration		Runtime (us)			Mean Memory Bandwidth			Mean Achieved Performance			Kernel Configuration			
remesh	**discretization**	**mean**	**min**	**max**	**private**	**local**	**global**	**BDW (%)**	**GFLOPS (FP32)**	**FLOP/B**	**global_size**	**local_size**	**P**	**A**
$\Lambda_{2,1}$	(2048, 2048)	72.61	70.66	80.90	23.5TiB/s	435.4GiB/s	646.2GiB/s	77.1%	1270.86 (8.5%)	1.83	(256, 2048)	(256, 1)	2	✓
$\Lambda_{2,1}$	(4096, 4096)	268.51	266.24	279.55	25.4TiB/s	468.3GiB/s	698.6GiB/s	83.4%	1374.61 (9.2%)	1.83	(512, 4096)	(512, 1)	2	✓
$\Lambda_{2,1}$	(8192, 8192)	1034.05	1030.14	1043.46	30.9TiB/s	494.9GiB/s	725.5GiB/s	86.6%	1427.78 (9.6%)	1.83	(256, 8192)	(256, 1)	1	✓
$\Lambda_{2,1}$	(16384, 16384)	4134.18	4128.77	4142.08	24.9TiB/s	485.2GiB/s	725.7GiB/s	86.6%	1428.48 (9.6%)	1.83	(1024, 16384)	(1024, 1)	2	✓
$\Lambda_{4,2}$	(2048, 2048)	73.47	71.68	83.97	32.5TiB/s	428.7GiB/s	638.8GiB/s	76.2%	3139.79 (21.1%)	4.58	(512, 2048)	(512, 1)	2	✓
$\Lambda_{4,2}$	(4096, 4096)	267.71	265.22	281.60	32.8TiB/s	470.6GiB/s	700.8GiB/s	83.6%	3446.79 (23.1%)	4.58	(512, 4096)	(512, 1)	2	✓
$\Lambda_{4,2}$	(8192, 8192)	1038.62	1035.26	1052.67	38.0TiB/s	496.5GiB/s	722.3GiB/s	86.2%	3553.73 (23.9%)	4.58	(256, 8192)	(256, 1)	1	✓
$\Lambda_{4,2}$	(16384, 16384)	4185.25	4177.92	4196.35	32.1TiB/s	479.7GiB/s	716.9GiB/s	85.5%	3527.62 (23.7%)	4.58	(1024, 16384)	(1024, 1)	2	✓
$\Lambda_{6,4}$	(2048, 2048)	76.99	75.78	86.02	47.0TiB/s	413.8GiB/s	609.8GiB/s	72.8%	7299.94 (49.0%)	11.15	(512, 2048)	(512, 1)	1	✓
$\Lambda_{6,4}$	(4096, 4096)	276.32	273.41	286.72	44.2TiB/s	456.8GiB/s	679.1GiB/s	81.0%	8136.03 (54.6%)	11.16	(512, 4096)	(512, 1)	2	✓
$\Lambda_{6,4}$	(8192, 8192)	1051.71	1048.58	1059.84	50.2TiB/s	494.0GiB/s	713.4GiB/s	85.1%	8550.43 (57.4%)	11.16	(256, 8192)	(256, 1)	1	✓
$\Lambda_{6,4}$	(16384, 16384)	4247.90	4240.38	4255.74	49.6TiB/s	480.0GiB/s	706.4GiB/s	84.3%	8467.79 (56.8%)	11.16	(512, 16384)	(512, 1)	1	✓
$\Lambda_{8,4}$	(2048, 2048)	78.40	76.80	88.06	52.1TiB/s	407.9GiB/s	599.1GiB/s	71.5%	8987.79 (60.3%)	13.97	(512, 2048)	(512, 1)	1	✓
$\Lambda_{8,4}$	(4096, 4096)	286.94	281.60	301.06	53.1TiB/s	456.0GiB/s	654.1GiB/s	78.0%	9822.73 (65.9%)	13.99	(256, 4096)	(256, 1)	1	✓
$\Lambda_{8,4}$	(8192, 8192)	1064.10	1059.84	1074.18	56.6TiB/s	491.9GiB/s	705.2GiB/s	84.1%	10595.18 (71.1%)	13.99	(256, 8192)	(256, 1)	1	✓
$\Lambda_{8,4}$	(16384, 16384)	4309.09	4298.75	4318.21	55.9TiB/s	475.0GiB/s	696.4GiB/s	83.1%	10465.59 (70.2%)	14.00	(512, 16384)	(512, 1)	1	✓
remesh	**discretization**	**mean**	**min**	**max**	**private**	**local**	**global**	**BDW (%)**	**GFLOPS (FP32)**	**FLOP/B**	**global_size**	**local_size**	**P**	**A**
$\Lambda_{2,1}$	(64, 64, 64)	10.30	9.22	21.50	21.6TiB/s	207.3GiB/s	293.2GiB/s	35.0%	559.70 (3.8%)	1.78	(64, 32, 32)	(64, 1, 1)	1	✓
$\Lambda_{2,1}$	(128, 128, 128)	40.54	38.91	51.20	38.4TiB/s	403.4GiB/s	587.1GiB/s	70.0%	1137.96 (7.6%)	1.81	(64, 128, 128)	(64, 1, 1)	2	✓
$\Lambda_{2,1}$	(256, 256, 256)	256.48	254.98	267.26	47.7TiB/s	533.1GiB/s	736.8GiB/s	87.9%	1439.09 (9.7%)	1.82	(64, 256, 256)	(64, 1, 1)	1	✓
$\Lambda_{2,1}$	(512, 512, 512)	1996.67	1993.73	2007.04	33.7TiB/s	524.3GiB/s	754.2GiB/s	90.0%	1478.86 (9.9%)	1.83	(64, 512, 512)	(64, 1, 1)	2	✗
$\Lambda_{4,2}$	(64, 64, 64)	10.50	9.22	20.48	27.9TiB/s	209.3GiB/s	290.8GiB/s	34.7%	1373.66 (9.2%)	4.40	(64, 64, 32)	(64, 1, 1)	1	✓
$\Lambda_{4,2}$	(128, 128, 128)	40.90	38.91	51.20	44.1TiB/s	405.9GiB/s	585.0GiB/s	69.8%	2820.41 (18.9%)	4.49	(64, 128, 128)	(64, 1, 1)	2	✓
$\Lambda_{4,2}$	(256, 256, 256)	258.21	257.02	268.29	54.7TiB/s	544.6GiB/s	733.7GiB/s	87.5%	3573.66 (24.0%)	4.54	(64, 256, 256)	(64, 1, 1)	1	✓
$\Lambda_{4,2}$	(512, 512, 512)	2013.66	2012.16	2022.40	40.9TiB/s	527.6GiB/s	748.8GiB/s	89.3%	3665.94 (24.6%)	4.56	(64, 512, 512)	(64, 1, 1)	2	✗
$\Lambda_{6,4}$	(64, 64, 64)	10.88	10.24	19.46	28.0TiB/s	207.5GiB/s	283.3GiB/s	33.8%	3228.32 (21.7%)	10.61	(64, 32, 32)	(64, 1, 1)	1	✓
$\Lambda_{6,4}$	(128, 128, 128)	42.02	40.96	51.20	75.8TiB/s	400.9GiB/s	572.3GiB/s	68.3%	6688.37 (44.9%)	10.88	(128, 128, 128)	(128, 1, 1)	1	✓
$\Lambda_{6,4}$	(256, 256, 256)	263.07	262.14	272.38	66.4TiB/s	549.4GiB/s	722.0GiB/s	86.1%	8545.75 (57.4%)	11.02	(64, 256, 256)	(64, 1, 1)	1	✓
$\Lambda_{6,4}$	(512, 512, 512)	2040.10	2036.74	2048.00	68.3TiB/s	528.5GiB/s	740.0GiB/s	88.3%	8815.85 (59.2%)	11.09	(128, 512, 512)	(128, 1, 1)	1	✓
$\Lambda_{8,4}$	(64, 64, 64)	12.19	11.26	25.60	26.9TiB/s	190.2GiB/s	255.3GiB/s	30.5%	3612.22 (24.2%)	13.18	(64, 16, 64)	(64, 1, 1)	1	✓
$\Lambda_{8,4}$	(128, 128, 128)	43.52	41.98	52.22	78.6TiB/s	392.7GiB/s	555.4GiB/s	66.3%	8095.62 (54.3%)	13.58	(128, 128, 128)	(128, 1, 1)	1	✓
$\Lambda_{8,4}$	(256, 256, 256)	267.26	266.24	279.55	72.5TiB/s	555.4GiB/s	712.5GiB/s	85.0%	10546.02 (70.8%)	13.78	(64, 256, 256)	(64, 1, 1)	1	✓
$\Lambda_{8,4}$	(512, 512, 512)	2091.81	2087.94	2104.32	73.8TiB/s	522.9GiB/s	722.7GiB/s	86.2%	10779.47 (72.3%)	13.89	(128, 512, 512)	(128, 1, 1)	1	✓
remesh	**discretization**	**mean**	**min**	**max**	**private**	**local**	**global**	**BDW (%)**	**GFLOPS (FP32)**	**FLOP/B**	**global_size**	**local_size**	**P**	**A**

Table C.3 – 2D and 3D single-precision directional remesh kernel statistics. Benchmark of remeshing kernels $\Lambda_{p,r}$ on the `NVidia Tesla V100-SXM2 GPU`. Kernel parameter `P` represent the number of remeshed particles and `A` that atomic intrinsics are used.

Configuration		Runtime (us)			Mean Memory Bandwidth			Mean Achieved Performance			Kernel Configuration			
remesh	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	P	A
$\Lambda_{2,1}$	(2048, 2048)	145.57	142.34	157.70	15.9TiB/s	434.4GiB/s	644.7GiB/s	76.9%	633.89 (8.5%)	0.9	(256, 256)	(256, 1)	2	✗
$\Lambda_{2,1}$	(4096, 4096)	541.25	535.55	549.89	16.8TiB/s	472.7GiB/s	693.2GiB/s	82.7%	681.94 (9.2%)	0.9	(128, 512)	(128, 1)	2	✗
$\Lambda_{2,1}$	(8192, 8192)	2305.60	2297.86	2316.29	16.5TiB/s	435.0GiB/s	650.7GiB/s	77.6%	640.35 (8.6%)	0.9	(1024, 4096)	(1024, 1)	2	✗
$\Lambda_{2,1}$	(16384, 16384)	9108.29	8866.82	9209.86	16.3TiB/s	440.4GiB/s	658.8GiB/s	78.6%	648.37 (8.7%)	0.9	(1024, 8192)	(1024, 1)	2	✗
$\Lambda_{4,2}$	(2048, 2048)	148.51	145.41	156.67	21.3TiB/s	473.4GiB/s	632.1GiB/s	75.4%	1553.32 (20.8%)	2.3	(32, 2048)	(32, 1)	2	✗
$\Lambda_{4,2}$	(4096, 4096)	550.08	543.74	559.10	22.8TiB/s	468.7GiB/s	682.2GiB/s	81.4%	1677.47 (22.5%)	2.3	(128, 512)	(128, 1)	2	✗
$\Lambda_{4,2}$	(8192, 8192)	2224.03	2206.72	2245.63	22.5TiB/s	477.7GiB/s	674.7GiB/s	80.5%	1659.59 (22.3%)	2.3	(64, 1024)	(64, 1)	2	✗
$\Lambda_{4,2}$	(16384, 16384)	9101.98	9018.37	9196.54	22.0TiB/s	494.4GiB/s	659.3GiB/s	78.7%	1622.06 (21.8%)	2.3	(32, 2048)	(32, 1)	2	✗
$\Lambda_{6,4}$	(2048, 2048)	154.11	152.58	165.89	33.3TiB/s	409.5GiB/s	609.3GiB/s	72.7%	3646.94 (49.0%)	5.6	(512, 2048)	(512, 1)	2	✗
$\Lambda_{6,4}$	(4096, 4096)	563.42	557.06	587.78	33.6TiB/s	461.0GiB/s	666.1GiB/s	79.5%	3990.15 (53.6%)	5.6	(128, 512)	(128, 1)	2	✗
$\Lambda_{6,4}$	(8192, 8192)	2245.73	2229.25	2263.04	33.7TiB/s	480.1GiB/s	668.2GiB/s	79.7%	4004.31 (53.7%)	5.6	(64, 1024)	(64, 1)	2	✗
$\Lambda_{6,4}$	(16384, 16384)	9223.26	9133.06	9280.51	32.8TiB/s	501.4GiB/s	650.7GiB/s	77.6%	3899.96 (52.3%)	5.6	(32, 2048)	(32, 1)	2	✗
$\Lambda_{8,4}$	(2048, 2048)	155.23	152.58	163.84	38.6TiB/s	407.3GiB/s	605.1GiB/s	72.2%	4539.29 (60.9%)	7.0	(512, 2048)	(512, 1)	2	✗
$\Lambda_{8,4}$	(4096, 4096)	580.35	577.54	599.04	39.9TiB/s	435.8GiB/s	646.8GiB/s	77.2%	4856.66 (65.2%)	7.0	(512, 4096)	(512, 1)	2	✗
$\Lambda_{8,4}$	(8192, 8192)	2275.36	2262.02	2300.93	39.3TiB/s	480.7GiB/s	659.6GiB/s	78.7%	4954.95 (66.5%)	7.0	(64, 1024)	(64, 1)	2	✗
$\Lambda_{8,4}$	(16384, 16384)	9303.81	9230.34	9376.77	38.5TiB/s	510.5GiB/s	645.1GiB/s	77.0%	4847.17 (65.1%)	7.0	(32, 2048)	(32, 1)	2	✗
remesh	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	P	A
$\Lambda_{2,1}$	(64, 64, 64)	13.12	12.29	26.62	15.9TiB/s	325.6GiB/s	460.6GiB/s	54.9%	439.57 (5.9%)	0.9	(32, 32, 64)	(32, 1, 1)	2	✗
$\Lambda_{2,1}$	(128, 128, 128)	74.24	72.70	86.02	26.4TiB/s	440.7GiB/s	641.3GiB/s	76.5%	621.46 (8.3%)	0.9	(64, 128, 128)	(64, 1, 1)	2	✗
$\Lambda_{2,1}$	(256, 256, 256)	523.46	521.22	532.48	29.9TiB/s	488.8GiB/s	722.0GiB/s	86.1%	705.12 (9.5%)	0.9	(128, 256, 256)	(128, 1, 1)	2	✗
$\Lambda_{2,1}$	(512, 512, 512)	4150.82	4145.15	4159.49	25.6TiB/s	487.5GiB/s	725.6GiB/s	86.6%	711.38 (9.5%)	0.9	(256, 256, 512)	(256, 1, 1)	2	✗
$\Lambda_{4,2}$	(64, 64, 64)	13.31	12.29	24.58	19.8TiB/s	330.1GiB/s	458.5GiB/s	54.7%	1083.08 (14.5%)	2.2	(32, 32, 64)	(32, 1, 1)	2	✗
$\Lambda_{4,2}$	(128, 128, 128)	74.53	72.70	88.06	27.4TiB/s	471.7GiB/s	642.1GiB/s	76.6%	1547.65 (20.8%)	2.2	(32, 128, 128)	(32, 1, 1)	2	✗
$\Lambda_{4,2}$	(256, 256, 256)	536.90	532.48	547.84	35.6TiB/s	480.2GiB/s	705.7GiB/s	84.2%	1718.67 (23.1%)	2.3	(128, 256, 256)	(128, 1, 1)	2	✗
$\Lambda_{4,2}$	(512, 512, 512)	4232.48	4221.95	4248.58	31.6TiB/s	479.9GiB/s	712.5GiB/s	85.0%	1744.13 (23.4%)	2.3	(256, 256, 512)	(256, 1, 1)	2	✗
$\Lambda_{6,4}$	(64, 64, 64)	14.62	13.31	26.62	24.9TiB/s	308.8GiB/s	421.5GiB/s	50.3%	2402.03 (32.2%)	5.3	(32, 32, 64)	(32, 1, 1)	2	✗
$\Lambda_{6,4}$	(128, 128, 128)	76.74	75.78	88.06	41.6TiB/s	439.1GiB/s	626.8GiB/s	74.8%	3662.15 (49.2%)	5.4	(64, 128, 128)	(64, 1, 1)	2	✗
$\Lambda_{6,4}$	(256, 256, 256)	543.20	539.65	555.01	47.0TiB/s	478.2GiB/s	699.3GiB/s	83.4%	4138.71 (55.6%)	5.5	(128, 256, 256)	(128, 1, 1)	2	✗
$\Lambda_{6,4}$	(512, 512, 512)	4266.27	4254.72	4278.27	43.3TiB/s	477.9GiB/s	707.8GiB/s	84.4%	4215.66 (56.6%)	5.5	(256, 256, 512)	(256, 1, 1)	2	✗
$\Lambda_{8,4}$	(64, 64, 64)	15.62	14.34	30.72	29.2TiB/s	297.0GiB/s	398.7GiB/s	47.6%	2820.20 (37.9%)	6.6	(32, 64, 64)	(32, 1, 1)	2	✗
$\Lambda_{8,4}$	(128, 128, 128)	77.86	76.80	89.09	46.6TiB/s	439.0GiB/s	620.9GiB/s	74.1%	4525.30 (60.7%)	6.8	(64, 128, 128)	(64, 1, 1)	2	✗
$\Lambda_{8,4}$	(256, 256, 256)	552.16	548.86	562.18	52.5TiB/s	474.0GiB/s	689.8GiB/s	82.3%	5104.63 (68.5%)	6.9	(128, 256, 256)	(128, 1, 1)	2	✗
$\Lambda_{8,4}$	(512, 512, 512)	4297.15	4286.46	4310.02	49.5TiB/s	476.3GiB/s	703.6GiB/s	83.9%	5247.33 (70.4%)	6.9	(256, 256, 512)	(256, 1, 1)	2	✗
remesh	discretization	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	P	A

Table C.4 – 2D and 3D double-precision directional remesh kernel statistics. Benchmark of remeshing kernels $\Lambda_{p,r}$ on the `NVidia Tesla V100-SXM2 GPU`. Kernel parameter `P` represent the number of remeshed particles and `A` that atomic intrinsics are used.

Configuration		Runtime (us)			Mean Memory Bandwidth			Mean Achieved Performance			Kernel Configuration			
integrator	order	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP32)	FLOP/B	global_size	local_size	V	S
Euler	FDC2	1555.14	1549.31	1569.79	49.0TiB/s	2.2TiB/s	644.3GiB/s	76.9%	1294.59 (8.7%)	1.87	(256, 512, 32)	(256, 1, 1)	2	✓
Euler	FDC4	1631.20	1627.14	1643.52	44.4TiB/s	2.7TiB/s	615.4GiB/s	73.4%	1439.93 (9.7%)	2.18	(64, 512, 512)	(64, 1, 1)	2	✓
Euler	FDC6	1861.57	1855.49	1872.90	49.4TiB/s	2.9TiB/s	540.3GiB/s	64.5%	1658.28 (11.1%)	2.86	(256, 512, 32)	(256, 1, 1)	2	✓
Euler	FDC8	1973.18	1970.18	1988.61	42.5TiB/s	3.3TiB/s	510.8GiB/s	60.9%	1734.53 (11.6%)	3.16	(64, 512, 256)	(64, 1, 1)	2	✓
RK2	FDC2	1778.53	1775.62	1789.95	54.5TiB/s	3.7TiB/s	564.5GiB/s	67.3%	2301.70 (15.4%)	3.80	(64, 512, 128)	(64, 1, 1)	2	✓
RK2	FDC4	2318.50	2311.17	2326.53	49.8TiB/s	3.9TiB/s	434.7GiB/s	51.9%	2344.54 (15.7%)	5.02	(64, 512, 64)	(64, 1, 1)	2	✓
RK2	FDC6	1945.63	1943.55	1954.82	108.1TiB/s	5.5TiB/s	520.0GiB/s	62.0%	3293.99 (22.1%)	5.90	(64, 512, 512)	(64, 1, 1)	1	✓
RK2	FDC8	2001.47	1996.80	2015.23	134.3TiB/s	7.0TiB/s	507.4GiB/s	60.5%	4258.28 (28.6%)	7.82	(128, 512, 256)	(128, 1, 1)	1	✓
RK4	FDC2	2168.38	2165.76	2176.00	104.8TiB/s	4.9TiB/s	464.8GiB/s	55.4%	3930.50 (26.4%)	7.88	(64, 512, 256)	(64, 1, 1)	1	✓
RK4	FDC4	2295.01	2290.69	2304.00	138.8TiB/s	7.2TiB/s	442.5GiB/s	52.8%	5321.90 (35.7%)	11.20	(128, 512, 64)	(128, 1, 1)	1	✓
RK4	FDC6	2767.81	2765.82	2774.02	143.3TiB/s	7.8TiB/s	369.8GiB/s	44.1%	5382.66 (36.1%)	13.56	(128, 512, 512)	(128, 1, 1)	1	✓
RK4	FDC8	3466.53	3463.17	3476.48	130.6TiB/s	7.7TiB/s	297.5GiB/s	35.5%	5072.08 (34.0%)	15.88	(128, 512, 256)	(128, 1, 1)	1	✓
RK4_38	FDC2	2200.13	2196.48	2209.79	102.6TiB/s	4.8TiB/s	458.1GiB/s	54.6%	4285.57 (28.8%)	8.71	(64, 512, 64)	(64, 1, 1)	1	✓
RK4_38	FDC4	2327.36	2320.38	2335.74	137.2TiB/s	7.1TiB/s	436.4GiB/s	52.1%	5680.45 (38.1%)	12.12	(128, 512, 32)	(128, 1, 1)	1	✓
RK4_38	FDC6	2954.56	2952.19	2966.53	132.2TiB/s	7.3TiB/s	346.4GiB/s	41.3%	5383.14 (36.1%)	14.47	(128, 512, 256)	(128, 1, 1)	1	✓
RK4_38	FDC8	3597.44	3585.02	3608.58	124.7TiB/s	7.4TiB/s	286.7GiB/s	34.2%	5167.33 (34.7%)	16.79	(128, 512, 64)	(128, 1, 1)	1	✓
integrator	order	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	V	S
Euler	FDC2	2856.64	2852.86	2865.15	41.7TiB/s	2.4TiB/s	701.5GiB/s	83.7%	704.77 (9.5%)	0.94	(256, 512, 512)	(256, 1, 1)	2	✓
Euler	FDC4	2961.09	2953.22	2970.62	43.5TiB/s	3.0TiB/s	678.1GiB/s	80.9%	861.22 (11.6%)	1.18	(256, 512, 512)	(256, 1, 1)	2	✓
Euler	FDC6	3010.18	3004.42	3023.87	46.0TiB/s	3.6TiB/s	668.3GiB/s	79.7%	1025.52 (13.8%)	1.43	(256, 512, 512)	(256, 1, 1)	2	✓
Euler	FDC8	3076.51	3069.95	3084.29	48.2TiB/s	4.1TiB/s	655.2GiB/s	78.2%	1177.92 (15.8%)	1.67	(256, 512, 512)	(256, 1, 1)	2	✓
RK2	FDC2	4274.08	4261.89	4283.39	25.0TiB/s	2.8TiB/s	469.8GiB/s	56.0%	855.72 (11.5%)	1.70	(32, 512, 512)	(32, 1, 1)	2	✓
RK2	FDC4	5169.02	5156.86	5183.49	25.0TiB/s	3.2TiB/s	389.9GiB/s	46.5%	941.26 (12.6%)	2.25	(32, 512, 512)	(32, 1, 1)	2	✓
RK2	FDC6	6961.57	6940.67	6983.68	23.8TiB/s	3.4TiB/s	290.7GiB/s	34.7%	968.81 (13.0%)	3.10	(32, 512, 256)	(32, 1, 1)	2	✓
RK2	FDC8	7327.36	7310.34	7348.22	26.3TiB/s	3.9TiB/s	277.2GiB/s	33.1%	1103.62 (14.8%)	3.71	(32, 512, 512)	(32, 1, 1)	2	✓
RK4	FDC2	6852.29	6837.25	6870.02	28.3TiB/s	3.4TiB/s	294.2GiB/s	35.1%	1278.07 (17.2%)	4.05	(32, 512, 512)	(32, 1, 1)	2	✓
RK4	FDC4	8522.08	8504.32	8546.30	28.7TiB/s	3.9TiB/s	238.4GiB/s	28.4%	1342.64 (18.0%)	5.25	(32, 512, 512)	(32, 1, 1)	2	✓
RK4	FDC6	9445.31	9426.94	9459.71	31.0TiB/s	4.6TiB/s	216.7GiB/s	25.9%	1499.15 (20.1%)	6.44	(64, 512, 128)	(64, 1, 1)	2	✓
RK4	FDC8	8662.56	8657.92	8670.21	61.3TiB/s	6.1TiB/s	238.1GiB/s	28.4%	2029.71 (27.2%)	7.94	(128, 512, 256)	(128, 1, 1)	1	✓
RK4_38	FDC2	6867.33	6852.61	6888.45	28.8TiB/s	3.4TiB/s	293.5GiB/s	35.0%	1421.85 (19.1%)	4.51	(32, 512, 512)	(32, 1, 1)	2	✓
RK4_38	FDC4	8572.74	8541.18	8595.46	28.9TiB/s	3.9TiB/s	236.9GiB/s	28.3%	1452.13 (19.5%)	5.71	(32, 512, 512)	(32, 1, 1)	2	✓
RK4_38	FDC6	9090.21	9078.78	9105.41	33.4TiB/s	4.7TiB/s	225.2GiB/s	26.9%	1668.46 (22.4%)	6.90	(64, 512, 512)	(64, 1, 1)	2	✓
RK4_38	FDC8	8605.41	8599.55	8616.96	63.0TiB/s	6.2TiB/s	239.7GiB/s	28.6%	2160.17 (29.0%)	8.39	(128, 512, 512)	(128, 1, 1)	1	✓

Table C.5 – 3D single- and double-precision directional diffusion kernel statistics. Benchmark of diffusion on arrays of 512^3 elements on the `NVidia Tesla V100-SXM2 GPU`. Kernel parameter `V` represent the vectorization and `S` that shared memory is used.

Configuration		Runtime (us)			Mean Memory Bandwidth			Mean Achieved Performance			Kernel Configuration			
integrator	order	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP32)	FLOP/B	global_size	local_size	V	S
Euler	FDC2	6645.73	6642.69	6656.00	49.5TiB/s	1.4TiB/s	678.3GiB/s	80.9%	716.96 (4.8%)	0.98	(64, 512, 512)	(64, 1, 1)	1	✓
Euler	FDC4	6785.41	6774.78	6795.26	65.6TiB/s	2.2TiB/s	665.5GiB/s	79.4%	1068.14 (7.2%)	1.49	(128, 512, 256)	(128, 1, 1)	1	✓
Euler	FDC6	6888.38	6882.30	6895.62	78.5TiB/s	3.1TiB/s	656.7GiB/s	78.3%	1393.15 (9.4%)	1.98	(64, 512, 512)	(64, 1, 1)	1	✓
Euler	FDC8	6961.76	6956.03	6980.61	92.9TiB/s	3.9TiB/s	650.9GiB/s	77.7%	1725.50 (11.6%)	2.47	(64, 512, 512)	(64, 1, 1)	1	✓
RK2	FDC2	12774.18	12744.70	12832.77	28.8TiB/s	1.9TiB/s	353.5GiB/s	42.2%	819.54 (5.5%)	2.16	(64, 512, 256)	(64, 1, 1)	2	✓
RK2	FDC4	11617.06	11575.30	11650.05	70.0TiB/s	3.1TiB/s	390.1GiB/s	46.5%	1357.54 (9.1%)	3.24	(64, 512, 256)	(64, 1, 1)	1	✓
RK2	FDC6	12685.22	12656.64	12726.27	83.6TiB/s	3.9TiB/s	358.4GiB/s	42.8%	1671.74 (11.2%)	4.34	(64, 512, 512)	(64, 1, 1)	1	✓
RK2	FDC8	12121.06	12105.73	12140.54	120.7TiB/s	5.7TiB/s	376.4GiB/s	44.9%	2447.16 (16.4%)	6.05	(128, 512, 512)	(128, 1, 1)	1	✓
RK4	FDC2	11849.73	11812.86	11880.45	79.4TiB/s	3.4TiB/s	383.1GiB/s	45.7%	1942.52 (13.0%)	4.72	(64, 512, 256)	(64, 1, 1)	1	✓
RK4	FDC4	13222.24	13204.48	13255.68	119.5TiB/s	5.5TiB/s	346.2GiB/s	41.3%	2852.40 (19.1%)	7.67	(128, 512, 128)	(128, 1, 1)	1	✓
RK4	FDC6	16162.18	16131.07	16195.58	130.2TiB/s	6.4TiB/s	285.7GiB/s	34.1%	3080.95 (20.7%)	10.04	(128, 512, 64)	(128, 1, 1)	1	✓
RK4	FDC8	19650.02	19612.67	19695.62	134.0TiB/s	6.7TiB/s	237.0GiB/s	28.3%	3148.82 (21.1%)	12.38	(128, 512, 64)	(128, 1, 1)	1	✓
RK4_38	FDC2	11640.93	11588.61	11680.77	80.9TiB/s	3.4TiB/s	389.9GiB/s	46.5%	2210.84 (14.8%)	5.28	(64, 512, 128)	(64, 1, 1)	1	✓
RK4_38	FDC4	13317.50	13297.66	13337.60	119.0TiB/s	5.5TiB/s	343.8GiB/s	41.0%	3058.76 (20.5%)	8.29	(128, 512, 128)	(128, 1, 1)	1	✓
RK4_38	FDC6	16211.36	16180.22	16241.66	130.2TiB/s	6.3TiB/s	284.8GiB/s	34.0%	3257.88 (21.9%)	10.65	(128, 512, 64)	(128, 1, 1)	1	✓
RK4_38	FDC8	19820.54	19780.61	19885.06	133.1TiB/s	6.7TiB/s	234.9GiB/s	28.0%	3274.09 (22.0%)	12.98	(128, 512, 64)	(128, 1, 1)	1	✓
integrator	order	mean	min	max	private	local	global	BDW (%)	GFLOPS (FP64)	FLOP/B	global_size	local_size	V	S
Euler	FDC2	12758.98	12730.37	12782.59	43.6TiB/s	1.5TiB/s	706.6GiB/s	84.3%	410.26 (5.5%)	0.54	(512, 512, 512)	(512, 1, 1)	1	✓
Euler	FDC4	13422.98	13412.35	13436.93	35.2TiB/s	2.3TiB/s	672.8GiB/s	80.3%	534.95 (7.2%)	0.74	(64, 512, 256)	(64, 1, 1)	1	✓
Euler	FDC6	13401.54	13390.85	13419.52	59.0TiB/s	3.1TiB/s	675.1GiB/s	80.5%	751.13 (10.1%)	1.04	(512, 512, 512)	(512, 1, 1)	1	✓
Euler	FDC8	13221.44	13209.60	13236.22	54.2TiB/s	4.1TiB/s	685.4GiB/s	81.8%	908.56 (12.2%)	1.23	(64, 512, 512)	(64, 1, 1)	1	✓
RK2	FDC2	16090.85	16072.70	16110.59	40.7TiB/s	2.8TiB/s	561.3GiB/s	67.0%	642.28 (8.6%)	1.07	(64, 512, 512)	(64, 1, 1)	1	✓
RK2	FDC4	18412.93	18384.90	18440.19	49.2TiB/s	3.9TiB/s	492.2GiB/s	58.7%	856.50 (11.5%)	1.62	(64, 512, 128)	(64, 1, 1)	1	✓
RK2	FDC6	20662.05	20622.34	20709.38	57.3TiB/s	4.8TiB/s	440.1GiB/s	52.5%	1026.35 (13.8%)	2.17	(64, 512, 512)	(64, 1, 1)	1	✓
RK2	FDC8	21962.69	21924.86	21996.54	72.9TiB/s	6.3TiB/s	415.5GiB/s	49.6%	1350.57 (18.1%)	3.03	(128, 512, 64)	(128, 1, 1)	1	✓
RK4	FDC2	19338.21	19295.23	19363.84	56.0TiB/s	4.1TiB/s	469.4GiB/s	56.0%	1190.30 (16.0%)	2.36	(64, 512, 256)	(64, 1, 1)	1	✓
RK4	FDC4	22885.92	22849.54	22929.41	78.3TiB/s	6.4TiB/s	400.1GiB/s	47.7%	1647.96 (22.1%)	3.84	(128, 512, 64)	(128, 1, 1)	1	✓
RK4	FDC6	27347.74	27272.19	27399.17	87.0TiB/s	7.5TiB/s	337.7GiB/s	40.3%	1820.80 (24.4%)	5.02	(128, 512, 32)	(128, 1, 1)	1	✓
RK4	FDC8	32270.72	32228.35	32302.08	92.3TiB/s	8.2TiB/s	288.6GiB/s	34.4%	1917.35 (25.7%)	6.19	(128, 512, 256)	(128, 1, 1)	1	✓
RK4_38	FDC2	19638.59	19610.62	19680.26	55.7TiB/s	4.1TiB/s	462.3GiB/s	55.1%	1310.49 (17.6%)	2.64	(64, 512, 256)	(64, 1, 1)	1	✓
RK4_38	FDC4	22992.03	22963.20	23011.33	78.6TiB/s	6.4TiB/s	398.2GiB/s	47.5%	1771.70 (23.8%)	4.14	(128, 512, 128)	(128, 1, 1)	1	✓
RK4_38	FDC6	27566.40	27531.26	27604.99	87.2TiB/s	7.5TiB/s	335.0GiB/s	40.0%	1915.91 (25.7%)	5.33	(128, 512, 256)	(128, 1, 1)	1	✓
RK4_38	FDC8	32771.71	32734.21	32804.86	91.3TiB/s	8.1TiB/s	284.2GiB/s	33.9%	1980.19 (26.6%)	6.49	(128, 512, 256)	(128, 1, 1)	1	✓

Table C.6 – 3D single- and double-precision directional stretching kernel statistics. Benchmark of stretching on arrays of 512^3 elements on the `NVidia Tesla V100-SXM2 GPU`. Kernel parameter `V` represent the vectorization and `S` that shared memory is used.

Bibliography

Abdelhakim, Lotfi (2018). "FEEL++ applications to engineering problems". In: *AIP Conference Proceedings*. Vol. 1978. 1. AIP Publishing, p. 470068 (cit. on p. 55).

Aji, Ashwin M, James Dinan, Darius Buntinas, Pavan Balaji, Wu-chun Feng, Keith R Bisset, and Rajeev Thakur (2012). "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems". In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE, pp. 647–654 (cit. on p. 223).

Aji, Ashwin Mandayam, Antonio J Pena, Pavan Balaji, and Wu-chun Feng (2015). "Automatic command queue scheduling for task-parallel workloads in opencl". In: *2015 IEEE International Conference on Cluster Computing*. IEEE, pp. 42–51 (cit. on p. 223).

Amada, Takashi, Masataka Imura, Yoshihiro Yasumuro, Yoshitsugu Manabe, and Kunihiro Chihara (2004). "Particle-based fluid simulation on GPU". In: *ACM workshop on general-purpose computing on graphics processors*. Vol. 41, p. 42 (cit. on p. 56).

Anderson, Christopher and Claude Greengard (1985). "On vortex methods". In: *SIAM journal on numerical analysis* 22.3, pp. 413–440 (cit. on p. 19).

Angot, Philippe, Charles-Henri Bruneau, and Pierre Fabrie (1999). "A penalization method to take into account obstacles in incompressible viscous flows". In: *Numerische Mathematik* 81.4, pp. 497–520 (cit. on p. 70).

Antao, Samuel F, Alexey Bataev, Arpith C Jacob, Gheorghe-Teodor Bercea, Alexandre E Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, et al. (2016). "Offloading support for OpenMP in Clang and LLVM". In: *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*. IEEE Press, pp. 1–11 (cit. on p. 47).

Apte, Sourabh V, Krishnan Mahesh, Parviz Moin, and Joseph C Oefelein (2003). "Large-eddy simulation of swirling particle-laden flows in a coaxial-jet combustor". In: *International Journal of Multiphase Flow* 29.8, pp. 1311–1331 (cit. on p. 28).

Arge, Erlend, A Bruaset, and Hans Petter Langtangen (1997). *Modern software tools for scientific computing*. Springer Science & Business Media (cit. on p. 133).

Arora, Manish (2012). "The architecture and evolution of cpu-gpu systems for general purpose computing". In: *By University of California, San Diago* 27 (cit. on p. 45).

Balarac, Guillaume, G-H Cottet, J-M Etancelin, J-B Lagaert, Franck Pérignon, and Christophe Picard (2014). "Multi-scale problems, high performance computing and hybrid numerical methods". In: *The Impact of Applications on Mathematics*. Springer, pp. 245–255 (cit. on p. 134).

Baqais, A, M Assayony, A Khan, and M Al-Mouhamed (2013). "Bank Conflict-Free Access for CUDA-Based Matrix Transpose Algorithm on GPUs". In: *International Conference on Computer Applications Technology* (cit. on p. 160).

Baranger, J, M Garbey, and F Oudin-Dardun (2003). "On Aitken like acceleration of Schwarz domain decomposition method using generalized Fourier". In: *Domain Decomposition Methods in Science and Engineering*, pp. 341–348 (cit. on p. 223).

Barker, Brandon (2015). “Message passing interface (mpi)”. In: *Workshop: High Performance Computing on Stampede*. Vol. 262 (cit. on p. 47).

Barney, Blaise (2009). “POSIX threads programming”. In: *National Laboratory. Disponível em:¡ https://computing. llnl. gov/tutorials/pthreads/¿ Acesso em* 5, p. 46 (cit. on p. 48).

Bassi, Caterina, Antonella Abbà, Luca Bonaventura, and Lorenzo Valdettaro (2018). “Direct and Large Eddy Simulation of three-dimensional non-Boussinesq gravity currents with a high order DG method”. In: *arXiv preprint arXiv:1804.04958* (cit. on p. 31).

Batchelor, George K (1959). “Small-scale variation of convected quantities like temperature in turbulent fluid Part 1. General discussion and the case of small conductivity”. In: *Journal of Fluid Mechanics* 5.1, pp. 113–133 (cit. on pp. 3, 20).

Beale, J Thomas and Andrew Majda (1982). “Vortex methods. II. Higher order accuracy in two and three dimensions”. In: *Mathematics of Computation* 39.159, pp. 29–52 (cit. on p. 64).

Beazley, David M et al. (1996). “SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++.” In: *Tcl/Tk Workshop*, p. 43 (cit. on p. 232).

Beckermann, C and R Viskanta (1988). “Double-diffusive convection during dendritic solidification of a binary mixture”. In: *PhysicoChemical Hydrodynamics* 10.2, pp. 195–213 (cit. on p. 33).

Behnel, Stefan, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith (2011). “Cython: The best of both worlds”. In: *Computing in Science & Engineering* 13.2, p. 31 (cit. on pp. 58, 157).

Bell, Nathan and Jared Hoberock (2012). “Thrust: A productivity-oriented library for CUDA”. In: *GPU computing gems Jade edition*. Elsevier, pp. 359–371 (cit. on p. 151).

Benjamin, T Brooke (1968). “Gravity currents and related phenomena”. In: *Journal of Fluid Mechanics* 31.2, pp. 209–248 (cit. on p. 29).

Bercovici, David (1994). “A theoretical model of cooling viscous gravity currents with temperature-dependent viscosity”. In: *Geophysical Research Letters* 21.12, pp. 1177–1180 (cit. on p. 29).

Bergstrom, Lars (2011). “Measuring NUMA effects with the STREAM benchmark”. In: *arXiv preprint arXiv:1103.3225* (cit. on p. 164).

Bhat, Krishnaraj (2017). *clpeak: A tool which profiles OpenCL devices to find their peak capacities.(2017)* (cit. on pp. 50, 232).

Biegert, Edward, Bernhard Vowinckel, and Eckart Meiburg (2017). “A collision model for grain-resolving simulations of flows over dense, mobile, polydisperse granular sediment beds”. In: *Journal of Computational Physics* 340, pp. 105–127 (cit. on p. 28).

Birman, VK and E Meiburg (2006). “High-resolution simulations of gravity currents”. In: *Journal of the Brazilian Society of Mechanical Sciences and Engineering* 28.2, pp. 169–173 (cit. on p. 31).

Birman, VK, E Meiburg, and M Ungarish (2007). “On gravity currents in stratified ambients”. In: *Physics of Fluids* 19.8, p. 086602 (cit. on p. 31).

Blaauw, Gerrit A and Frederick P Brooks Jr (1997). *Computer architecture: Concepts and evolution*. Addison-Wesley Longman Publishing Co., Inc. (cit. on p. 41).

Bluestein, Leo (1970). "A linear filtering approach to the computation of discrete Fourier transform". In: *IEEE Transactions on Audio and Electroacoustics* 18.4, pp. 451–455 (cit. on p. 99).

Bode, Brett M, Jason J Hill, and Troy R Benjegerdes (2004). "Cluster interconnect overview". In: *Proceedings of USENIX 2004 Annual Technical Conference, FREENIX Track*, pp. 217–223 (cit. on p. 187).

Boivin, Marc, Olivier Simonin, and Kyle D Squires (1998). "Direct numerical simulation of turbulence modulation by particles in isotropic turbulence". In: *Journal of Fluid Mechanics* 375, pp. 235–263 (cit. on p. 28).

Bonnecaze, Roger T, Herbert E Huppert, and John R Lister (1993). "Particle-driven gravity currents". In: *Journal of Fluid Mechanics* 250, pp. 339–369 (cit. on p. 29).

Bonnecaze, Roger T and John R Lister (1999). "Particle-driven gravity currents down planar slopes". In: *Journal of Fluid Mechanics* 390, pp. 75–91 (cit. on p. 30).

Boussinesq, Joseph (1877). *Essai sur la théorie des eaux courantes*. Impr. nationale (cit. on p. 14).

Brandvik, Tobias and Graham Pullan (2010). "SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms". In: *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, pp. 1181–1188 (cit. on p. 90).

Britter, RE and JE Simpson (1978). "Experiments on the dynamics of a gravity current head". In: *Journal of Fluid Mechanics* 88.2, pp. 223–240 (cit. on p. 29).

Brown, David L (1995). "Performance of under-resolved two-dimensional incompressible flow simulations". In: *Journal of Computational Physics* 122.1, pp. 165–183 (cit. on p. 145).

Burns, Keaton J, Geoffrey M Vasil, Jeffrey S Oishi, Daniel Lecoanet, and Benjamin Brown (2016). "Dedalus: Flexible framework for spectrally solving differential equations". In: *Astrophysics Source Code Library* (cit. on p. 57).

Burns, P and E Meiburg (2012). "Sediment-laden fresh water above salt water: linear stability analysis". In: *Journal of Fluid Mechanics* 691, pp. 279–314 (cit. on pp. 2, 37, 197, 198, 205, 210).

— (2015). "Sediment-laden fresh water above salt water: nonlinear simulations". In: *Journal of Fluid Mechanics* 762, pp. 156–195 (cit. on pp. 2, 37, 40, 199, 201, 204, 206–210, 212, 213, 216, 217, 224).

Butcher, John C (1963). "Coefficients for the study of Runge-Kutta integration processes". In: *Journal of the Australian Mathematical Society* 3.2, pp. 185–201 (cit. on p. 81).

Butcher, John Charles and JC Butcher (1987). *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Vol. 512. Wiley New York (cit. on p. 81).

Buyevich, Yu A (1971). "Statistical hydromechanics of disperse systems Part 1. Physical background and general equations". In: *Journal of Fluid Mechanics* 49.3, pp. 489–507 (cit. on p. 27).

Caldwell, Douglas R (1973). "Thermal and Fickian diffusion of sodium chloride in a solution of oceanic concentration". In: *Deep Sea Research and Oceanographic Abstracts*. Vol. 20. 11. Elsevier, pp. 1029–1039 (cit. on p. 35).

Canon, Louis-Claude, Loris Marchal, Bertrand Simon, and Frédéric Vivien (2018). "Online scheduling of task graphs on hybrid platforms". In: *European Conference on Parallel Processing*. Springer, pp. 192–204 (cit. on p. 146).

Canuto, Claudio, M Yousuff Hussaini, Alfio Quarteroni, A Thomas Jr, et al. (2012). *Spectral methods in fluid dynamics*. Springer Science & Business Media (cit. on p. 71).

Cao, Chongxiao, Jack Dongarra, Peng Du, Mark Gates, Piotr Luszczek, and Stanimire Tomov (2014). "clMAGMA: High performance dense linear algebra with OpenCL". In: *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, p. 1 (cit. on p. 159).

Carvalho, Carlos (2002). "The gap between processor and memory speeds". In: *Proc. of IEEE International Conference on Control and Automation* (cit. on p. 43).

Cary, John R, Svetlana G Shasharina, Julian C Cummings, John VW Reynders, and Paul J Hinker (1997). "Comparison of C++ and Fortran 90 for object-oriented scientific programming". In: *Computer Physics Communications* 105.1, pp. 20–36 (cit. on p. 133).

Çengel, Yunus A, Robert H Turner, and John M Cimbala (2001). *Fundamentals of thermal-fluid sciences*. Vol. 703. McGraw-Hill New York (cit. on p. 17).

Cercos-Pita, Jose L (2015). "AQUAgpusph, a new free 3D SPH solver accelerated with OpenCL". In: *Computer Physics Communications* 192, pp. 295–312 (cit. on p. 56).

Chan, S-C and K-L Ho (1992). "Fast algorithms for computing the discrete cosine transform". In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39.3, pp. 185–190 (cit. on p. 176).

Chan, SC and KL Ho (1990). "Direct methods for computing discrete sinusoidal transforms". In: *IEE Proceedings F (Radar and Signal Processing)*. Vol. 137. 6. IET, pp. 433–442 (cit. on p. 176).

Chatelain, Philippe, Alessandro Curioni, Michael Bergdorf, Diego Rossinelli, Wanda Andreoni, and Petros Koumoutsakos (2008a). "Billion vortex particle direct numerical simulations of aircraft wakes". In: *Computer Methods in Applied Mechanics and Engineering* 197.13-16, pp. 1296–1304 (cit. on p. 56).

— (2008b). "Vortex methods for massively parallel computer architectures". In: *International Conference on High Performance Computing for Computational Science*. Springer, pp. 479–489 (cit. on p. 56).

Chauchat, Julien, Zhen Cheng, Tim Nagel, Cyrille Bonamy, and Tian-Jian Hsu (2017). "SedFoam-2.0: a 3-D two-phase flow numerical model for sediment transport". In: *Geoscientific Model Development* 10.12 (cit. on p. 55).

Chen, CF and DH Johnson (1984). "Double-diffusive convection: a report on an engineering foundation conference". In: *Journal of Fluid Mechanics* 138, pp. 405–416 (cit. on p. 33).

Chen, Chen-Tung and Frank J Millero (1977). "Precise equation of state of seawater for oceanic ranges of salinity, temperature and pressure". In: *Deep Sea Research* 24.4, pp. 365–369 (cit. on pp. 33, 34).

Chen, Goong, Qingang Xiong, Philip J Morris, Eric G Paterson, Alexey Sergeev, and Y Wang (2014). "OpenFOAM for computational fluid dynamics". In: *Not. AMS* 61.4, pp. 354–363 (cit. on p. 55).

Chorin, Alexandre Joel (1973). "Numerical study of slightly viscous flow". In: *Journal of fluid mechanics* 57.4, pp. 785–796 (cit. on pp. 19, 64).

— (1978). "Vortex sheet approximation of boundary layers". In: *Journal of computational physics* 27.3, pp. 428–442 (cit. on p. 19).

Chrzeszczyk, Andrzej (2017). *Matrix Computations on the GPU with ArrayFire-Python and ArrayFire-C/C++* (cit. on p. 151).

Cohen, Nadia, Pierre-Yves Passaggia, Alberto Scotti, and Brian White (2018). "Experiments on turbulent horizontal convection at high Schmidt numbers". In: *Bulletin of the American Physical Society* 63 (cit. on p. 224).

Compute, NVIDIA (2010). "PTX: Parallel thread execution ISA version 2.3". In: *Dostopno na: http://developer. download. nvidia. com/compute/cuda* 3 (cit. on p. 48).

Cooke, David and Timothy Hochberg (2017). *Numexpr. Fast evaluation of array expressions by using a vector-based virtual machine* (cit. on p. 157).

Cooley, James W and John W Tukey (1965). "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90, pp. 297–301 (cit. on pp. 98, 99).

Coquerelle, Mathieu and G-H Cottet (2008). "A vortex level set method for the two-way coupling of an incompressible fluid with colliding rigid bodies". In: *Journal of Computational Physics* 227.21, pp. 9121–9137 (cit. on p. 28).

Costa, Pedro (2018). "A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows". In: *Computers & Mathematics with Applications* 76.8, pp. 1853–1862 (cit. on p. 57).

Cottet, G-H, Guillaume Balarac, and Mathieu Coquerelle (2009a). "Subgrid particle resolution for the turbulent transport of a passive scalar". In: *Advances in Turbulence XII*. Springer, pp. 779–782 (cit. on p. 21).

Cottet, G-H, J-M Etancelin, Franck Pérignon, and Christophe Picard (2014). "High order semi-Lagrangian particle methods for transport equations: numerical analysis and implementation issues". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 48.4, pp. 1029–1060 (cit. on pp. 75, 82, 134).

Cottet, Georges-Henri (1988). "A new approach for the analysis of vortex methods in two and three dimensions". In: *Annales de l'Institut Henri Poincare (C) Non Linear Analysis*. Vol. 5. 3. Elsevier, pp. 227–285 (cit. on p. 19).

— (2016). "Semi-Lagrangian particle methods for hyperbolic problems". In: (cit. on p. 3).

Cottet, Georges-Henri, Petros D Koumoutsakos, Petros D Koumoutsakos, et al. (2000). *Vortex methods: theory and practice*. Cambridge university press (cit. on pp. 19, 62, 64, 71, 74).

Cottet, Georges-Henri and Adrien Magni (2009b). "TVD remeshing formulas for particle methods". In: *Comptes Rendus Mathematique* 347.23-24, pp. 1367–1372 (cit. on pp. 82, 134).

Cottet, Georges-Henri, Jean-Matthieu Etancelin, Franck Perignon, Christophe Picard, Florian De Vuyst, and Christophe Labourdette (2013). "Is GPU the future of Scientific Computing ?" en. In: *Annales mathématiques Blaise Pascal* 20.1, pp. 75–99 (cit. on p. 71).

Courant, Richard, Eugene Isaacson, and Mina Rees (1952). "On the solution of nonlinear hyperbolic differential equations by finite differences". In: *Communications on Pure and Applied Mathematics* 5.3, pp. 243–255 (cit. on p. 71).

Courant, Richard, Kurt Friedrichs, and Hans Lewy (1967). "On the partial difference equations of mathematical physics". In: *IBM journal of Research and Development* 11.2, pp. 215–234 (cit. on p. 66).

Crank, John and Phyllis Nicolson (1947). "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type". In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 43. 1. Cambridge University Press, pp. 50–67 (cit. on p. 89).

Cummins, Chris, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather (2015). "Autotuning OpenCL workgroup size for stencil patterns". In: *arXiv preprint arXiv:1511.02490* (cit. on p. 90).

Dagum, Leonardo and Ramesh Menon (1998). "OpenMP: An industry-standard API for shared-memory programming". In: *Computing in Science & Engineering* 1, pp. 46–55 (cit. on p. 47).

Dalcín, Lisandro, Rodrigo Paz, and Mario Storti (2005). "MPI for Python". In: *Journal of Parallel and Distributed Computing* 65.9, pp. 1108–1115 (cit. on p. 230).

Dalcin, Lisandro, Mikael Mortensen, and David E Keyes (2019). "Fast parallel multidimensional FFT using advanced MPI". In: *Journal of Parallel and Distributed Computing* 128, pp. 137–150 (cit. on pp. 193, 223).

Datta, Kaushik, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick (2008). "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures". In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, p. 4 (cit. on pp. 90, 159).

Datta, Kaushik and Katherine A Yelick (2009). *Auto-tuning stencil codes for cache-based multicore platforms*. University of California, Berkeley (cit. on p. 159).

Davies, D Rhodri, Cian R Wilson, and Stephan C Kramer (2011). "Fluidity: A fully unstructured anisotropic adaptive mesh computational modeling framework for geodynamics". In: *Geochemistry, Geophysics, Geosystems* 12.6 (cit. on p. 55).

De Vuyst, Florian (2016). "Performance modeling of a compressible hydrodynamics solver on multicore CPUs". In: *Parallel Computing: On the Road to Exascale* 27, p. 449 (cit. on p. 56).

De Vuyst, Florian, Thibault Gasc, Renaud Motte, Mathieu Peybernes, and Raphaël Poncet (2016). "Lagrange-flux schemes: reformulating second-order accurate Lagrange-remap schemes for better node-based HPC performance". In: *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles* 71.6, p. 64 (cit. on p. 56).

Deacon, GER (1945). "Water circulation and surface boundaries in the oceans". In: *Quarterly Journal of the Royal Meteorological Society* 71.307-308, pp. 11–27 (cit. on p. 33).

Demidov, Denis (2012). *VexCL: Vector expression template library for OpenCL* (cit. on p. 151).

Derevich, IV and LI Zaichik (1988). "Particle deposition from a turbulent flow". In: *Fluid Dynamics* 23.5, pp. 722–729 (cit. on p. 27).

Dolbeau, Romain, François Bodin, and Guillaume Colin de Verdiere (2013). "One OpenCL to rule them all?" In: *2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*. IEEE, pp. 1–6 (cit. on pp. 52, 159).

Dongarra, Jack and Michael A Heroux (2013). "Toward a new metric for ranking high performance computing systems". In: *Sandia Report, SAND2013-4744* 312, p. 150 (cit. on p. 43).

Dongarra, Jack J, Piotr Luszczek, and Antoine Petitet (2003). "The LINPACK benchmark: past, present and future". In: *Concurrency and Computation: practice and experience* 15.9, pp. 803–820 (cit. on p. 42).

Donzis, DA and PK Yeung (2010). "Resolution effects and scaling in numerical simulations of passive scalar mixing in turbulence". In: *Physica D: Nonlinear Phenomena* 239.14, pp. 1278–1287 (cit. on p. 20).

Drew, Donald A and Stephen L Passman (2006). *Theory of multicomponent fluids*. Vol. 135. Springer Science & Business Media (cit. on p. 26).

Du, Peng, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra (2012). "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming". In: *Parallel Computing* 38.8, pp. 391–407 (cit. on p. 159).

Duhamel, Pierre and Henk Hollmann (1984). "Split radix'FFT algorithm". In: *Electronics letters* 20.1, pp. 14–16 (cit. on p. 99).

Duncan, Ralph (1990). "A survey of parallel computer architectures". In: *Computer* 23.2, pp. 5–16 (cit. on p. 51).

Duvall, Paul M, Steve Matyas, and Andrew Glover (2007). *Continuous integration: improving software quality and reducing risk*. Pearson Education (cit. on p. 151).

Edmonds, Douglas A and Rudy L Slingerland (2010). "Significant effect of sediment cohesion on delta morphology". In: *Nature Geoscience* 3.2, p. 105 (cit. on p. 22).

Elghobashi, Said (1983). "A two-equation turbulence model for two-phase flows". In: *Physics of Fluids* 26.4, p. 931 (cit. on p. 27).

— (1991). "Particle-laden turbulent flows: direct simulation and closure models". In: *Applied Scientific Research* 48.3-4, pp. 301–314 (cit. on p. 24).

— (1994). "On predicting particle-laden turbulent flows". In: *Applied scientific research* 52.4, pp. 309–329 (cit. on p. 25).

Etancelin, Jean-Matthieu (Dec. 2014). "Model coupling and hybrid computing for multi-scale CFD". Theses. Université de Grenoble (cit. on pp. 3, 69, 83, 92, 133, 135, 146, 149, 152, 165).

Etancelin, Jean-Matthieu, Georges-Henri Cottet, Franck Pérignon, and Christophe Picard (May 2014). "Multi-CPU and multi-GPU hybrid computations of multi-scale scalar transport". In: *26th International Conference on Parallel Computational Fluid Dynamics*. Trondheim, Norway, pp. 83–84 (cit. on pp. 3, 19, 57, 74, 134).

Euler, Leonhard (1757). "Principes généraux de l'état d'équilibre des fluides". In: *Mémoires de l'académie des sciences de Berlin*, pp. 217–273 (cit. on p. 7).

Faas, Richard W (1984). "Time and density-dependent properties of fluid mud suspensions, NE Brazilian continental shelf". In: *Geo-Marine Letters* 4.3-4, pp. 147–152 (cit. on p. 23).

Faeth, Gerard M (1987). "Mixing, transport and combustion in sprays". In: *Progress in energy and combustion science* 13.4, pp. 293–345 (cit. on p. 27).

Falch, Thomas L and Anne C Elster (2015). "Machine learning based auto-tuning for enhanced opencl performance portability". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, pp. 1231–1240 (cit. on p. 160).

Fang, Jianbin, Ana Lucia Varbanescu, and Henk Sips (2011). "A comprehensive performance comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. IEEE, pp. 216–225 (cit. on pp. 149, 158, 159).

Ferguson, RI and M Church (2004). "A simple universal equation for grain settling velocity". In: *Journal of sedimentary Research* 74.6, pp. 933–937 (cit. on pp. 2, 38).

Filiba, Tomer (2012). URL: https://web.archive.org/web/20160818105926/http://tomerfiliba.com/blog/Code-Generation-Context-Managers/ (cit. on p. 153).

Fornberg, Bengt (1988). "Generation of finite difference formulas on arbitrarily spaced grids". In: *Mathematics of computation* 51.184, pp. 699–706 (cit. on p. 86).

— (1998). *A practical guide to pseudospectral methods*. Vol. 1. Cambridge university press (cit. on p. 71).

Foster, Ian T and Patrick H Worley (1997). "Parallel algorithms for the spectral transform method". In: *SIAM Journal on Scientific Computing* 18.3, pp. 806–837 (cit. on p. 192).

Fredsøe, J (1992). "Mechanics of coastal sediment transport, Adv. Ser". In: *Ocean Eng* 3, p. 369 (cit. on p. 22).

Freundlich, H and AD Jones (1936). "Sedimentation Volume, Dilatancy, Thixotropic and Plastic Properties of Concentrated Suspensions." In: *The Journal of Physical Chemistry* 40.9, pp. 1217–1236 (cit. on p. 23).

Frigo, Matteo and Steven G Johnson (1998). "FFTW: An adaptive software architecture for the FFT". In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*. Vol. 3. IEEE, pp. 1381–1384 (cit. on p. 229).

— (2005). "The design and implementation of FFTW3". In: *Proceedings of the IEEE* 93.2, pp. 216–231 (cit. on pp. 120, 176, 193).

— (2012). "FFTW: Fastest Fourier transform in the west". In: *Astrophysics Source Code Library* (cit. on pp. 159, 192, 229).

Fu, Haohuan, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. (2016). "The Sunway TaihuLight supercomputer: system and applications". In: *Science China Information Sciences* 59.7, p. 072001 (cit. on p. 41).

Gabriel, Edgar, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. (2004). "Open MPI: Goals, concept, and design of a next generation MPI implementation". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, pp. 97–104 (cit. on p. 47).

Garbey, Marc and Damien Tromeur-Dervout (2002). "On some Aitken-like acceleration of the Schwarz method". In: *International journal for numerical methods in fluids* 40.12, pp. 1493–1513 (cit. on p. 223).

Gautschi, Walter (2004). *Orthogonal polynomials*. Oxford university press Oxford (cit. on p. 119).

Geller, Tom (2011). "Supercomputing's exaflop target". In: *Communications of the ACM* 54.8, pp. 16–18 (cit. on p. 43).

Gentleman, W Morven and Gordon Sande (1966). "Fast Fourier Transforms: for fun and profit". In: *Proceedings of the November 7-10, 1966, fall joint computer conference*. ACM, pp. 563–578 (cit. on p. 99).

Gibbs, Ronald J (1985). "Estuarine flocs: their size, settling velocity and density". In: *Journal of Geophysical Research: Oceans* 90.C2, pp. 3249–3251 (cit. on pp. 37, 197).

Good, Irving John (1958). "The interaction algorithm and practical Fourier analysis". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 20.2, pp. 361–372 (cit. on p. 99).

Gotoh, T, S Hatanaka, and H Miura (2012). "Spectral compact difference hybrid computation of passive scalar in isotropic turbulence". In: *Journal of Computational Physics* 231.21, pp. 7398–7414 (cit. on p. 21).

Gottlieb, David and Steven A Orszag (1977). *Numerical analysis of spectral methods: theory and applications*. Vol. 26. Siam (cit. on p. 98).

Goz, David, Luca Tornatore, Giuliano Taffoni, and Giuseppe Murante (2017). "Cosmological Simulations in Exascale Era". In: *arXiv preprint arXiv:1712.00252* (cit. on p. 56).

Grandclement, Philippe (2006). "Introduction to spectral methods". In: *Stellar fluid dynamics and numerical simulations: from the sun to neutron stars*. Ed. by M. Rieutord and B. Dubrulle. 20 pages, 15 figures. France: EAS Publication series, volume 21, p. 153 (cit. on p. 119).

Grauer-Gray, Scott, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos (2012). "Auto-tuning a high-level language targeted to GPU codes". In: *2012 Innovative Parallel Computing (InPar)*. Ieee, pp. 1–10 (cit. on p. 159).

Green, Theodore (1987). "The importance of double diffusion to the settling of suspended material". In: *Sedimentology* 34.2, pp. 319–331 (cit. on p. 36).

Guelton, Serge, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud (2015). "Pythran: Enabling static optimization of scientific python programs". In: *Computational Science & Discovery* 8.1, p. 014001 (cit. on pp. 58, 157).

Gupta, Anshul and Vipin Kumar (1993). "The scalability of FFT on parallel computers". In: *IEEE Transactions on Parallel and Distributed Systems* 4.8, pp. 922–932 (cit. on p. 192).

Hak, Mohamed Gad-el (1995). "Questions in fluid mechanics". In: *Journal of Fluids Engineering* 117.3, p. 5 (cit. on p. 12).

Hamuraru, Anca (2016). *Atomic operations for floats in OpenCL*. URL: https://streamhpc.com/blog/2016-02-09/atomic-operations-for-floats-in-opencl-improved/ (visited on 07/01/2019) (cit. on p. 168).

Harlow, Francis H and J Eddie Welch (1965). "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface". In: *The physics of fluids* 8.12, pp. 2182–2189 (cit. on p. 86).

Hart, William B (2010). "Fast library for number theory: an introduction". In: *International Congress on Mathematical Software*. Springer, pp. 88–91 (cit. on p. 230).

Härtel, Carlos, Eckart Meiburg, and Frieder Necker (2000). "Analysis and direct numerical simulation of the flow at a gravity-current head. Part 1. Flow topology and front speed

for slip and no-slip boundaries". In: *Journal of Fluid Mechanics* 418, pp. 189–212 (cit. on p. 30).

Haworth, Daniel Connell and SB Pope (1986). "A generalized Langevin model for turbulent flows". In: *The Physics of fluids* 29.2, pp. 387–405 (cit. on p. 27).

He, Qingyun, Hongli Chen, and Jingchao Feng (2015). "Acceleration of the OpenFOAM-based MHD solver using graphics processing units". In: *Fusion Engineering and Design* 101, pp. 88–93 (cit. on p. 55).

Hejazialhosseini, Babak, Diego Rossinelli, Christian Conti, and Petros Koumoutsakos (2012). "High throughput software for direct numerical simulations of compressible two-phase flows". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, p. 16 (cit. on p. 56).

Hemmert, Scott (2010). "Green hpc: From nice to necessity". In: *Computing in Science & Engineering* 12.6, p. 8 (cit. on p. 43).

Henry, Sylvain, Alexandre Denis, Denis Barthou, Marie-Christine Counilh, and Raymond Namyst (2014). "Toward OpenCL automatic multi-device support". In: *European Conference on Parallel Processing*. Springer, pp. 776–787 (cit. on p. 223).

Hérault, Alexis, Giuseppe Bilotta, and Robert A Dalrymple (2010). "Sph on gpu with cuda". In: *Journal of Hydraulic Research* 48.S1, pp. 74–79 (cit. on p. 56).

Higuchi, Tomokazu, Naoki Yoshifuji, Tomoya Sakai, Yoriyuki Kitta, Ryousei Takano, Tsutomu Ikegami, and Kenjiro Taura (2019). "ClPy: A NumPy-Compatible Library Accelerated with OpenCL". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, pp. 933–940 (cit. on p. 151).

Hillewaert, K (2012). "Direct numerical simulation of the taylor-green vortex at Re= 1600". In: *2nd international high order cfd workshop. On the WWW. http://www. as. dlr. de/hiocfd* (cit. on pp. 181, 182).

Hjulstrom, Filip (1935). "Studies of the morphological activity of rivers as illustrated by the River Fyris, Bulletin". In: *Geological Institute Upsalsa* 25, pp. 221–527 (cit. on p. 24).

Hoefler, Torsten, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood (2015). "Remote memory access programming in MPI-3". In: *ACM Transactions on Parallel Computing* 2.2, p. 9 (cit. on p. 47).

Hoffmann, Gary, Mohamad M Nasr-Azadani, and Eckart Meiburg (2015). "Sediment wave formation caused by erosional and depositional turbidity currents: A numerical investigation". In: *Procedia IUTAM* 15, pp. 26–33 (cit. on pp. 31, 55).

Holewinski, Justin, Louis-Noël Pouchet, and Ponnuswamy Sadayappan (2012). "High-performance code generation for stencil computations on GPU architectures". In: *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, pp. 311–320 (cit. on p. 90).

Hoomans, BPB, JAM Kuipers, Willem J Briels, and Willibrordus Petrus Maria van Swaaij (1996). "Discrete particle simulation of bubble and slug formation in a two-dimensional gas-fluidised bed: a hard-sphere approach". In: *Chemical Engineering Science* 51.1, pp. 99–118 (cit. on p. 27).

Houk, D and T Green (1973). "Descent rates of suspension fingers". In: *Deep Sea Research and Oceanographic Abstracts*. Vol. 20. 8. Elsevier, pp. 757–761 (cit. on pp. 1, 36).

Hoyal, David CJD, Marcus I Bursik, and Joseph F Atkinson (1999). "The influence of diffusive convection on sedimentation from buoyant plumes". In: *Marine Geology* 159.1-4, pp. 205–220 (cit. on p. 37).

Hughes, DW and MRE Proctor (1988). "Magnetic fields in the solar convection zone: magnetoconvection and magnetic buoyancy". In: *Annual Review of Fluid Mechanics* 20.1, pp. 187–223 (cit. on p. 33).

Hunt, John (2019). "Concurrency with AsyncIO". In: *Advanced Guide to Python 3 Programming*. Springer, pp. 407–417 (cit. on p. 223).

Huppert, Herbert E (1982). "The propagation of two-dimensional and axisymmetric viscous gravity currents over a rigid horizontal surface". In: *Journal of Fluid Mechanics* 121, pp. 43–58 (cit. on p. 29).

Huppert, Herbert E and Peter C Manins (1973). "Limiting conditions for salt-fingering at an interface". In: *Deep Sea Research and Oceanographic Abstracts*. Vol. 20. 4. Elsevier, pp. 315–323 (cit. on p. 36).

Huppert, Herbert E and J Stewart Turner (1981). "Double-diffusive convection". In: *Journal of Fluid Mechanics* 106, pp. 299–329 (cit. on p. 33).

Huppert, Herbert E and R Stephen J Sparks (1984). "Double-diffusive convection due to crystallization in magmas". In: *Annual Review of Earth and Planetary Sciences* 12.1, pp. 11–37 (cit. on p. 33).

Hyland, KE, S McKee, and MW Reeks (1999). "Derivation of a pdf kinetic equation for the transport of particles in turbulent flows". In: *Journal of Physics A: Mathematical and General* 32.34, p. 6169 (cit. on p. 27).

Hynninen, Antti-Pekka and Dmitry I Lyakh (2017). "cutt: A high-performance tensor transpose library for cuda compatible gpus". In: *arXiv preprint arXiv:1705.01598* (cit. on pp. 162, 223).

Ibrahimoglu, Bayram Ali (2016). "Lebesgue functions and Lebesgue constants in polynomial interpolation". In: *Journal of Inequalities and Applications* 2016.1, p. 93 (cit. on p. 119).

Ingham, Merton C (1965). "The salinity extrema of the world ocean". PhD thesis (cit. on pp. 33, 34).

Jääskeläinen, Pekka, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg (2015). "pocl: A performance-portable OpenCL implementation". In: *International Journal of Parallel Programming* 43.5, pp. 752–785 (cit. on p. 48).

Jacobi, CGJ (1859). "Untersuchungen über die Differentialgleichung der hypergeometrischen Reihe." In: *Journal für die reine und angewandte Mathematik* 56, pp. 149–165 (cit. on p. 119).

Jeannot, Emmanuel and Guillaume Mercier (2010). "Near-optimal placement of MPI processes on hierarchical NUMA architectures". In: *European Conference on Parallel Processing*. Springer, pp. 199–210 (cit. on p. 47).

Jedouaa, Meriem, C-H Bruneau, and Emmanuel Maitre (2019). "An efficient interface capturing method for a large collection of interacting bodies immersed in a fluid". In: *Journal of Computational Physics* 378, pp. 143–177 (cit. on p. 28).

Jia, Zhe, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza (2018). "Dissecting the nvidia volta gpu architecture via microbenchmarking". In: *arXiv preprint arXiv:1804.06826* (cit. on p. 174).

Jodra, Jose L, Ibai Gurrutxaga, and Javier Muguerza (2015). "Efficient 3D transpositions in graphics processing units". In: *International Journal of Parallel Programming* 43.5, pp. 876–891 (cit. on p. 91).

Johansson, Fredrik (2013). "Arb: a C library for ball arithmetic." In: *ACM Comm. Computer Algebra* 47.3/4, pp. 166–169 (cit. on p. 230).

Johnson, Steven G (2011). "Notes on FFT-based differentiation". In: *MIT Applied Mathematics* April (cit. on p. 100).

Kadoch, Benjamin, Dmitry Kolomenskiy, Philippe Angot, and Kai Schneider (2012). "A volume penalization method for incompressible flows and scalar advection–diffusion with moving obstacles". In: *Journal of Computational Physics* 231.12, pp. 4365–4383 (cit. on p. 70).

Kelley, Dan E (1990). "Fluxes through diffusive staircases: A new formulation". In: *Journal of Geophysical Research: Oceans* 95.C3, pp. 3365–3371 (cit. on p. 36).

Kelley, DE, HJS Fernando, AE Gargett, J Tanny, and E Özsoy (2003). "The diffusive regime of double-diffusive convection". In: *Progress in Oceanography* 56.3-4, pp. 461–481 (cit. on pp. 33, 35).

Kempe, Tobias and Jochen Fröhlich (2012). "An improved immersed boundary method with direct forcing for the simulation of particle laden flows". In: *Journal of Computational Physics* 231.9, pp. 3663–3684 (cit. on p. 28).

Keys, Robert (1981). "Cubic convolution interpolation for digital image processing". In: *IEEE transactions on acoustics, speech, and signal processing* 29.6, pp. 1153–1160 (cit. on p. 72).

Khajeh-Saeed, Ali and J Blair Perot (2013). "Direct numerical simulation of turbulence using GPU accelerated supercomputers". In: *Journal of Computational Physics* 235, pp. 241–257 (cit. on p. 187).

Khodkar, MA, MM Nasr-Azadani, and E Meiburg (2018). "Gravity currents propagating into two-layer stratified fluids: vorticity-based models". In: *Journal of Fluid Mechanics* 844, pp. 994–1025 (cit. on p. 31).

Khronos, OpenCL Working Group et al. (2010). *The OpenCL specification version 1.1*. URL: http://web.archive.org/web/20190826050045/https://www.khronos.org/registry/OpenCL/specs/opencl-1.1.pdf (visited on 09/01/2019) (cit. on p. 149).

— (2011). *The OpenCL Specification, version 1.2*. URL: http://web.archive.org/web/20190826144807/https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf (visited on 12/01/2019) (cit. on pp. 149, 156).

Kim, Hahn and Robert Bond (2009). "Multicore software technologies". In: *IEEE Signal Processing Magazine* 26.6, pp. 80–89 (cit. on p. 47).

Kindratenko, Volodymyr and Pedro Trancoso (2011). "Trends in high-performance computing". In: *Computing in Science & Engineering* 13.3, p. 92 (cit. on p. 44).

Kleen, Andi (2005). "A numa api for linux". In: *Novel Inc* (cit. on pp. 47, 163).

Klöckner, Andreas, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih (2012). "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3, pp. 157–174 (cit. on pp. 58, 149, 150, 157, 158, 230).

Komatsu, Hikosaburo (1960). "A characterization of real analytic functions". In: *Proceedings of the Japan Academy* 36.3, pp. 90–93 (cit. on p. 106).

Konopliv, N, L Lesshafft, and E Meiburg (2018). "The influence of shear on double-diffusive and settling-driven instabilities". In: *Journal of Fluid Mechanics* 849, pp. 902–926 (cit. on pp. 36, 224).

Konstantinidis, Elias and Yiannis Cotronis (2015). "A practical performance model for compute and memory bound GPU kernels". In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, pp. 651–658 (cit. on p. 232).

— (2017). "A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling". In: *Journal of Parallel and Distributed Computing* 107, pp. 37–56 (cit. on p. 54).

Koumoutsakos, Petros (1997). "Inviscid axisymmetrization of an elliptical vortex". In: *Journal of Computational Physics* 138.2, pp. 821–857 (cit. on p. 82).

Koumoutsakos, Petros and A Leonard (1995). "High-resolution simulations of the flow around an impulsively started cylinder using vortex methods". In: *Journal of Fluid Mechanics* 296, pp. 1–38 (cit. on p. 74).

Krishnamurti, R, Y-H Jo, and A Stocchino (2002). "Salt fingers at low Rayleigh numbers". In: *Journal of Fluid Mechanics* 452, pp. 25–37 (cit. on p. 36).

Kuerten, Johannes GM (2016). "Point-Particle DNS and LES of Particle-Laden Turbulent flow-a state-of-the-art review". In: *Flow, turbulence and combustion* 97.3, pp. 689–713 (cit. on p. 28).

Kutta, Wilhelm (1901). "Beitrag zur naherungsweisen Integration totaler Differentialgleichungen". In: *Z. Math. Phys.* 46, pp. 435–453 (cit. on p. 80).

Kwok, Yu-Kwong and Ishfaq Ahmad (1999). "Static scheduling algorithms for allocating directed task graphs to multiprocessors". In: *ACM Computing Surveys (CSUR)* 31.4, pp. 406–471 (cit. on p. 146).

Ladyzhenskaya, Olga A (1969). *The mathematical theory of viscous incompressible flow*. Vol. 2. Gordon and Breach New York (cit. on p. 7).

Lagaert, J-B, Guillaume Balarac, and G-H Cottet (2014). "Hybrid spectral-particle method for the turbulent transport of a passive scalar". In: *Journal of Computational Physics* 260, pp. 127–142 (cit. on pp. 3, 19–21, 57, 134).

Lagaert, Jean-Baptiste, Guillaume Balarac, Georges-Henri Cottet, and Patrick Bégou (2012). "Particle method: an efficient tool for direct numerical simulations of a high Schmidt number passive scalar in turbulent flow". In: (cit. on pp. 2, 21, 134).

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert (2015). "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, p. 7 (cit. on pp. 58, 231).

Lameter, Christoph et al. (2013). "NUMA (Non-Uniform Memory Access): An Overview." In: *Acm queue* 11.7, p. 40 (cit. on pp. 41, 163).

Lani, Andrea, Nadege Villedie, Khalil Bensassi, Lilla Koloszar, Martin Vymazal, Sarp M Yalim, and Marco Panesi (2013). "COOLFluiD: an open computational platform for multi-physics simulation and research". In: *21st AIAA Computational Fluid Dynamics Conference*, p. 2589 (cit. on p. 56).

Lani, Andrea, Mehmet Sarp Yalim, and Stefaan Poedts (2014). "A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids". In: *Computer Physics Communications* 185.10, pp. 2538–2557 (cit. on p. 56).

Lattner, Chris and Vikram Adve (2004). "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, p. 75 (cit. on pp. 166, 232).

Leonard, Anthony (1975). "Numerical simulation of interacting three-dimensional vortex filaments". In: *Proceedings of the Fourth International Conference on Numerical Methods in Fluid Dynamics*. Springer, pp. 245–250 (cit. on p. 19).

— (1985). "Computing three-dimensional incompressible flows with vortex elements". In: *Annual Review of Fluid Mechanics* 17.1, pp. 523–559 (cit. on p. 19).

Leray, Jean et al. (1934). "Sur le mouvement d'un liquide visqueux emplissant l'espace". In: *Acta mathematica* 63, pp. 193–248 (cit. on p. 7).

Lewis, Warren K (1922). "The evaporation of a liquid into a gas." In: *Trans. ASME.* 44, pp. 325–340 (cit. on p. 17).

Li, Ning and Sylvain Laizet (2010). "2decomp & fft-a highly scalable 2d decomposition library and fft interface". In: *Cray User Group 2010 conference*, pp. 1–13 (cit. on p. 193).

Li, Wei, Zhe Fan, Xiaoming Wei, and Arie Kaufman (2003). "GPU-based flow simulation with complex boundaries". In: *GPU Gems* 2, pp. 747–764 (cit. on p. 90).

Li, Yan, Yun-Quan Zhang, Yi-Qun Liu, Guo-Ping Long, and Hai-Peng Jia (2013). "MPFFT: An auto-tuning FFT library for OpenCL GPUs". In: *Journal of Computer Science and Technology* 28.1, pp. 90–105 (cit. on p. 159).

Linden, PF (1971). "Salt fingers in the presence of grid-generated turbulence". In: *Journal of Fluid Mechanics* 49.3, pp. 611–624 (cit. on p. 36).

List, E, Robert CY Koh, Jorgcoaut Imberger, et al. (1979). *Mixing in inland and coastal waters*. Tech. rep. (cit. on p. 204).

Liu, Jijun (2002). "Numerical solution of forward and backward problem for 2-D heat conduction equation". In: *Journal of Computational and Applied Mathematics* 145.2, pp. 459–482 (cit. on p. 105).

Logg, Anders and Garth N Wells (2010). "DOLFIN: Automated finite element computing". In: *ACM Transactions on Mathematical Software (TOMS)* 37.2, p. 20 (cit. on p. 55).

Logg, Anders, Kent-Andre Mardal, and Garth Wells (2012). *Automated solution of differential equations by the finite element method: The FEniCS book*. Vol. 84. Springer Science & Business Media (cit. on p. 55).

Lord, Rayleigh (1900). "Investigation of the character of the equilibrium of an incompressible heavy fluid of variable density". In: *Scientific papers*, pp. 200–207 (cit. on p. 201).

Luebke, David (2008). "CUDA: Scalable parallel programming for high-performance scientific computing". In: *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, pp. 836–838 (cit. on pp. 48, 71).

Luebke, David, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck (2006). "GPGPU: general-purpose computation on graphics hardware". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, p. 208 (cit. on p. 44).

Lyakh, Dmitry I (2015). "An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU". In: *Computer Physics Communications* 189, pp. 84–91 (cit. on p. 91).

Magni, Adrien and Georges-Henri Cottet (2012). "Accurate, non-oscillatory, remeshing schemes for particle methods". In: *Journal of Computational Physics* 231.1, pp. 152–172 (cit. on pp. 69, 80, 134).

Malcolm, James, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos (2012). "ArrayFire: a GPU acceleration platform". In: *Modeling and simulation for defense systems and applications VII*. Vol. 8403. International Society for Optics and Photonics, 84030A (cit. on p. 151).

Malecha, Z, Ł Mirosław, T Tomczak, Z Koza, M Matyka, W Tarnawski, Dominik Szczerba, et al. (2011). "GPU-based simulation of 3D blood flow in abdominal aorta using OpenFOAM". In: *Archives of Mechanics* 63.2, pp. 137–161 (cit. on p. 55).

Martucci, Stephen A (1994). "Symmetric convolution and the discrete sine and cosine transforms". In: *IEEE Transactions on Signal Processing* 42.5, pp. 1038–1051 (cit. on p. 110).

Mathew, J and R Vijayakumar (2011). "The Performance of Parallel Algorithms by Amdahl's Law, Gustafson's Trend". In: *International Journal of Computer Science and Information Technologies* 2.6, pp. 2796–2799 (cit. on p. 62).

Matuttis, HG, S Luding, and HJ Herrmann (2000). "Discrete element simulations of dense packings and heaps made of spherical and non-spherical particles". In: *Powder technology* 109.1-3, pp. 278–292 (cit. on p. 27).

Maxworthy, T (1999). "The dynamics of sedimenting surface gravity currents". In: *Journal of Fluid Mechanics* 392, pp. 27–44 (cit. on p. 36).

McCool, Michael D (2008). "Scalable programming models for massively multicore processors". In: *Proceedings of the IEEE* 96.5, pp. 816–831 (cit. on p. 47).

Meiburg, Eckart and Ben Kneller (2010). "Turbidity currents and their deposits". In: *Annual Review of Fluid Mechanics* 42, pp. 135–156 (cit. on p. 1).

Meiburg, Eckart, Senthil Radhakrishnan, and Mohamad Nasr-Azadani (2015). "Modeling gravity and turbidity currents: computational approaches and challenges". In: *Applied Mechanics Reviews* 67.4, p. 040802 (cit. on p. 31).

Meuer, Hans Werner and Erich Strohmaier (1993). "Die TOP 500 Supercomputer in der Welt." In: *Praxis der Informationsverarbeitung und Kommunikation* 16.4, pp. 225–230 (cit. on p. 42).

Meurer, Aaron, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. (2017). "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3, e103 (cit. on p. 230).

Meyer, Mathias (2014). "Continuous integration and its tools". In: *IEEE software* 31.3, pp. 14–16 (cit. on p. 151).

Micikevicius, Paulius (2009). "3D finite difference computation on GPUs using CUDA". In: *Proceedings of 2nd workshop on general purpose processing on graphics processing units*. ACM, pp. 79–84 (cit. on pp. 90, 187).

Miller, Christina Cruickshank (1924). "The Stokes-Einstein law for diffusion in solution". In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 106.740, pp. 724–749 (cit. on p. 197).

Milliman, John D and Katherine L Farnsworth (2013). *River discharge to the coastal ocean: a global synthesis*. Cambridge University Press (cit. on p. 1).

Mimeau, Chloé (July 2015). "Conception and implementation of a hybrid vortex penalization method for solid-fluid-porous media : application to the passive control of incompressible flows". Theses. Université Grenoble Alpes (cit. on pp. 95, 135, 203).

Mimeau, Chloé, Georges-Henri Cottet, and Iraj Mortazavi (2014). "Passive flow control around a semi-circular cylinder using porous coatings". In: (cit. on p. 134).

Mimeau, Chloe, Federico Gallizio, Georges-Henri Cottet, and Iraj Mortazavi (2015). "Vortex penalization method for bluff body flows". In: *International Journal for Numerical Methods in Fluids* 79.2, pp. 55–83 (cit. on p. 134).

Mimeau, Chloé, G-H Cottet, and Iraj Mortazavi (2016). "Direct numerical simulations of three-dimensional flows past obstacles with a vortex penalization method". In: *Computers & Fluids* 136, pp. 331–347 (cit. on pp. 62, 137).

Mimeau, Chloé, Iraj Mortazavi, and G-H Cottet (2017). "Passive control of the flow around a hemisphere using porous media". In: *European Journal of Mechanics-B/Fluids* 65, pp. 213–226 (cit. on p. 134).

Mohanan, Ashwin Vishnu, Cyrille Bonamy, Miguel Calpe Linares, and Pierre Augier (2018). "FluidSim: Modular, object-oriented python package for high-performance CFD simulations". In: *arXiv preprint arXiv:1807.01769* (cit. on p. 57).

Monaghan, JJ (1985). "Extrapolating B splines for interpolation". In: *Journal of Computational Physics* 60.2, pp. 253–262 (cit. on p. 82).

Monaghan, Joseph J, Ray AF Cas, AM Kos, and M Hallworth (1999). "Gravity currents descending a ramp in a stratified tank". In: *Journal of Fluid Mechanics* 379, pp. 39–69 (cit. on p. 30).

Moncrieff, ACM (1989). "Classification of poorly-sorted sedimentary rocks". In: *Sedimentary Geology* 65.1-2, pp. 191–194 (cit. on p. 21).

Moore, Gordon E et al. (1975). "Progress in digital integrated electronics". In: *Electron Devices Meeting*. Vol. 21, pp. 11–13 (cit. on p. 42).

Mortensen, Mikael (2018). "Shenfun: High performance spectral Galerkin computing platform." In: *J. Open Source Software* 3.31, p. 1071 (cit. on p. 57).

Mortensen, Mikael and Hans Petter Langtangen (2016). "High performance Python for direct numerical simulations of turbulent flows". In: *Computer Physics Communications* 203, pp. 53–65 (cit. on p. 57).

Mortensen, Mikael, Lisandro Dalcin, and David E Keyes (2019). "mpi4py-fft: Parallel Fast Fourier Transforms with MPI for Python". In: (cit. on p. 57).

Moureau, Vincent, Pascale Domingo, and Luc Vervisch (2011). "Design of a massively parallel CFD code for complex geometries". In: *Comptes Rendus Mécanique* 339.2-3, pp. 141–148 (cit. on p. 134).

Mouyen, Maxime, Laurent Longuevergne, Philippe Steer, Alain Crave, Jean-Michel Lemoine, Himanshu Save, and Cécile Robin (2018). "Assessing modern river sediment discharge to

the ocean using satellite gravimetry". In: *Nature communications* 9.1, p. 3384 (cit. on p. 1).

Mulder, Thierry and James PM Syvitski (1995). "Turbidity currents generated at river mouths during exceptional discharges to the world oceans". In: *The Journal of Geology* 103.3, pp. 285–299 (cit. on pp. 39, 199).

Mulder, Thierry, James PM Syvitski, Sébastien Migeon, Jean-Claude Faugeres, and Bruno Savoye (2003). "Marine hyperpycnal flows: initiation, behavior and related deposits. A review". In: *Marine and Petroleum Geology* 20.6-8, pp. 861–882 (cit. on pp. 2, 37).

Nasr-Azadani, MM and E Meiburg (2014). "Turbidity currents interacting with three-dimensional seafloor topography". In: *Journal of Fluid Mechanics* 745, pp. 409–443 (cit. on p. 31).

Navier, Claude Louis Marie Henri (1821). "Sur les Lois des Mouvement des Fluides, en Ayant Egard a L'adhesion des Molecules". In: *Ann. Chim. Paris* 19, pp. 244–260 (cit. on p. 7).

Neagoe, V-E (1990). "Chebyshev nonuniform sampling cascaded with the discrete cosine transform for optimum interpolation". In: *IEEE transactions on acoustics, speech, and signal processing* 38.10, pp. 1812–1815 (cit. on p. 120).

Necker, Frieder, Carlos Hartel, Leonhard Kleiser, and Eckart Meiburg (1999). "Direct numerical simulation of particle-driven gravity currents". In: *TSFP digital library online*. Begel House Inc. (cit. on p. 30).

Nettelmann, N, JJ Fortney, K Moore, and C Mankovich (2015). "An exploration of double diffusive convection in Jupiter as a result of hydrogen-helium phase separation". In: *Monthly Notices of the Royal Astronomical Society* 447.4, pp. 3422–3441 (cit. on p. 33).

Nickolls, John and William J Dally (2010). "The GPU computing era". In: *IEEE micro* 30.2, pp. 56–69 (cit. on p. 45).

Nishino, Ryosuke Okuta Yuya Unno Daisuke and Shohei Hido Crissman Loomis (2017). "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations". In: *31st confernce on neural information processing systems* (cit. on pp. 151, 158).

Nocentino, Anthony E and Philip J Rhodes (2010). "Optimizing memory access on GPUs using morton order indexing". In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM, p. 18 (cit. on p. 90).

Novillo, Diego (2006). "OpenMP and automatic parallelization in GCC". In: *the Proceedings of the GCC Developers Summit* (cit. on p. 47).

Nugteren, Cedric and Valeriu Codreanu (2015). "CLTune: A generic auto-tuner for OpenCL kernels". In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, pp. 195–202 (cit. on p. 160).

Nukada, Akira, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka (2008). "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA". In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, p. 5 (cit. on p. 91).

Okong'o, Nora A and Josette Bellan (2004). "Consistent large-eddy simulation of a temporal mixing layer laden with evaporating drops. Part 1. Direct numerical simulation, formulation and a priori analysis". In: *Journal of Fluid Mechanics* 499, pp. 1–47 (cit. on p. 28).

Oliphant, Travis E (2007a). "Python for scientific computing". In: *Computing in Science & Engineering* 9.3, pp. 10–20 (cit. on p. 133).

Oliphant, Travis E (2007b). "SciPy: Open source scientific tools for Python". In: *Computing in Science and Engineering* 9.1, pp. 10–20 (cit. on p. 230).

O'Rourke, Peter John (1981). *Collective drop effects on vaporizing liquid sprays*. Tech. rep. Los Alamos National Lab., NM (USA) (cit. on p. 28).

Orszag, Steven A (1969). "Numerical methods for the simulation of turbulence". In: *The Physics of Fluids* 12.12, pp. II–250 (cit. on p. 98).

Ouillon, Raphael, Nadav G Lensky, Vladimir Lyakhovsky, Ali Arnon, and Eckart Meiburg (2019). "Halite precipitation from double-diffusive salt fingers in the Dead Sea: Numerical simulations". In: *Water Resources Research* 55.5, pp. 4252–4265 (cit. on p. 37).

Owens, John D, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips (2008). "GPU computing". In: (cit. on p. 71).

Parsons, Jeffrey D and Macelo H Garcia (2000). "Enhanced sediment scavenging due to double-diffusive convection". In: *Journal of Sedimentary Research* 70.1, pp. 47–52 (cit. on p. 37).

Pekurovsky, Dmitry (2012). "P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions". In: *SIAM Journal on Scientific Computing* 34.4, pp. C192–C209 (cit. on pp. 191, 193).

Peterson, Pearu (2009). "F2PY: a tool for connecting Fortran and Python programs". In: *International Journal of Computational Science and Engineering* 4.4, pp. 296–305 (cit. on p. 231).

Piacsek, SA and J Toomre (1980). "Nonlinear evolution and structure of salt fingers". In: *Elsevier Oceanography Series*. Vol. 28. Elsevier, pp. 193–219 (cit. on p. 36).

Pippig, Michael (2013). "PFFT: An extension of FFTW to massively parallel architectures". In: *SIAM Journal on Scientific Computing* 35.3, pp. C213–C236 (cit. on p. 193).

Poisson, A and A Papaud (1983). "Diffusion coefficients of major ions in seawater". In: *Marine Chemistry* 13.4, pp. 265–280 (cit. on p. 35).

Pons, Michel and Patrick Le Quéré (2007). "Modeling natural convection with the work of pressure-forces: a thermodynamic necessity". In: *International Journal of Numerical Methods for Heat & Fluid Flow* 17.3, pp. 322–332 (cit. on p. 16).

Prager, W (1928). "Die druckverteilung an körpern in ebener potentialströmung". In: *Physik. Zeitschr* 29.865 (cit. on p. 19).

Price, James and Simon McIntosh-Smith (2015). "Oclgrind: An extensible OpenCL device simulator". In: *Proceedings of the 3rd International Workshop on OpenCL*. ACM, p. 12 (cit. on pp. 166, 232).

Prud'Homme, Christophe, Vincent Chabannes, Vincent Doyeux, Mourad Ismail, Abdoulaye Samake, and Gonçalo Pena (2012). "Feel++: A computational framework for galerkin methods and advanced numerical methods". In: *ESAIM: Proceedings*. Vol. 38. EDP Sciences, pp. 429–455 (cit. on p. 55).

Rader, Charles M (1968). "Discrete Fourier transforms when the number of data samples is prime". In: *Proceedings of the IEEE* 56.6, pp. 1107–1108 (cit. on p. 99).

Ramachandran, Prabhu (2016). "PySPH: a reproducible and high-performance framework for smoothed particle hydrodynamics". In: *Proceedings of the 15th python in science conference*, pp. 127–135 (cit. on p. 56).

Rashti, Mohammad Javad, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp (2011). "Multi-core and network aware MPI topology functions". In: *European MPI Users' Group Meeting*. Springer, pp. 50–60 (cit. on p. 47).

Raudkivi, Arved J (1998). *Loose boundary hydraulics*. CRC Press (cit. on p. 38).

Rayleigh, Lord (1916). "LIX. On convection currents in a horizontal layer of fluid, when the higher temperature is on the under side". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 32.192, pp. 529–546 (cit. on p. 17).

Recktenwald, Gerald W (2004). "Finite-difference approximations to the heat equation". In: *Mechanical Engineering* 10, pp. 1–27 (cit. on p. 105).

Reeks, M Wi (1980). "Eulerian direct interaction applied to the statistical motion of particles in a turbulent fluid". In: *Journal of Fluid Mechanics* 97.3, pp. 569–590 (cit. on p. 27).

Reynolds, Osborne (1883). "XXIX. An experimental investigation of the circumstances which determine whether the motion of water shall be direct or sinuous, and of the law of resistance in parallel channels". In: *Philosophical Transactions of the Royal society of London* 174, pp. 935–982 (cit. on p. 16).

Ronacher, Armin (2008). *Jinja2 (the Python template engine)* (cit. on p. 152).

Rosenhead, Louis (1931). "The formation of vortices from a surface of discontinuity". In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 134.823, pp. 170–192 (cit. on p. 19).

Ross, Andrew Neil (2000). "Gravity currents on slopes". PhD thesis. University of Cambridge (cit. on p. 30).

Rossinelli, Diego and Petros Koumoutsakos (2008). "Vortex methods for incompressible flow simulations on the GPU". In: *The Visual Computer* 24.7-9, pp. 699–708 (cit. on pp. 3, 19, 56).

Rossinelli, Diego, Michael Bergdorf, Georges-Henri Cottet, and Petros Koumoutsakos (2010). "GPU accelerated simulations of bluff body flows using vortex particle methods". In: *Journal of Computational Physics* 229.9, pp. 3316–3333 (cit. on pp. 3, 56).

Rossinelli, Diego, Babak Hejazialhosseini, Daniele G Spampinato, and Petros Koumoutsakos (2011). "Multicore/multi-gpu accelerated simulations of multiphase compressible flows using wavelet adapted grids". In: *SIAM Journal on Scientific Computing* 33.2, pp. 512–540 (cit. on p. 55).

Rossinelli, Diego, Babak Hejazialhosseini, Wim van Rees, Mattia Gazzola, Michael Bergdorf, and Petros Koumoutsakos (2015). "MRAG-I2D: Multi-resolution adapted grids for remeshed vortex methods on multicore architectures". In: *Journal of Computational Physics* 288, pp. 1–18 (cit. on p. 57).

Ruetsch, Greg and Paulius Micikevicius (2009). "Optimizing matrix transpose in CUDA". In: *Nvidia CUDA SDK Application Note* 18 (cit. on pp. 91, 160).

Rupp, Karl (2013). *CPU, GPU and MIC Hardware Characteristics over Time*. URL: https://web.archive.org/web/20190326080846/https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time (visited on 08/01/2019) (cit. on pp. 45, 88, 150).

Rustico, Eugenio, Giuseppe Bilotta, G Gallo, Alexis Herault, C Del Negro, and Robert Anthony Dalrymple (2012). "A journey from single-GPU to optimized multi-GPU SPH with CUDA". In: *7th SPHERIC Workshop* (cit. on p. 56).

Sakurai, Teluo, Katsunori Yoshimatsu, Naoya Okamoto, and Kai Schneider (2019). "Volume penalization for inhomogeneous Neumann boundary conditions modeling scalar flux in complicated geometry". In: *Journal of Computational Physics* 390, pp. 452–469 (cit. on p. 70).

Salihi, Mohamed Lemine Ould (1998). "Couplage de méthodes numériques en simulation directe d'écoulements incompressibles". PhD thesis. Université Joseph-Fourier-Grenoble I (cit. on p. 82).

Sanner, Michel F et al. (1999). "Python: a programming language for software integration and development". In: *J Mol Graph Model* 17.1, pp. 57–61 (cit. on p. 133).

Sbalzarini, Ivo F, Jens H Walther, Michael Bergdorf, Simone Elke Hieber, Evangelos M Kotsalis, and Petros Koumoutsakos (2006). "PPM–A highly efficient parallel particle–mesh library for the simulation of continuum systems". In: *Journal of Computational Physics* 215.2, pp. 566–588 (cit. on pp. 19, 56).

Schatzman, James C (1996). "Accuracy of the discrete Fourier transform and the fast Fourier transform". In: *SIAM Journal on Scientific Computing* 17.5, pp. 1150–1166 (cit. on p. 99).

Schmid, Rolf (1981). "Descriptive nomenclature and classification of pyroclastic deposits and fragments". In: *Geologische Rundschau* 70.2, pp. 794–799 (cit. on pp. 21, 33, 34).

Schmidt, David P and CJ Rutland (2000). "A new droplet collision algorithm". In: *Journal of Computational Physics* 164.1, pp. 62–80 (cit. on p. 28).

Schmitt, Raymond W and David L Evans (1978). "An estimate of the vertical mixing due to salt fingers based on observations in the North Atlantic Central Water". In: *Journal of Geophysical Research: Oceans* 83.C6, pp. 2913–2919 (cit. on pp. 1, 36, 37).

Schmitt Jr, Raymond W (1979). "The growth rate of super-critical salt fingers". In: *Deep Sea Research Part A. Oceanographic Research Papers* 26.1, pp. 23–40 (cit. on p. 35).

Schoenberg, Isaac Jacob (1988). "Contributions to the problem of approximation of equidistant data by analytic functions". In: *IJ Schoenberg Selected Papers*. Springer, pp. 3–57 (cit. on p. 82).

Schulte, B, N Konopliv, and E Meiburg (2016). "Clear salt water above sediment-laden fresh water: Interfacial instabilities". In: *Physical Review Fluids* 1.1, p. 012301 (cit. on p. 37).

Schwarzkopf, John D, Martin Sommerfeld, Clayton T Crowe, and Yutaka Tsuji (2011). *Multiphase flows with droplets and particles*. CRC press (cit. on pp. 26, 29).

Schwertfirm, Florian and Michael Manhart (2010). "A numerical approach for simulation of turbulent mixing and chemical reaction at high Schmidt numbers". In: *Micro and Macro Mixing*. Springer, pp. 305–324 (cit. on p. 224).

Segre, Philip N, Fang Liu, Paul Umbanhowar, and David A Weitz (2001). "An effective gravitational temperature for sedimentation". In: *Nature* 409.6820, p. 594 (cit. on pp. 2, 39).

Shen, Jie, Tao Tang, and Li-Lian Wang (2011). *Spectral methods: algorithms, analysis and applications*. Vol. 41. Springer Science & Business Media (cit. on p. 125).

Shields, Albert (1936). "Application of similarity principles and turbulence research to bed-load movement". In: *California Institute of Technology Hydraulics Laboratory* (cit. on p. 22).

Shraiman, Boris I and Eric D Siggia (2000). "Scalar turbulence". In: *Nature* 405.6787, p. 639 (cit. on p. 20).

Simon, Bertrand (2018). "Scheduling task graphs on modern computing platforms". PhD thesis (cit. on p. 146).

Simonin, O, E Deutsch, and JP Minier (1993). "Eulerian prediction of the fluid/particle correlated motion in turbulent two-phase flows". In: *Applied Scientific Research* 51.1-2, pp. 275–283 (cit. on p. 27).

Simpson, John E (1999). *Gravity currents: In the environment and the laboratory*. Cambridge university press (cit. on p. 29).

Sinclair, HD (1997). "Tectonostratigraphic model for underfilled peripheral foreland basins: An Alpine perspective". In: *Geological Society of America Bulletin* 109.3, pp. 324–346 (cit. on p. 29).

Singh, OP, D Ranjan, J Srinivasan, and KR Sreenivas (2011). "A study of basalt fingers using experiments and numerical simulations in double-diffusive systems". In: *Journal of Geography and Geology* 3.1, p. 42 (cit. on p. 33).

Singh, OP and J Srinivasan (2014). "Effect of Rayleigh numbers on the evolution of double-diffusive salt fingers". In: *Physics of Fluids* 26.6, p. 062104 (cit. on pp. 33, 36).

Smith, Gordon D and Gordon D Smith (1985). *Numerical solution of partial differential equations: finite difference methods*. Oxford university press (cit. on p. 85).

Smith, James E, W-C Hsu, and C Hsiung (1990). "Future general purpose supercomputer architectures". In: *Supercomputing'90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. IEEE, pp. 796–804 (cit. on p. 41).

Smith, Ross (2016). "Performance of MPI Codes Written in Python with NumPy and mpi4py". In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE, pp. 45–51 (cit. on p. 58).

Sommeijer, Ben P, Laurence F Shampine, and Jan G Verwer (1998). "RKC: An explicit solver for parabolic PDEs". In: *Journal of Computational and Applied Mathematics* 88.2, pp. 315–326 (cit. on p. 223).

Sommerfeld, Arnold (1908). *Ein beitrag zur hydrodynamischen erklaerung der turbulenten fluessigkeitsbewegungen* (cit. on p. 16).

Sommerfeld, Martin (2001). "Validation of a stochastic Lagrangian modelling approach for inter-particle collisions in homogeneous isotropic turbulence". In: *International Journal of Multiphase Flow* 27.10, pp. 1829–1858 (cit. on p. 28).

Spafford, Kyle, Jeremy Meredith, and Jeffrey Vetter (2010). "Maestro: data orchestration and tuning for OpenCL devices". In: *European Conference on Parallel Processing*. Springer, pp. 275–286 (cit. on p. 159).

Springel, Volker (2005). "The cosmological simulation code GADGET-2". In: *Monthly notices of the royal astronomical society* 364.4, pp. 1105–1134 (cit. on p. 56).

Springer, Paul, Aravind Sankaran, and Paolo Bientinesi (2016). "TTC: A tensor transposition compiler for multiple architectures". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, pp. 41–46 (cit. on pp. 91, 162).

Springer, Paul, Tong Su, and Paolo Bientinesi (2017). "HPTT: a high-performance tensor transposition C++ library". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, pp. 56–62 (cit. on p. 162).

Squires, Kyle D and John K Eaton (1991). "Measurements of particle dispersion obtained from direct numerical simulations of isotropic turbulence". In: *Journal of Fluid Mechanics* 226, pp. 1–35 (cit. on p. 28).

Sreenivas, KR, OP Singh, and J Srinivasan (2009a). "Effect of Rayleigh numbers on the evolution of double-diffusive salt fingers". In: *Physics of Fluid* 21, pp. 26601–15 (cit. on p. 32).

— (2009b). "On the relationship between finger width, velocity, and fluxes in thermohaline convection". In: *Physics of Fluids* 21.2, p. 026601 (cit. on p. 36).

Stallman, Richard M and Zachary Weinberg (1987). "The C preprocessor". In: *Free Software Foundation* (cit. on p. 152).

Staniforth, Andrew and Jean Côté (1991). "Semi-Lagrangian integration schemes for atmospheric models-A review". In: *Monthly weather review* 119.9, pp. 2206–2223 (cit. on p. 74).

Steinbach, Peter and Matthias Werner (2017). "gearshifft–the fft benchmark suite for heterogeneous platforms". In: *International Supercomputing Conference*. Springer, pp. 199–216 (cit. on pp. 176, 232).

Stern, Melvin E (1960). "The "salt-fountain" and thermohaline convection". In: *Tellus* 12.2, pp. 172–175 (cit. on p. 33).

— (1969). "Collective instability of salt fingers". In: *Journal of Fluid Mechanics* 35.2, pp. 209–218 (cit. on p. 33).

— (1975). *Ocean circulation physics*. Vol. 19. Academic Press (cit. on p. 35).

Stokes, George Gabriel (1851). *On the effect of the internal friction of fluids on the motion of pendulums*. Vol. 9. Pitt Press Cambridge (cit. on pp. 7, 16).

— (1880). "On the theories of the internal friction of fluids in motion, and of the equilibrium and motion of elastic solids". In: *Transactions of the Cambridge Philosophical Society* 8 (cit. on pp. 7, 12).

Stommel, Henry (1956). "An oceanographic curiosity: the perpetual salt fountain". In: *Deep-Sea Res.* 3, pp. 152–153 (cit. on p. 33).

Stone, Harold S (1973). "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations". In: *Journal of the ACM (JACM)* 20.1, pp. 27–38 (cit. on p. 105).

Stone, John E, David Gohara, and Guochun Shi (2010). "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3, p. 66 (cit. on pp. 48, 71).

Strang, Gilbert (1968). "On the construction and comparison of difference schemes". In: *SIAM journal on numerical analysis* 5.3, pp. 506–517 (cit. on p. 67).

Su, Bor-Yiing and Kurt Keutzer (2012a). "clSpMV: A cross-platform OpenCL SpMV framework on GPUs". In: *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, pp. 353–364 (cit. on p. 159).

Su, Ching-Lung, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu (2012b). "Overview and comparison of OpenCL and CUDA technology for GPGPU". In: *2012 IEEE Asia Pacific Conference on Circuits and Systems*. IEEE, pp. 448–451 (cit. on p. 149).

Su, Huayou, Nan Wu, Mei Wen, Chunyuan Zhang, and Xing Cai (2013). "On the GPU performance of 3D stencil computations implemented in OpenCL". In: *International Supercomputing Conference*. Springer, pp. 125–135 (cit. on p. 90).

Subramaniam, Shankar (2013). "Lagrangian–Eulerian methods for multiphase flows". In: *Progress in Energy and Combustion Science* 39.2-3, pp. 215–245 (cit. on p. 27).

Sugimoto, Yuki, Fumihiko Ino, and Kenichi Hagihara (2014). "Improving cache locality for GPU-based volume rendering". In: *Parallel Computing* 40.5-6, pp. 59–69 (cit. on p. 90).

Sundaram, Shivshankar and Lance R Collins (1997). "Collision statistics in an isotropic particle-laden turbulent suspension. Part 1. Direct numerical simulations". In: *Journal of Fluid Mechanics* 335, pp. 75–109 (cit. on p. 28).

Swailes, David C and Kirsty FF Darbyshire (1997). "A generalized Fokker-Planck equation for particle transport in random media". In: *Physica A: Statistical Mechanics and its Applications* 242.1-2, pp. 38–48 (cit. on p. 27).

Swarztrauber, Paul N (1982). "Vectorizing the ffts". In: *Parallel computations*. Elsevier, pp. 51–83 (cit. on p. 229).

Sweet, James, David H Richter, and Douglas Thain (2018). "GPU acceleration of Eulerian–Lagrangian particle-laden turbulent flow simulations". In: *International Journal of Multiphase Flow* 99, pp. 437–445 (cit. on p. 56).

Sych, Terry (2013). *OpenCL Device Fission for CPU Performance*. URL: https://web.archive.org/web/20160401010521/https://software.intel.com/en-us/articles/opencl-device-fission-for-cpu-performance (visited on 08/01/2019) (cit. on p. 163).

Sylvester, Zoltan (2013). *Exploring grain settling with Python*. URL: https://web.archive.org/web/20190905220840/https://hinderedsettling.com/2013/08/09/grain-settling-python (visited on 10/16/2019) (cit. on p. 38).

Taylor, John and Paul Bucens (1989). "Laboratory experiments on the structure of salt fingers". In: *Deep Sea Research Part A. Oceanographic Research Papers* 36.11, pp. 1675–1704 (cit. on p. 36).

Teisson, C (1991). "Cohesive suspended sediment transport: feasibility and limitations of numerical modeling". In: *Journal of Hydraulic Research* 29.6, pp. 755–769 (cit. on p. 22).

Terrel, Andy R (2011). "From equations to code: Automated scientific computing". In: *Computing in Science & Engineering* 13.2, p. 78 (cit. on p. 152).

Tezduyar, T, S Aliabadi, M Behr, A Johnson, V Kalro, and M Litke (1996). "Flow simulation and high performance computing". In: *Computational Mechanics* 18.6, pp. 397–412 (cit. on p. 55).

Tompson, Jonathan and Kristofer Schlachter (2012). "An introduction to the opencl programming model". In: *Person Education* 49, p. 31 (cit. on p. 149).

Traxler, A, Stephan Stellmach, Pascale Garaud, T Radko, and N Brummell (2011). "Dynamics of fingering convection. Part 1 Small-scale fluxes and large-scale instabilities". In: *Journal of fluid mechanics* 677, pp. 530–553 (cit. on p. 36).

Trench, William F (1964). "An algorithm for the inversion of finite Toeplitz matrices". In: *Journal of the Society for Industrial and Applied Mathematics* 12.3, pp. 515–522 (cit. on p. 105).

Trevett, Neil (2012). "Opencl overview". In: *Vortrag bei SIGGRAPH Asia. Zugriff am* 26, p. 2016 (cit. on p. 48).

Turner, JS (1985). "Multicomponent convection". In: *Annual Review of Fluid Mechanics* 17.1, pp. 11–44 (cit. on pp. 32, 201).

Uhlmann, Markus (2000). *The need for de-aliasing in a Chebyshev pseudo-spectral method.* Potsdam Institute for Climate Impact Research (cit. on p. 124).

— (2005). "An immersed boundary method with direct forcing for the simulation of particulate flows". In: *Journal of Computational Physics* 209.2, pp. 448–476 (cit. on p. 28).

Ungarish, Marius and Herbert E Huppert (2002). "On gravity currents propagating at the base of a stratified ambient". In: *Journal of Fluid Mechanics* 458, pp. 283–301 (cit. on p. 31).

Valdez-Balderas, Daniel, José M Domínguez, Benedict D Rogers, and Alejandro JC Crespo (2013). "Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-GPU clusters". In: *Journal of Parallel and Distributed Computing* 73.11, pp. 1483–1493 (cit. on p. 56).

Van Der Walt, Stefan, S Chris Colbert, and Gael Varoquaux (2011). "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2, p. 22 (cit. on pp. 157, 230).

Van Rees, Wim M, Anthony Leonard, DI Pullin, and Petros Koumoutsakos (2011). "A comparison of vortex and pseudo-spectral methods for the simulation of periodic vortical flows at high Reynolds numbers". In: *Journal of Computational Physics* 230.8, pp. 2794–2805 (cit. on pp. 62, 181–183).

Van Werkhoven, Ben, Jason Maassen, Frank J Seinstra, and Henri E Bal (2014). "Performance models for CPU-GPU data transfers". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, pp. 11–20 (cit. on pp. 220, 223).

Vedurada, Jyothi, Arjun Suresh, Aravind Sukumaran Rajam, Jinsung Kim, Changwan Hong, Ajay Panyala, Sriram Krishnamoorthy, V Krishna Nandivada, Rohit Kumar Srivastava, and P Sadayappan (2018). "TTLG-an efficient tensor transposition library for gpus". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 578–588 (cit. on pp. 162, 223).

Veldhuizen, Todd (1995). "Expression templates". In: *C++ Report* 7.5, pp. 26–31 (cit. on p. 151).

Verma, Anshuman, Ahmed E Helal, Konstantinos Krommydas, and Wu-Chun Feng (2016). *Accelerating workloads on fpgas via opencl: A case study with opendwarfs*. Tech. rep. Department of Computer Science, Virginia Polytechnic Institute (cit. on p. 90).

Verma, Mahendra K, Anando Chatterjee, K Sandeep Reddy, Rakesh K Yadav, Supriyo Paul, Mani Chandra, and Ravi Samtaney (2013). "Benchmarking and scaling studies of pseudospectral code Tarang for turbulence simulations". In: *Pramana* 81.4, pp. 617–629 (cit. on p. 57).

Vowinckel, B, J Withers, Paolo Luzzatto-Fegiz, and E Meiburg (2019). "Settling of cohesive sediment: particle-resolved simulations". In: *Journal of Fluid Mechanics* 858, pp. 5–44 (cit. on p. 28).

Vuduc, Richard, James W Demmel, and Katherine A Yelick (2005). "OSKI: A library of automatically tuned sparse matrix kernels". In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing, p. 521 (cit. on p. 159).

Wagner, Michael, Germán Llort, Estanislao Mercadal, Judit Giménez, and Jesús Labarta (2017). "Performance Analysis of Parallel Python Applications". In: *Procedia Computer Science* 108, pp. 2171–2179 (cit. on p. 58).

Waldrop, M Mitchell (2016). "The chips are down for Moore's law". In: *Nature News* 530.7589, p. 144 (cit. on p. 42).

Walker, David W and Jack J Dongarra (1996). "MPI: a standard message passing interface". In: *Supercomputer* 12, pp. 56–68 (cit. on p. 47).

Wang, Hao, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K Panda (2013). "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation". In: *IEEE Transactions on Parallel and Distributed Systems* 25.10, pp. 2595–2605 (cit. on p. 141).

Wang, Shuo and Yun Liang (2017). "A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, pp. 1–6 (cit. on p. 90).

Wang, Zhongde (1985). "On computing the discrete Fourier and cosine transforms". In: *IEEE transactions on acoustics, speech, and signal processing* 33.5, pp. 1341–1344 (cit. on p. 176).

Watanabe, T, T Naito, Y Sakai, K Nagata, and Y Ito (2015). "Mixing and chemical reaction at high Schmidt number near turbulent/nonturbulent interface in planar liquid jet". In: *Physics of Fluids* 27.3, p. 035114 (cit. on p. 224).

Weber, Rick, Akila Gothandaraman, Robert J Hinde, and Gregory D Peterson (2010). "Comparing hardware accelerators in scientific applications: A case study". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1, pp. 58–68 (cit. on p. 44).

Werkhoven, Ben van (2019). "Kernel Tuner: A search-optimizing GPU code auto-tuner". In: *Future Generation Computer Systems* 90, pp. 347–358 (cit. on p. 160).

Whaley, R Clinton and Jack J Dongarra (1998). "Automatically tuned linear algebra software". In: *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, pp. 38–38 (cit. on p. 159).

Willhalm, Thomas and Nicolae Popovici (2008). "Putting intel® threading building blocks to work". In: *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, pp. 3–4 (cit. on p. 56).

Williams, Samuel, Andrew Waterman, and David Patterson (2009). *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States) (cit. on pp. 43, 54, 88).

Wu, Jing and Joseph JaJa (2013). "High performance FFT based poisson solver on a CPU-GPU heterogeneous platform". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, pp. 115–125 (cit. on p. 58).

Yariv, Aridor, Fiksman Evgeny, and Singer Doron (2013). *OpenCL device partition by names extension*. URL: https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_device_partition_by_names.txt (visited on 07/01/2019) (cit. on p. 163).

Yu, Xiao, Tian-Jian Hsu, and S Balachandar (2013). "Convective instability in sedimentation: Linear stability analysis". In: *Journal of Geophysical Research: Oceans* 118.1, pp. 256–272 (cit. on p. 37).

— (2014). "Convective instability in sedimentation: 3-D numerical study". In: *Journal of Geophysical Research: Oceans* 119.11, pp. 8141–8161 (cit. on pp. 39, 40).

Zhang, Shu-Rong, Xi Xi Lu, David Laurence Higgitt, Chen-Tung Arthur Chen, Hui-Guo Sun, and Jing-Tai Han (2007). "Water chemistry of the Zhujiang (Pearl River): natural processes and anthropogenic influences". In: *Journal of Geophysical Research: Earth Surface* 112.F1 (cit. on p. 23).

Zhang, Yao, Mark Sinclair, and Andrew A Chien (2013). "Improving performance portability in OpenCL programs". In: *International Supercomputing Conference*. Springer, pp. 136–150 (cit. on p. 159).

Zhang, Yongpeng and Frank Mueller (2012). "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, pp. 155–164 (cit. on p. 90).

Zhang, Z and A Prosperetti (2005). "A second-order method for three-dimensional particle simulation". In: *Journal of Computational Physics* 210.1, pp. 292–324 (cit. on p. 28).

Zohouri, Hamid Reza, Artur Podobas, and Satoshi Matsuoka (2018). "Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, pp. 153–162 (cit. on p. 90).

Modélisation numérique et calcul haute performance de transport de sédiments

Résumé — La dynamique des écoulements sédimentaires est un sujet qui concerne de nombreuses applications en géophysiques, qui vont des questions d'ensablement des estuaires à la compréhension des bassins sédimentaires. Le sujet de cette thèse porte sur la modélisation numérique à haute résolution de ces écoulements et l'implémentation des algorithmes associés sur accélérateurs. Les écoulements sédimentaires font intervenir plusieurs phases qui interagissent, donnant lieu à plusieurs types d'instabilités comme les instabilités de Rayleigh-Taylor et de double diffusivité. Les difficultés pour la simulation numérique de ces écoulements tiennent à la complexité des interactions fluides/sédiments qui font intervenir des échelles physiques différentes. En effet, ces interactions sont difficiles à traiter du fait de la grande variabilité des paramètres de diffusion dans les deux phases et les méthodes classiques présentent certaines limites pour traiter les cas où le rapport des diffusivités, donné par le nombre de Schmidt, est trop élevé. Cette thèse étend les récents résultats obtenus sur la résolution directe de la dynamique du transport d'un scalaire passif à haut Schmidt sur architecture hybride CPU-GPU et valide cette approche sur les instabilités qui interviennent dans des écoulements sédimentaires. Ce travail revisite tout d'abord les méthodes numériques adaptées aux écoulements à haut Schmidt afin de pouvoir appliquer des stratégies d'implémentations efficaces sur accélérateurs et propose une implémentation de référence open source nommée HySoP. L'implémentation proposée permet, entre autres, de simuler des écoulements régis par les équations de Navier-Stokes incompressibles entièrement sur accélérateur ou coprocesseur grâce au standard OpenCL et tend vers des performances optimales indépendamment du matériel utilisé. La méthode numérique et son implémentation sont tout d'abord validées sur plusieurs cas tests classiques avant d'être appliquées à la dynamique des écoulements sédimentaires qui font intervenir un couplage bidirectionnel entre les scalaires transportés et les équations de Navier-Stokes. Nous montrons que l'utilisation conjointe de méthodes numériques adaptées et de leur implémentation sur accélérateur permet de décrire précisément, à coût très raisonnable, le transport sédimentaire pour des nombres de Schmidt difficilement accessibles par d'autres méthodes.

Mots clés : Transport sédimentaire, couplage bidirectionnel, méthodes particulaires, double diffusivité, HPC, GPU, HySoP

Laboratoire Jean Kuntzmann
Bâtiment IMAG - Université Grenoble Alpes
700 Avenue Centrale
Campus de Saint Martin d'Hères
38401 Domaine Universitaire de Saint-Martin-d'Hères
France

Numerical modelling and High Performance Computing for sediment flows

Abstract — The dynamic of sediment flows is a subject that covers many applications in geophysics, ranging from estuary silting issues to the comprehension of sedimentary basins. This PhD thesis deals with high resolution numerical modeling of sediment flows and implementation of the corresponding algorithms on hybrid calculators. Sedimentary flows involve multiple interacting phases, giving rise to several types of instabilities such as Rayleigh-Taylor instabilities and double diffusivity. The difficulties for the numerical simulation of these flows arise from the complex fluid/sediment interactions involving different physical scales. Indeed, these interactions are difficult to treat because of the great variability of the diffusion parameters in the two phases. When the ratio of the diffusivities, given by the Schmidt number, is too high, conventional methods show some limitations. This thesis extends the recent results obtained on the direct resolution of the transport of a passive scalar at high Schmidt number on hybrid CPU-GPU architectures and validates this approach on instabilities that occur in sediment flows. This work first reviews the numerical methods which are adapted to high Schmidt flows in order to apply effective accelerator implementation strategies and proposes an open source reference implementation named HySoP. The proposed implementation makes it possible, among other things, to simulate flows governed by the incompressible Navier-Stokes equations entirely on accelerator or coprocessor thanks to the OpenCL standard and tends towards optimal performances independently of the hardware. The numerical method and its implementation are first validated on several classical test cases and then applied to the dynamics of sediment flows which involve a two-way coupling between the transported scalars and the Navier-Stokes equations. We show that the joint use of adapted numerical methods and their implementation on accelerator makes it possible to describe accurately, at a very reasonable cost, sediment transport for Schmidt numbers difficult to reach with other methods.

Keywords: Sediment transport, dual-way coupling, particle methods, double diffusivity, HPC, GPU, HySoP

Laboratoire Jean Kuntzmann
Bâtiment IMAG - Université Grenoble Alpes
700 Avenue Centrale
Campus de Saint Martin d'Hères
38401 Domaine Universitaire de Saint-Martin-d'Hères
France