Design of the InKS programming model for the separation of algorithm and optimizations in simulation codes : application to the 6D Vlasov-Poisson system solving

Ksander Ejjaaouani

UNIVERSITÉ DE STRASBOURG

ÉCOLE DOCTORALE MSII

Maison de la Simulation

THÈSE présentée par:

Ksander EJJAAOUANI

soutenue le : **25 octobre 2019**

pour obtenir le grade de: **Docteur de l'université de Strasbourg**

Discipline/ Spécialité: Informatique

Conception du modèle de programmation INKS pour la séparation des préoccupations algorithmiques et d'optimisation dans les codes de simulation numérique ; application à la résolution du système Vlasov/Poisson 6D.

THÈSE dirigée par :

M. BASTOUL Cédric	Professeur, Université de Strasbourg
M. MEHRENBERGER Michel	Professeur, Aix-Marseille Université

RAPPORTEURS :

M. LATU Guillaume	Ingénieur-Chercheur, CEA Cadarache
M. PÉREZ Christian	Directeur de recherche, Inria Grenoble Rhône-Alpes

---

AUTRES MEMBRES DU JURY :

M. BIGOT Julien	Ingénieur-Chercheur, CEA Saclay
M. CLAUSS Philippe	Professeur, Université de Strasbourg
Mme. COULLON Hélène	Maître de conférence, IMT Atlantique
Mme. KORMANN Katharina	Chercheuse, MAX-Planck-Institut für Plasmaphysik

---

MEMBRE INVITÉ DU JURY :

M. AUMAGE Olivier	Chercheur, Inria Bordeaux – Sud Ouest

Remerciements

En premier lieu, je veux saisir cette occasion pour remercier mes encadrants, Julien BIGOT et Olivier AUMAGE. Merci à vous deux de m'avoir accueilli en stage de M2, il y a maintenant 3 ans et demi, et de m'avoir soutenu, conseillé et guidé depuis. Vos conseils, méthodologiques, techniques et scientifiques, ont été précieux pour moi, au moins autant que votre gentillesse. Sans vous, rien n'aurait été possible. Je tiens également à remercier mes directeurs de thèse, Michel MEHRENBERGER et Cédric BASTOUL. Michel, je te remercie pour tes cours en mathématiques, ta patience pour me laisser les mémoriser et ton accueil à Strasbourg (je me rappelle encore de ce restaurant que tu m'as fait essayer, où j'ai goûté pour la première fois des Spaetzles !). Cédric, merci pour ton implication dans cette thèse et tes remarques et conseils concernant les aspects scientifiques, notamment lors du comité de suivi de thèse.

Je tiens à remercier les rapporteurs, Guillaume LATU et Christian PÉREZ, d'avoir pris le temps d'évaluer ce manuscrit. Les discussions à la suite de la soumission ont réellement apporté une nouvelle perspective à ces travaux. Plus généralement, je remercie l'ensemble des membres du jury : Hélène COULLON, Philippe CLAUSS et Katharina KORMANN. Again, thank you Katharina for your collaboration in this thesis, giving me access to the 6D Vlasov-Poisson system implementation, offering me to present my work at PASC'17 (my first conference) and giving me advice on the language design and some optimizations.

Je remercie également tous les membres de la Maison de la Simulation pour vos conseils, vos discussions et votre bonne humeur. Merci à Édouard, Michel, Valérie, Pierre, Olivier, Vineet, Charles, Abel, Pierre, Jérôme, Maxime, Martial, aux Julien et tous ceux qu'il serait trop long de lister. Un grand merci à Hélène, pour tes nombreux conseils concernant l'écriture mathématique, à Thomas, pour nos discussions sur le sport et l'informatique, Mathieu, pour nos conversations relatives à la politique, l'économie et l'écologie, Pascal pour ton support concernant la simulation et la science en général, and Felix, for your help with the English grammar.

Je tiens bien évidemment à remercier Arnodu et Momo pour leur présence et leur soutien lors de nos soirées studieuses sur Discord.

Merci aux équipes IRMA et Icube de Strasbourg et LaBRI de Bordeaux pour leur accueil chaleureux. Merci aussi aux professeurs de l'université de Reims et notamment Christophe JAILLET, Pierre DELISLE, Cyril RABAT. Thanks to the Programming Environment Research Team of Kobe for your hospitality. In particular, thank you Sato Sensei for welcoming me, Nakao San and Murai San for your help with XMP, and Takahashi San and Nakashima San for your assistance and good mood.

Merci à ma famille et tous mes proches et amis, dont Julie, Cédric, Maxime, Dylan, Adeline, Axelle, Ari, Jérôme, Joséphine, Élodie, Megan et tous les autres.

Enfin, je remercie tous ceux qui méritent d'être remerciés et qui n'apparaissent pas dans ces quelques lignes.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Résumé en français

Sommaire

Les ordinateur font partie intégrante de nos sociétés modernes. Ils contrôlent une grande variété d'appareils électroniques : des objets très communs, comme les fours à micro-ondes ou les télécommandes, aux robots industriels, en passant par les ordinateurs personnels ou les smartphones. Les ordinateurs sont au cœur de nombreux domaines, aussi divers que la médecine, la physique, l'économie, mais aussi la photographie ou le cinéma. Des réactions de fusion nucléaire à l'interaction entre différentes molécules et cellules, en passant par les dépendances entre le climat et la biosphère, les simulations informatiques sont de plus en plus utilisées pour étudier des phénomènes complexes, comprendre des systèmes chaotiques ou contredire ou appuyer la validité d'hypothèses et théories scientifiques.

Dans de nombreux domaines scientifiques, il est de plus en plus commun d'identifier la simulation numérique comme étant le *troisième pilier de la science*, de pair avec la théorie et l'expérimentation. Les exemples de formidables réalisations scientifiques et techniques attribuables à l'informatique sont innombrables. Néanmoins, en 2005, le *PITAC*, une agence gouvernementale étasunienne, propose un exemple intéressant d'une percée scientifique majeure due notamment à l'informatique : le décodage du génome humain. En 1990, il est communément admis que la compréhension du génome humain permettra d'importantes avancées en médecine. Ainsi, des agences scientifiques américaines lancent le projet *"Human Genome"*. Originellement prévu pour durer plusieurs décennies, le projet fut achevé en 2001 via la collaboration des milliers de chercheurs et l'assistance cruciale de l'informatique.

Étant donné que l'informatique permet des calculs rapides sur des volumes de données qu'aucun être humain ne peut traiter au cours d'une

vie, elle offre aux scientifiques la possibilité d'obtenir des résultats en quelques heures, plutôt qu'en semaines ou années. Ce nouveau paradigme change considérablement l'éventail des études scientifiques qui peuvent être réalisées. Par exemple, les études sur le changement climatique, qui simulent des milliers d'années terrestres, ne sont utiles que si la durée pour simuler une année de climat est d'au plus quelques heures. En conclusion, l'efficacité est une condition sine qua non de telles études. C'est particulièrement vrai si l'on prend en compte le besoin de comprendre la sensibilité des prédictions aux hypothèses concernant divers impacts, comme les émissions de dioxyde de carbone ou les caractéristiques du modèle utilisé. Dans ces situations, les scientifiques doivent mener de nombreuses simulations, ce qui demande une grande puissance de calcul. Ce paradigme de recherche ne se limite pas aux études sur le climat. En effet, la plupart des domaines scientifiques s'appuient largement sur la puissance de calcul offerte par les processeurs modernes. Nous pouvons citer MODIS [Xu et al., 2014], un code de simulation implémentant les interactions géosphères-biosphères, ou LTM [Pijanowski et al., 2014], qui simule l'expansion urbaine sur les environnements naturels, qui reposent largement sur l'informatique.

Cependant, les architectures informatiques modernes, équipées de processeurs vectoriels multi-cœurs, de mémoires hiérarchiques et d'accélérateurs basés sur les processeurs graphiques ou reprogrammables, sont loin d'être simples à programmer. Bien qu'efficaces, ces outils de calcul sont mis en place au détriment des développeurs d'applications, qui doivent, lors de la conception de leurs applications, tenir compte de ce degré de parallélisme et des possibles goulots d'étranglement dus aux différentes mémoires afin d'obtenir de bonnes performances. Ainsi, avec l'augmentation de la diversité des architectures informatiques, cette situation s'empire chaque année, faisant de la portabilité des performances un problème particulièrement complexe. Atteindre un bon niveau de performances requiert des ajustements du code de façon à ce qu'il soit en adéquation avec les paramètres spécifiques d'une machine, tels que le nombre de cœurs, la taille du cache ou encore la bande passante mémoire. Des changements plus invasifs peuvent être nécessaires lors du portage d'un code d'une architecture vers une autre. Ce fut le cas lors du remplacement des

machines vectorielles par des machines équipées d'accélérateurs. De façon similaire, les architectures exaflopiques sont attendues pour 2020, mais les développeurs auront besoin de mois, voire d'années, pour mettre au point les meilleures stratégies d'optimisation pour ces nouvelles architectures.

En plus des problèmes de programmabilité des architectures actuelles, les modèles de programmation existants tendent à nécessiter une réécriture de larges parties de codes pour chaque expérimentation de nouvelles stratégies d'optimisation, rendant l'identification des meilleures techniques et paramètres d'autant plus chronophage. En outre, préserver la validité du code pendant ce processus est un objectif compliqué. Cela impose le fardeau d'une compréhension profonde de l'algorithme du programme aux spécialistes de l'optimisation. De plus, les codes de simulations hautement optimisés sont extrêmement complexes à lire et à modifier. À l'instar du problème précédent, les scientifiques du domaine doivent également être experts dans l'art de l'optimisation informatique, en plus de leur propre domaine d'expertise. Une cause profonde de cette situation vient de la ferme intrication de deux préoccupations distinctes, imposée par la plupart des modèles de programmation. D'un côté, l'algorithme de la simulation est le fruit de l'expertise de scientifiques du domaine et n'est pas lié à l'architecture cible. De l'autre, les optimisations informatiques forment un autre domaine d'expertise, largement liées à chaque architecture.

De nombreuses approches ont été proposées pour améliorer cette situation. En particulier, des outils d'optimisation et des bibliothèques simplifient l'expression des optimisations classiques. Cependant, le choix et la paramétrisation de l'optimisation restent enchevêtrés dans le code lié au domaine, tandis que les optimisations pour les architectures à venir ne sont pas toujours disponibles dans ces outils. D'autres approches proposent d'automatiser la phase d'optimisation, notamment au travers de langages et compilateurs dédiés. Bien qu'ils séparent clairement les aspects optimisations de l'algorithme, ils limitent généralement l'utilisabilité de l'approche à un domaine précis.

Une approche applicable à l'optimisation de codes de production, ciblant les architectures les plus récentes, est encore à concevoir. L'objectif de cette thèse est, premièrement, d'étendre l'ensemble des modèles de pro-

grammation pour le calcul haute performance vers un paradigme séparant l'algorithme de la simulation des optimisations spécifiques à une architecture : le modèle de programmation IN*dependent Kernel Scheduling* (INKS). Deuxièmement, cette thèse présente un cas d'application du modèle INKS à la physique des plasmas.

Ainsi, cette thèse présente notre contribution vers la simplification des codes de simulation haute performance. Ce chapitre résume, en français, l'ensemble des travaux de thèse. Les chapitres suivants, en anglais, présentent ces travaux avec plus de détails. Ce chapitre est organisé comme suit :

Travaux connexes présente les modèles de programmation visant à améliorer la productivité des développeurs de codes de simulation numérique, tout en analysant les aspects qui limitent la séparation des préoccupations. En outre, cette partie présente le modèle Polyédrique, qui est l'un des piliers de nos travaux.

Le modèle de programmation *Independent Kernel Scheduling* propose un modèle de programmation ayant pour objectif l'amélioration de la productivité des développeurs et la séparation des préoccupations dans les codes de calcul haute performance : INKS. Nous présentons également le langage INKS pour l'expression des aspects algorithmiques, ou **A**lgorithme **I**ndépendant de la **P**lateforme , $\text{INKS}_{\mathsf{PIA}}$.

Expression des optimisations dans le modèle de programmation InKS dans lequel nous présentons toutes les méthodes développées pour prendre en compte les aspects optimisations du code. En particulier, nous nous concentrons sur le langage INKS pour l'expression d'optimisation générale, ou **O**ptimisation **S**pécifique à une **P**lateforme, $\text{INKS}_{\mathsf{PSO}}$. Ce second langage permet d'exprimer un large éventail de choix d'optimisations basés sur la spécification de l'algorithme en $\text{INKS}_{\mathsf{PIA}}$.

Application et évaluation du modèle de programmation InKS évalue le modèle de programmation INKS. Dans cette évaluation, nous présentons d'abord un cas d'application réel ciblant la physique

des plasmas, le système de Vlasov-Poisson 6D, l'implémentons en utilisant le modèle InKS et évaluons notre approche sous quatre angles : son gain en productivité, sa généralité, sa simplicité et ses performances.

Les contributions présentées dans les différents chapitres de cette thèse ont été publiées dans les articles scientifiques suivantes: [Ejjaaouani et al., 2017], [Ejjaaouani et al., 2018], [Ejjaaouani et al., 2019b] and [Ejjaaouani et al., 2019a]. Finalement, nous concluons ce manuscrit et discutons de perspectives d'amélioration pour le modèle InKS.

1.1 Travaux connexes

Dans cette section, nous exposons l'état de l'art des approches utilisées dans le domaine du calcul haute performance. Tout d'abord, nous présentons quelques langages généralistes largement utilisés dans les codes de simulations, puis nous décrivons un ensemble d'outils d'optimisation utilisés en conjugaison de ces langages. Nous présentons ensuite plusieurs Langages Spécifiques à un Domaine (DSL). Par la suite, nous analysons toutes ces approches et mettons en évidence comment elles accomplissent le double objectif que nous cherchons à atteindre ; à savoir la séparation des préoccupations et l'absence de coût en matière de performance. Finalement, nous décrivons le modèle Polyédrique, qui se trouve au fondement de notre proposition.

1.1.1 Langages généralistes

Dès l'origine du traitement automatique de l'information, les scientifiques se sont appropriés l'ordinateur comme outils pour accélérer la recherche. Ainsi de nombreux langages généralistes ont été développés avec pour objectif de répondre aux besoins des scientifiques. Ces langages fournissent une généralité et un contrôle fin de tous les aspects du programme. Dans cette section, nous présentons les langages généralistes les plus utilisés de nos jours dans la communauté du calcul haute performance.

Fortran (FORmula TRANSlator) est un langage impératif compilé développé en 1954 [Backus, 1954], notamment pour le calcul scientifique. Depuis lors, *Fortran* domine cette spécialité de l'informatique, étant largement utilisée dans les codes de calcul intensif comme les programmes de prévisions météorologiques.

Le *C* est un langage impératif et compilé développé dans les années 70 [Ritchie, 1993]. Il propose un contrôle précis des accès mémoire et des constructions qui s'adaptent parfaitement aux instructions machines. Le *C* étant à la fois efficace et multi-plateforme, il est largement utilisé dans la programmation système et le calcul scientifique.

Le *C++* a été originellement conçu pour étendre les fonctionnalités du *C*, en 1983 [Stroustrup, 1997]. Désormais, c'est un langage à part entière, multi-paradigme offrant la performance du *C* et des fonctionnalités supplémentaires, notamment la programmation générique et orientée objet.

Ces langages doivent leur popularité dans le domaine du calcul numérique au contrôle des codes et de leurs optimisations tout en étant accompagnés de modèles de programmations parallèles efficaces (*OpenMP*, *MPI*). Ils fournissent des constructions semblables, adaptées aux calculs haute performance : des tableaux bas-niveau et des nids de boucles. Tout d'abord, les mesures d'un phénomène distribuées dans l'espace sont représentées par des tableaux multidimensionnels où chaque cellule stocke une valeur représentant l'intensité de cette mesure. Ensuite, des nids de boucles permettent de parcourir ces tableaux et de combiner des valeurs afin de réaliser des calculs. Ainsi, un programme de simulation typique est généralement composé de trois parties : une partie *allocation*, où les tableaux sont alloués ; une partie *initialisation*, où l'état initial de la simulation est stocké dans ces tableaux ; et enfin la partie *boucle en temps*, où s'effectuent itérativement les diagnostics et les calculs.

1.1.2 Outils d'optimisations pour le calcul haute performance

Bien que les langages généralistes pour le calcul haute performance soient efficaces et flexibles, ils demeurent verbeux et complexes à utiliser.

Pour remédier à ces problèmes, de nombreux outils proposent d'encoder un ensemble d'optimisations difficilement exprimables dans le langage d'origine. Cette stratégie offre un gain de productivité en impactant peu, voire pas, les performances. Ainsi, nous présentons quelques unes de ces approches, largement utilisées dans les codes de calcul haute performance.

Un premier type d'approche propose des extensions au langage via des directives de compilation. Cette stratégie ajoute des informations au sein d'un code existant, limitant sa modification et ajoutant des nouvelles fonctionnalités, typiquement liées au parallélisme. Par exemple, *OpenMP* est une interface de programmation supportant le parallélisme en mémoire partagée pour le *Fortran*, le *C* et le *C++* [Chandra et al., 2001]. Il propose un ensemble de directives qui contrôlent l'exécution parallèle du programme. En particulier, l'outil offre la construction `omp parallel for` permettant de distribuer les itérations d'une boucle sur les cœurs de calculs disponibles.

D'autres approches étendent les langages généralistes, typiquement le C++, avec des fonctionnalités reliées à la gestion de données et au parallélisme. Par exemple, *Kokkos* [Edwards et al., 2014] est une bibliothèque C++ basée sur la programmation générique qui offre des structures de données et de contrôle assurant une portabilité des performances. Ainsi, en modifiant des parties restreintes du code, il est possible de passer d'une implémentation séquentielle à une implémentation parallèle pour processeurs ou accélérateurs. Ensuite, à la compilation, *Kokkos* gère automatiquement la disposition mémoire et les constructions parallèles pour s'adapter à l'architecture cible choisie.

D'autres approches, basées sur les environnements d'exécution, assistent les développeurs en prenant automatiquement en compte les aspects optimisations (répartition de charge, localité des données, etc.), à l'exécution du programme. *StarPU* [Augonnet et al., 2011] est un représentant de ces approches. *StarPU* est une bibliothèque qui supporte la programmation multi-tâches sur architectures hétérogènes. L'utilisateur exprime son programme comme un ensemble de tâches et un ensemble de contraintes sous forme de dépendances de données entre ces tâches. À l'exécution, *StarPU* construit un graphe de tâches et s'occupe de la répartition de celles-ci sur les unités de calcul disponibles, tenant compte de la localité des données ou

de la répartition de charge, par exemple.

Le modèle Polyédrique (*c.f.* Section 1.1.5) propose une représentation symbolique adaptée à la transformation de code. Ainsi, de nombreux outils ont été proposés pour transformer et optimiser automatiquement un code en utilisant ce modèle. C'est par exemple le cas de *XFor* [Fassi and Clauss, 2015]. Ce dernier propose une structure de boucle n'incluant que les dépendances minimales entre chaque itération. Cette spécification peut être automatiquement manipulée pour générer des boucles *C* incluant diverses optimisations. *XFor* utilise le modèle Polyédrique pour encoder de complexes transformations de boucles.

1.1.3 Des *DSL* pour le calcul haute performance

Les langages généralistes et leurs outils pré-cités sont limités par le paradigme de programmation sous-jacent, soit le paradigme impératif. Bien qu'efficaces, d'autres approches s'appuient sur des paradigmes différents et donc, proposent un langage adapté aux constructions de ces paradigmes : un DSL (*Domain Specific Language*). À l'instar des outils d'optimisations, certains DSLs proposent, a contrario des langages généralistes, une manière plus adaptée d'exprimer un type d'optimisation. En revanche, d'autres DSLs renoncent à la généralité d'expression pour se focaliser sur des préoccupations très spécifiques, facilitant l'écriture du code comprenant ces préoccupations.

La programmation par flot de données est une alternative à la programmation par échange de messages, largement utilisée pour gérer le parallélisme. Ce paradigme propose de décrire un programme comme un graphe de données échangées entre des opérations. De nombreux DSL explorent ce paradigme pour simplifier l'exploitation des architectures hétérogènes. C'est le cas de *PaRSEC*, un environnement d'exécution pour architectures distribuées et hétérogènes, qui gère le placement des tâches, des données ainsi que leurs communications. *PaRSEC* [Bosilca et al., 2013] propose plusieurs DSL pour décrire le graphe de données. Une en particulier, PTG [Danalis et al., 2014] (pour Parameterized Task Graph), exprime ce graphe comme un ensemble d'opérations, chacune accédant à des données

produites par d'autres, à la manière d'un Makefile.

La **p**rogrammation **o**rientée **a**spect (**AOP**) est un paradigme de programmation proposant de séparer les préoccupations dans un code. Il s'agit d'ajouter des comportements à un code existant sans modifier directement ce code. À la place, il est possible d'identifier des régions de codes via un ensemble de règles, nommé *Point de coupe*, puis d'y associer une *Action*, soit un ajout ou une transformation de code. Pour illustration, la règle "toute boucle qui contient exactement une instruction" peut être un *Point de coupe*, tandis que "enregistrer le nombre d'itérations exécutées dans ces boucles" est une *Action* possible. Ce paradigme a récemment été exploré par la communauté du calcul haute performance, notamment au travers de *LARA* [Cardoso et al., 2012]. Ce DSL propose des *Points de coupe* et des *Actions* adaptés à la simulation numérique. En outre, ces spécifications peuvent être paramétrées par les caractéristiques de l'architecture cible (taille du cache, nombre de cœurs, etc.).

Une des opérations les plus fréquentes dans le calcul scientifique est le *stencil*. Il s'agit de l'approximation numérique d'une valeur en fonction de la valeur de ses voisins. Bien qu'ordinaire, l'expression optimisée d'un tel calcul est complexe. C'est pourquoi de nombreux DSL ont été proposés pour répondre à ce besoin. C'est notamment le cas de *Pochoir* [Tang et al., 2011], qui permet l'expression d'un *stencil* en laissant son optimisation à un compilateur, ou *PATUS* [Christen et al., 2011] qui offre, tout d'abord, un DSL pour l'expression de ces calculs, puis un ensemble de stratégies prédéfinies pour optimiser le calcul.

1.1.4 Discussions

Depuis les années 1950, le calcul haute performance a drastiquement changé ; d'abord basé sur des langages impératifs généraux et compliqués puis sur l'utilisation massive de bibliothèques et langages spécialisés. Cette tendance s'est accélérée ces trois dernières décennies avec une évolution plus notable encore des architectures informatiques ; de dizaines de nœuds de calculs équipés de processeurs mono-cœur à des milliers de nœuds accompagnés de plusieurs processeurs multi-cœurs, de mémoires hiérarchiques,

d'interconnexions réseaux et d'accélérateurs dédiés.

Ces bibliothèques et langages ont facilité l'écriture de codes HPC efficaces, en comparaison des langages généralistes, via deux méthodes : **1)** La proposition d'une abstraction encodant des optimisations réutilisables : simplifiant l'écriture de code efficace dans de nombreux domaines d'applications. **2)** L'utilisation d'un langage dédié et d'un compilateur réalisant automatiquement les optimisations : boostant la productivité dans un nombre de domaines plus restreints. Ainsi, *OpenMP* ou *Kokkos* sont des représentants de cette première méthode. Par exemple, *OpenMP* cache la complexité du parallélisme en mémoire partagé via des constructions facilement utilisables. Cependant, ces constructions doivent être adaptées à d'autres optimisations, comme la disposition mémoire ou l'ordre des boucles. D'autres approches, hybrides, comme *StarPU*, cachent la complexité des optimisations tout en s'appuyant sur des compilateurs accompagnés d'heuristiques efficaces. Toutes ces approches sont très générales et combinables avec les langages généralistes. Au contraire, des outils plus automatiques, comme *Pochoir*, se basent sur la seconde méthode. En restreignant le domaine d'application, les compilateurs peuvent mettre en place des stratégies d'optimisation complexes et efficaces.

En résumé, ces travaux peuvent être placés sur un axe allant d'approches très générales où l'optimisation est manuelle à des approches de plus en plus spécifiques ou l'optimisation est gérée automatiquement. Cette analyse permet de formuler les deux affirmations suivantes. Premièrement, aucune approche à la fois générale et automatique n'a été proposée dans le domaine du HPC, puisqu'une telle approche ne sera probablement pas performante. Deuxièmement, il y a une forte relation entre la généralité d'une approche et son besoin de s'appuyer sur l'utilisateur. Un objectif de cette thèse est de fournir un modèle de programmation général, se diriger vers une approche manuelle permettrait d'assurer performance et généralité.

Nous avons comparé les approches existantes suivant un seul axe : *"Manuel & Général" / "Automatique & Spécialisé"*. Cependant, d'autres perspectives sont à considérer. Par exemple, nous n'avons pas discuté des approches basées sur l'AOP. Bien que ces approches s'apparentent à des langages généralistes, de par leur généralité et leur aspect manuel, elles offrent

un support pour la séparation des préoccupations. Ainsi, il est intéressant d'étudier à quel point les travaux existants supportent la séparation des aspects tout en offrant de bonnes performances. En commençant par l'AOP, ceux-ci offrent, à première vue, une bonne séparation, via le système de *Point de coupe* et d'*Action*. Cependant, cette stratégie limite les possibilités d'optimisation. Par conséquent, ces optimisations se retrouvent alors dans l'algorithme, impactant la séparation. Ainsi, malgré la satisfaction de leurs objectives, nous pouvons noter que plus une de ces approches est spécifique à un domaine, plus elle offre une bonne séparation dans le domaine restreint qu'elle couvre, et vice-versa. Par exemple, *OpenMP* est utilisable dans de nombreux domaines et offre de bonnes performances, mais ne gère que le parallélisme d'instruction. A contrario, *Pochoir*, en étant dédié aux *stencils*, propose un langage dédié à ce type de calcul tandis que le compilateur gère toutes les optimisations : la séparation est parfaite, mais limitée aux *stencils*. Certaines approches se distinguent des précédentes dans leurs manières d'exprimer les aspects d'optimisations, comme *XFor* et *PATUS*. Tout d'abord, elle propose de décrire la partie invariante du code, c.-à-d. l'algorithme, puis, de spécifier le type d'optimisation à mettre en place. Par exemple, *PATUS* permet d'exprimer un *stencil* et s'accompagne d'un ensemble de stratégies d'optimisations prédéfinies. Ces approches, bien que séparant les préoccupations d'un code de simulation, ne s'appliquent qu'à un ensemble restreint de domaines.

À nouveau, ce second axe d'analyse souligne deux points. Premièrement, les langages spécifiques (comme *Pochoir*) offrent une bonne séparation de l'algorithme et des optimisations grâce à un langage dédié pour le premier et un compilateur pour gérer le second. Deuxièmement, les approches AOP fournissent également une bonne séparation en divisant physiquement les aspects et en proposant plusieurs langages, chacun dédié à une préoccupation. Notre objectif étant de fournir un modèle de programmation efficace et général, on ne peut s'appuyer uniquement sur la stratégie des langages spécifiques. Cependant, mélanger cette approche à celle des AOP semble être une bonne façon de procéder : proposer deux langages spécifiques, chacun dédié à un aspect, peut assurer performance, généralité et séparation.

1.1.5 Le modèle Polyédrique

Le modèle Polyédrique [Feautrier and Lengauer, 2011] est une abstraction mathématique pour l'analyse de programmes composés de nids de boucles affines. Une boucle est dite affine si ses bornes ainsi que les accès aux données faits dans son corps sont des fonctions affines des indices de boucles englobantes et de paramètres constants à l'exécution. Le modèle s'appuie sur des polyèdres pour représenter de telles boucles, les paramètres déterminant la taille du problème. Les exécutions d'une instruction sont représentées par un ensemble de points contenus dans ces polyèdres.

Dans [Feautrier, 1991], Feautrier propose le modèle Polyédrique pour représenter les programmes composés de nids de boucles. Ce modèle est calculable dans l'arithmétique de Presburger [Stansifer, 1984], un système décidable ne contenant que l'égalité, l'inégalité et l'addition. Ainsi, il est possible de concevoir des algorithmes traitant de nombreuses opérations (analyse, transformation, etc.), même sur une représentation symbolique, c.-à-d. avec paramètres.

Dans le modèle Polyédrique, un programme est représenté par un graphe de calculs, où chaque nœud représente une itération d'une instruction et chaque arête décrit une dépendance de données. Chaque tâche est associée à un tuple d'entiers unique. Ces tuples sont regroupés sous la forme d'un polyèdre, défini par une conjonction d'inégalités. Via la programmation linéaire, il est possible de manipuler de tels polyèdres. Ainsi, ce modèle est adapté à l'analyse de programmes affines ; soit des programmes où les bornes des boucles et les accès mémoires sont des fonctions affines des indices d'itérations et des paramètres constants. Un paramètre est un invariant symbolique de boucle ; généralement, un ensemble de paramètres borne la taille du problème. Cette classe de programmes est dénommée *Static Control Parts* (*SCoP*) dans la littérature scientifique.

Dans le modèle Polyédrique, une instruction est représentée par un vecteur des valeurs de tous les indices d'itérations englobants cette instruction. Toutes les instances d'une instruction sont représentées de manière compacte par un domaine d'itération, définit par un système d'inégalités où chaque formule est une équation affine des index englobants.

Ensuite, les fonctions de diffusion, ajoute des contraintes sur l'ordre d'exécution des itérations. Il s'agit de relations associant un vecteur d'instruction à un vecteur temps. Ce dernier représentant un espace multi-dimensionnel virtuel sur lequel les instructions sont exécutées dans l'ordre lexicographique [Baader and Nipkow, 1998].

Finalement, les fonctions d'accès, des relations associant un vecteur d'instructions et des cellules de tableaux accédées en lecture ou écriture par cette instruction. Les fonctions d'accès permettent de définir une notion de dépendance entre deux instructions : deux instructions sont en dépendance s'il y a une intersection entre l'ensemble des cellules lues ou écrites par l'une et écrites par l'autre.

Le modèle Polyédrique est présenté avec plus de détails, en anglais, dans la Section 3.5.

1.2 Le modèle de programmation *Independent Kernel Scheduling*

Comme mis en évidence dans la Section 1.1.4, les approches existantes n'offrent pas à la fois une vaste expressivité, de bonnes performances et la séparation des aspects algorithmiques et optimisations. Partant de ce constat, nous proposons le modèle de programmation *Independent Kernel Scheduling* (InKS). InKS sépare les aspects algorithmiques et d'optimisation dans les codes de simulation numérique dans le but d'améliorer la productivité des développeurs et la lisibilité des codes tout en facilitant le portage des applications. Il s'accompagne du langage InKS_{PIA}, pour exprimer la partie algorithmique, sans tenir compte des aspects optimisations, et du langage InKS_{PSO} pour décrire ces optimisations, en se basant sur l'algorithme. InKS, les compilateurs et des exemples sont disponibles à `https://github.com/Armassarion/InKS`.

Cette section présente le modèle de programmation InKS. Nous commençons par classifier les différents aspects des codes de simulation. En partant de cette catégorisation, nous proposons le modèle InKS ainsi que le langage d'algorithme InKS_{PIA}. Finalement, nous concluons ce chapitre

et discutons les choix que nous avons pris lors de l'élaboration du modèle InKS.

1.2.1 Classification des préoccupations

Nous définissons l'*algorithme* de la simulation comme l'ensemble des parties du code qui ne changent jamais, quel que soit l'architecture exécutant le code ; en d'autres termes, le code *indépendant de la plateforme*. Cela correspond au code relatif à un domaine, notamment composé de résolutions mathématiques. Au contraire, nous définissons les *choix d'optimisations* comme étant toute instruction qui n'est pas essentielle à la validité de l'application (c.-à-d. le résultat obtenu à partir de données d'entrées), mais plutôt responsable des bonnes performances de celle-ci : les *optimisations spécifiques à une plateforme*. Ces optimisations respectent l'algorithme. Ainsi, n'importe quel choix d'optimisations combiné à l'algorithme devrait donner les mêmes résultats pour des données d'entrée fixées.

Nous avons classifié ces préoccupations en étudiant plusieurs versions de la résolution par méthode des différences finies de l'équation de la chaleur en 3D. Nous avons identifié cinq aspects qui forment l'algorithme de la simulation : les valeurs qui existent au cours de la simulation, les calculs effectués, l'ensemble des coordonnées traversées par ces calculs, les contraintes sur leur ordre et le sous-ensemble des valeurs passées en entrée de la simulation et attendues en sortie. Nous avons aussi mis en évidence deux types de choix d'optimisations : la disposition mémoire des valeurs et l'ordonnancement des calculs.

1.2.2 Le modèle de programmation InKS

Dans la section précédente, nous avons vu comment les approches traditionnelles mixent les préoccupations algorithmiques et d'optimisation au sein d'un même code. Un modèle de programmation séparant ces préoccupations serait souhaitable pour faciliter le développement des codes de simulations, par rapport aux langages généralistes comme le C. Contrairement aux outils pour le HPC, ce modèle limiterait le coût de l'ajustement des optimisations d'un code, non pas en les encodant ou en

s'appuyant sur un compilateur, mais en se basant sur une partie invariante du code : l'algorithme. Cette stratégie offre deux avantages majeurs. D'abord, elle permet de faciliter la collaboration entre les différents chercheurs et ingénieurs. Et surtout, implémenter de nouvelles optimisations ne demande plus de réécrire l'application, mais seulement de nouveaux choix d'optimisation, augmentant la productivité des développeurs et facilitant la maintenabilité de multiples versions du code. Cet aspect est particulièrement intéressant concernant les architectures à venir, comme les architectures exaflopiques, où les meilleures stratégies d'optimisation n'ont potentiellement pas encore été mises au point. En outre, cette séparation de préoccupations doit être applicable à de nombreux types d'application sans impacter les performances.

Le modèle de programmation, illustré en Figure 4.2, propose deux langages pour séparer l'expression de l'algorithme et des optimisations. Chaque langage s'accompagne de concepts et notions adaptés aux aspects qui constituent la préoccupation à décrire. Le premier langage, InKS$_{\mathsf{PIA}}$, permet l'expression complète des préoccupations algorithmiques, indépendamment de toutes optimisations. En particulier, cela inclut les points identifiés dans la section précédente. Le modèle fournit également le langage InKS$_{\mathsf{PSO}}$ pour décrire les choix d'optimisation, en accord avec l'algorithme. Ces choix sont notamment formés par le placement mémoire et de l'ordonnancement des calculs. Toutes autres informations proviennent du code d'algorithme. Ainsi, pour une entrée en particulier, les valeurs de sortie d'un programme basé sur l'algorithme InKS$_{\mathsf{PIA}}$ seront identiques, à la précision numérique près, quel que soit la version des choix d'optimisation InKS$_{\mathsf{PSO}}$. Un programme InKS est ainsi une combinaison de l'algorithme et d'une version des choix d'optimisation, le tout formant une fonction utilisable depuis un code C++.

1.2.3 InKS$_{\mathsf{PIA}}$: exprimer l'algorithme dans le modèle InKS

Nous décrivons maintenant le langage d'algorithme, InKS$_{\mathsf{PIA}}$. À la manière de certains travaux connexes (*c.f.* Section 1.1.4), le langage

d'algorithme s'appuie sur le modèle Polyédrique. Cette approche permet au langage d'être applicable à une large gamme de programmes (*c.f.* Section 1.1.5) tout en étant statiquement analysable. Par ailleurs, il est possible d'appliquer à un tel programme des transformations, typiquement d'optimisation : c'est cette voie que nous suivrons dans l'implémentation du langage d'optimisation, InKS$_{\mathsf{PSO}}$.

InKS$_{\mathsf{PIA}}$ est un langage fonctionnel décrivant l'algorithme de la simulation tout en laissant la porte ouverte à toute future optimisation. Un code InKS$_{\mathsf{PIA}}$ est composé de trois parties. Premièrement, un ensemble de tableaux logiques dans lesquels une cellule ne peut être écrite qu'une seule fois. Ainsi, la réutilisation et le placement mémoire ne sont pas spécifiés en InKS$_{\mathsf{PIA}}$; ceux-ci pouvant être spécifiés plus tard, en InKS$_{\mathsf{PSO}}$. La deuxième partie est constituée d'*opérations* à grain très fin (de la taille de quelques opérations mathématiques, sans boucles) qui accèdent, en lecture ou en écriture, aux tableaux logiques. Ces accès forment les dépendances de données ; desquels nous pouvons tirer des dépendances minimales entre les opérations. La dernière partie déclare un ensemble de coordonnées auxquelles appliquer chaque opération, en fonction de paramètres d'entrée, typiquement liés à la taille du problème. Ainsi, l'ordre d'exécution total (ou parallèle) de ces opérations n'est pas spécifié et est laissé aux futurs choix d'optimisation. In fine, un code InKS$_{\mathsf{PIA}}$ spécifie un graphe de tâches (opérations) travaillant à grain très fin sur des tableaux logiques. Ces informations (ensemble d'instances d'opérations et leurs dépendances respectives) sont exprimables dans le modèle Polyédrique. Il est ainsi possible d'en obtenir un ordre d'exécution partiel, que tout choix d'optimisation devra respecter.

1.2.4 Conclusion et discussions

Après avoir classifié les différentes préoccupations comme liées à l'algorithme ou aux optimisations, nous avons proposé le modèle de programmation InKS. Celui-ci s'accompagne du langage d'algorithme InKS$_{\mathsf{PIA}}$ qui ne spécifie aucun choix d'optimisation. Ces derniers peuvent alors être exprimés avec InKS$_{\mathsf{PSO}}$, le langage d'optimisation du modèle.

Il est notable que, bien que le modèle InKS soit différent des approches existantes, le langage InKS_{PIA} partage de nombreuses similitudes avec les *DSL* de plusieurs approches. Citons *PIPES* [Kong et al., 2016] ou le *DSL* PTG de *PaRSEC* [Danalis et al., 2014] qui propose globalement les mêmes constructions que InKS_{PIA}. Néanmoins, la mise en place des choix d'optimisation est différente : *PaRSEC* et *PIPES* s'appuient sur un compilateur décidant automatiquement de ces choix. Bien qu'efficace dans de nombreuses situations, un tel compilateur ne prendra pas systématiquement les bonnes décisions. Ainsi, ce n'est pas la voie que nous choisissons (*c.f.* Section 1.1.4).

Plutôt que proposer un nouveau langage, InKS_{PIA}, il est concevable d'utiliser un sous-ensemble du C comme langage d'algorithme, des outils permettant de passer d'un code C à des objets du modèle Polyédrique. Cependant, un code C est intrinsèquement lié à des optimisations. Il serait alors complexe pour les spécialistes de l'optimisation de retrouver l'ordre partiel d'exécution ainsi que de déceler dans quelles conditions un emplacement mémoire peut être réutilisé. InKS_{PIA} n'est pas plus complexe que le C et présente déjà ces caractéristiques, facilitant la future optimisation.

1.3 Expression des optimisations dans le modèle de programmation InKS

Après avoir défini le langage d'algorithme, InKS_{PIA}, cette section s'applique à décrire les langages d'optimisation mis au point au cours de cette thèse. En effet, avant de définir InKS_{PSO}, un langage d'optimisation général, nous avons expérimenté plusieurs langages, chacun spécialisé dans un type d'optimisation. Cependant, toutes ces approches partagent la caractéristique principale d'InKS_{PSO} : ils s'appuient sur les informations contenues dans l'algorithme pour restreindre la partie de code à écrire à l'optimisation uniquement.

1.3.1 Une approche automatique pour le placement mémoire et l'ordre des calculs

Dans la section précédente, nous avons mis en évidence la possibilité pour un compilateur d'extraire, de l'algorithme InKS$_{\mathsf{PIA}}$, les informations concernant les dépendances de données entre opérations et l'ensemble des valeurs existantes au cours de l'exécution. Ainsi, il est possible de mettre au point un compilateur utilisant ces informations pour générer des choix d'optimisation valides, c.-à-d. un ordre d'exécution total en accord avec cet ordre partiel et un placement des données permettant la réutilisation de la mémoire.

Mettre au point une approche automatique générant des choix d'optimisation efficaces dans toutes les situations semble compliqué, comme nous l'avons mis en évidence dans la Section 1.1.4. Cependant, un tel compilateur apporte une première version des choix d'optimisation afin de s'assurer de la validité de l'algorithme InKS$_{\mathsf{PIA}}$. Puisque nous ne cherchons pas la performance, mais la vérification, nous avons mis au point un compilateur produisant un placement mémoire et un ordre séquentiel des opérations (c.-à-d. en accord avec l'algorithme). Pour cela, nous utilisons la bibliothèque *Integer Set Library* [Verdoolaege, 2010] pour encoder les informations InKS$_{\mathsf{PIA}}$ dans une représentation basée sur le modèle Polyédrique. Ces informations (ordre partiel d'exécution et ensemble des valeurs existantes) sont suffisantes pour produire un code séquentiel et un placement mémoire valides.

1.3.2 InKS$_{\mathsf{Loop}}$: un *DSL* pour la description des nids de boucles

InKS$_{\mathsf{Loop}}$ est un *DSL* pour la description de l'ordre d'exécution de boucles entourant une opération InKS$_{\mathsf{PIA}}$. Les autres choix d'optimisation sont exprimables en C++. InKS$_{\mathsf{Loop}}$ se compose de quatre parties. Premièrement, l'utilisateur définit l'opération InKS$_{\mathsf{PIA}}$ à considérer. La seconde partie définit le domaine d'itération des boucles. Ce domaine provient de l'algorithme, mais peut être restreint facilement. La troisième partie permet d'ordonner les différentes boucles. Enfin, la dernière par-

tie permet d'appliquer, à l'ordre d'itération, certaines transformations pour l'optimisation. InKS$_{\mathsf{Loop}}$ supporte notamment le *cache blocking* multidimensionnel.

Nous avons évalué les performances de InKS$_{\mathsf{Loop}}$ en implémentant certains noyaux du benchmark NAS [Bailey et al., 1991] ainsi qu'un solveur de l'équation de la chaleur en 3D via la méthode des différences finies. Cette évaluation a mis en évidence la facilité d'expression des optimisations supportées par InKS$_{\mathsf{Loop}}$, bien qu'elles soient limitées. Par ailleurs, le *DSL*, associé à notre compilateur, produit du code aussi efficace que du code écrit manuellement.

1.3.3 InKS$_{\mathsf{XMP}}$: InKS et XMP pour la gestion de la mémoire sur architecture distribuée

InKS$_{\mathsf{XMP}}$ est un langage d'optimisation du modèle InKS à base de directives permettant la distribution de données en architecture à mémoire distribuée. Le langage s'appuie sur *XcalableMP* (XMP), un langage PGAS (Partitioned Global Address Space) développé au RIKEN Center for Computational Science au Japon [Lee and Sato, 2010]. XMP simplifie la distribution et la communication de données en présentant l'espace mémoire d'une architecture distribuée comme un espace mémoire global. Ainsi, InKS$_{\mathsf{XMP}}$ reprend une partie des fonctionnalités de XMP — principalement concernant la distribution de données — en y associant les informations contenues dans l'algorithme. Cette combinaison permet de limiter l'usage de XMP aux seuls choix d'optimisation. Un compilateur source-à-source transforme ensuite le code InKS$_{\mathsf{XMP}}$ en C+XMP.

Nous avons évalué les performances de InKS$_{\mathsf{XMP}}$ et son compilateur en implémentant certains noyaux du benchmark NAS [Bailey et al., 1991] ainsi qu'un solveur de l'équation de la chaleur en 3D via la méthode des différences finies. Cette évaluation a mis en évidence l'efficacité du code produit par rapport à du code XMP écrit manuellement.

1.3.4 InKS$_{\mathsf{PSO}}$: un langage pour l'expression d'optimisations

InKS$_{\mathsf{PIA}}$ permet d'exprimer l'algorithme en laissant libres les choix concernant le placement mémoire et l'ordre des calculs. Les approches présentées précédemment permettent de décrire, dans un cadre limité, certains de ces choix. Nous décrivons maintenant InKS$_{\mathsf{PSO}}$, le langage d'optimisation du modèle InKS permettant la formulation d'optimisations variées et complexes. InKS$_{\mathsf{PSO}}$ est un langage impératif pour exprimer les choix d'optimisation d'un algorithme décrits en InKS$_{\mathsf{PIA}}$.

Un code InKS$_{\mathsf{PSO}}$ se compose de quatre parties. Premièrement, l'utilisateur alloue des tableaux multidimensionnels. Dans un deuxième temps, il définit des relations entre les cellules de ces tableaux et celles des tableaux logiques décrites en InKS. Cette correspondance entre mémoire physique et logique permet l'expression du placement des valeurs en mémoire. Ensuite, l'utilisateur peut demander la mise à jour, éventuellement parallèle, d'une région d'un tableau logique. Les tableaux physiques liés à cette région, ainsi qu'à celles qui devront être lues pour l'exécution de l'opération, seront accéder par l'opération. Finalement la réutilisation mémoire intervient lorsque l'utilisateur redéfinit les relations entre tableaux physiques et logiques. Ainsi, un programme InKS$_{\mathsf{PSO}}$ est une séquence de ces quatre parties dont l'exécution produit des tableaux physiques contenant les valeurs de sortie de l'algorithme.

1.4 Application et évaluation du modèle de programmation InKS

Dans cette section, nous présentons un cas d'application du modèle de programmation InKS : la simulation du système Vlasov-Poisson 6D. Nous nous baserons sur cette application pour évaluer notre modèle dans sa globalité.

1.4.1 Le système Vlasov-Poisson 6D et son implémentation InKS

Le système Vlasov-Poisson 6D décrit le mouvement de particules dans un plasma, ainsi que le champ électrique en résultant. Nous étudions sa résolution pour une espèce sur un maillage Cartésien périodique à 6 dimensions, représentant l'espace des phases. La principale inconnue est f, la fonction de distribution des particules dans l'espace des phases 6D. La partie Vlasov s'appuie sur un splitting de Strang, tandis que nous résolvons Poisson avec des transformations de Fourier. Cela conduit à six advections 1D – trois dans les dimensions d'espaces et trois dans les dimensions de vitesses – basées sur des interpolations de Lagrange de degré 3 ou 4 selon les dimensions. La résolution du système est implémentée dans SeLaLib [Inria, IPP, IRMA, IRMAR, LJLL, 2018] en Fortran et passe par une succession de ces trois étapes : advections en espace, résolution de Poisson via transformation de Fourier, advections en vitesse.

La première étape pour implémenter le système Vlasov-Poisson 6D en InKS consiste à en spécifier l'algorithme en $\text{InKS}_{\mathsf{PIA}}$. Nous l'avons fait en nous basant sur la version de SeLaLib. Nous avons ensuite implémenté, de manière incrémentale, les différentes optimisations mises en œuvre dans SeLaLib pour résoudre le système de manière efficace. En particulier, nous nous sommes concentrés sur les optimisations relatives à l'utilisation efficiente du cache, à la mise en place de la vectorisation, et à la parallélisation en mémoire partagée (OpenMP).

1.4.2 Évaluation

Dans cette évaluation, nous comparons l'implémentation InKS ($\text{InKS}_{\mathsf{PIA}}$ + $\text{InKS}_{\mathsf{PSO}}$) et la version de référence, venant de SeLaLib (Fortran), de la résolution du système Vlasov-Poisson 6D. Quatre versions ont été développées en InKS, chacune ajoutant une optimisation par rapport à la précédente :

1. une version naïve, peu performante et non implémentée dans SeLaLib ;

2. une version avec buffer intermédiaire pour la mise en place de la vectorisation ;

3. une version avec *blocking*, améliorant l'utilisation du cache ;

4. une version parallèle, s'appuyant sur OpenMP.

Cette comparaison se concentre sur quatre points : la productivité, la généralité, la simplicité et les performances d'InKS par rapport au Fortran. Le Tableau 1.1 présente le résultat des expériences.

Version	InKS	SeLaLib	Écart
Naïve	29.60 (±3.02%)	N/A	N/A
Buffer intermédiaire	34.07 (±2.80%)	44.26 (±0.32%)	-23.03%
Blocking	16.64 (±2.61%)	24.99 (±0.64%)	-33.43%
OpenMP (8 threads)	2.54 (±3.94%)	3.08 (±1.78%)	-17.53%

Table 1.1 – Comparaison des implémentations InKS et SeLaLib du système Vlasov-Poisson 6D. Temps (seconde) par itération des implémentations InKS et SeLaLib sur la résolution du système Vlasov-Poisson 6D (taille=32^6). Médiane et écart relatif maximal sur 10 pas de temps.

Version	InKS	SeLaLib modifiée		Écart
Buffer intermédiaire	34.07 (±2.80%)	44.82	(±2.09%)	-23.99%
Blocking	16.64 (±2.61%)	20.09	(±0.64%)	-17.19%
OpenMP (8 threads)	2.54 (±3.94%)	2.51	(±2.53%)	1.52 %

Table 1.2 – Comparaison des implémentations InKS et SeLaLib modifiée du système Vlasov-Poisson 6D. Temps (seconde) par itération des implémentations InKS et SeLaLib modifiée sur la résolution du système Vlasov-Poisson 6D (taille=32^6). Médiane et écart relatif maximal sur 10 pas de temps.

Le modèle de programmation InKS sépare les préoccupations algorithmiques et d'optimisation en fournissant deux langages distincts, améliorant la lisibilité des codes. Fixer l'algorithme facilite la coopération entre spécialistes du domaine de simulation et spécialistes en optimisation informatiques et permet de limiter la partie de code à écrire lors de l'expérimentation de nouvelles stratégies d'optimisation. Cependant, un

avantage mis en avant par notre expérience est lié à l'efficacité du code : le compilateur InKS, en ayant de nombreuses informations venant de l'algorithme, est capable de générer un code plus efficace que la référence en Fortran, notamment sous la forme de directives de vectorisation. Comme le montre le Tableau 1.1, les versions InKS$_{\mathsf{PSO}}$, et notamment la version *Blocking*, sont jusqu'à 1.5 fois plus efficaces que les versions Fortran. Cependant, en modifiant manuellement le code SeLaLib, et donc en s'appuyant sur le programmeur (Tableau 1.2), l'écart se réduit et est négligeable pour la version parallèle.

La généralité d'InKS vient largement du modèle Polyédrique sur lequel il s'appuie. Ce dernier nous permet d'offrir un modèle de programmation capable d'exprimer des codes de simulation complexes, comme le système Vlasov-Poisson 6D, mais aussi des choix d'optimisation d'efficacité comparable à du Fortran. En outre, cette généralité s'accompagne de gains en productivité, de par le support qu'InKS$_{\mathsf{PIA}}$ offre à InKS$_{\mathsf{PSO}}$. Ainsi, il est possible d'expérimenter de nouvelles optimisations sans réécrire une grande partie de code.

1.5 Conclusion et perspectives

À première vue, proposer un modèle alliant performance, productivité et lisibilité semble irréalisable. Dans le domaine du calcul haute performance, un développeur doit tirer parti de l'architecture sur laquelle l'application s'exécutera ; cela revient à utiliser habilement, entre autres, le cache, les instructions de vectorisation ou le parallélisme disponible. Cependant, cette efficacité se paye au prix d'un code difficilement lisible, notamment par des non-initiés à l'optimisation informatique. Séparer le code en de multiples fonctions, chacune dédiée à une tâche précise – calcul ou optimisation pour une architecture spécifique – produit un code à la fois performant et lisible. Cependant, cette accumulation de fonctions réduit considérablement la productivité des développeurs et la maintenabilité de l'application.

Dans cette thèse, nous proposons une nouvelle approche assurant efficacité, productivité et lisibilité dans les codes de calcul haute performance : le modèle de programmation InKS. S'accompagnant de deux lan-

gages distincts, dédiés à chacune des préoccupations, il sépare l'algorithme des optimisations. Cette scission facilite la coopération entre chercheurs et développeurs et améliore la lisibilité de l'application sans impacter négativement les performances. Nous avons évalué notre implémentation du modèle InKS sur un cas d'application de la physique des plasmas : la résolution du système Vlasov-Poisson 6D. Nous avons ainsi mis en lumière les avantages d'InKS en matière de performance. Par ailleurs, le modèle améliore également la productivité et la lisibilité du code.

Par ailleurs, notre proposition offre de nombreuses perspectives d'évolution. Premièrement, il serait intéressant d'explorer plus en profondeur les possibilités d'un compilateur intelligent ayant accès, à la fois, à l'algorithme et aux instructions d'optimisation. En effet, actuellement, nous n'avons mis en place que quelques directives de vectorisation. Néanmoins, l'algorithme contient d'autres informations intéressantes pour l'optimisation, permettant ainsi de déduire de possibles optimisations, par exemple concernant l'alignement mémoire. Ces informations pourraient alors être transcrites dans le code généré.

Le modèle InKS et notamment le langage d'optimisation peut être enrichi. Puisque le langage s'appuie sur le modèle Polyédrique, il serait judicieux d'intégrer à notre compilateur les outils d'optimisation de ce modèle, comme Pluto [Bondhugula et al., 2008] ou CHiLL [Basu et al., 2017]. Bien que possible en dehors du modèle InKS, le support d'un modèle de programmation parallèle en mémoire distribué – par exemple en s'appuyant sur un modèle PGAS – est aussi une étape importante pour la démocratisation de notre approche.

Un dernier exemple d'amélioration du modèle InKS se trouve dans les fondations mêmes du modèle : nous utilisons largement le modèle Polyédrique, notamment dans l'analyse du code. Cependant, cela contraint la classe de programmes exprimables en InKS. Il pourrait être intéressant d'essayer d'autres approches, quitte à perdre l'analyse statique offerte par le modèle Polyédrique.

Chapter 2

Introduction

In the modern human era, computers are ubiquitous. They are widely used as control systems for a wide variety of devices. This includes simple purpose devices like microwave ovens and remote controls, factory devices such as industrial robots, and also general-purpose devices like personal computers and mobile devices such as smartphones. They largely support domains as diverse as medicine, physics, mathematics, economics, as well as photography, cinema or music. From nuclear fusion reactions, to the interaction between diverse molecules and cells, to dependencies between the climate and the biosphere, computer simulations are increasingly used to study complex phenomena precisely, to better understand intricate and chaotic systems or to contradict or support the validity of hypotheses and theories.

In various scientific domains, it is more and more common to identify numerical simulation as the *third pillar of science*, a peer alongside theory and experimentation. Examples of scientific and technical achievements attributable to the computational science are uncountable, but in 2005, the *PITAC*, a US government agency, gives an interesting example of a major scientific breakthrough that relied on computational science in [President's Information Technology Advisory Committee, 2005]: the decoding of the human genome. In 1990, the Department of Energy and the National Institutes of Health, American government agencies, launch the *Human Genome Project*, as understanding better the genetic instructions for life was critical to the future of medical science. Even though it was expected to take

decades, by February 2001, the challenge was overcome by more than a thousand scientists and the crucial assistance of the computational science.

As computational science enables fast computations on volumes of data that no human could complete in a lifetime, it enables scientists to obtain results in hours, rather than weeks or years, which dramatically changes the range of studies scientists can conduct. For instance, climate change studies, which simulate thousands of Earth years, are feasible only if the time to simulate a year of climate is a few hours. In other words, efficiency is a sine qua non condition for such studies. This is especially true when we take into account the need to understand the sensitivity of climate predictions to assumptions about various impacts, like carbon dioxide emissions, or model characteristics. In these situations, scientists must conduct entire suites of climate simulations, which require prodigious amounts of computing power. More interesting, this is not limited to climate studies. Most scientific domains, from Formal science to the Humanities and Social sciences, rely on computing power offered by processors to partially deal with the curse of dimensionality, phenomena that arise when processing data of hundreds or thousands of dimensions; a problem typically faced when studying complex models. For instance, MODIS [Xu et al., 2014], which is a simulation code proposing to understand geosphere-biosphere interactions, or LTM [Pijanowski et al., 2014], which simulates urban expansion over natural areas, heavily count on computing power.

However, nowadays architectures are far from simple to program. Until 2001, computing architectures were composed of clusters of single-core CPUs, relatively simple to program. Shrinking the transistors composing a CPU enabled the increase of its frequency and the decrease of its energy consumption. All in all, the overall performance of CPUs was largely increased every two years [Schaller, 1997]. At this time, one could run a program on the newest CPU and observe drastic gains in efficiency. However, with cooling requirements and quantum effects reaching a plateau, processor manufacturers used the additional transistors, not to increase the plain performance of the CPU, but to multiply the number of CPUs on a single die. Theoretically, this parallelism increase performance, however, in practice, it comes at the expense of users who must design their applications

to fit this parallelism level.

In addition, while processors became more and more efficient and parallel, memory performance did not follow the same trend. This led to a so-called processor-memory performance gap, where the throughput of the former improved faster than the throughput and latency of the latter. Notably, the further is the memory from the processor, the longer it takes to access its content. This motivated the use of hierarchical memory, where a small and close to the processor memory caches a bigger and further memory and so on. Hence, pieces of data that would normally be read from global memory could be cheaply and quickly read from local memory on the condition that it is already there. All this strategy also came at a cost for the users: programs need to be conceived all the way down with data locality.

This situation worsens every year, as the diversity in terms of architectures of the latest supercomputers keeps growing and makes performance portability a particularly challenging problem. Achieving good performance requires code adjustments to fit a specific set of machine parameters, such as the number of cores, cache size, cache line size, number of registers, memory bandwidth, etc. Even more invasive changes can be required to move from one architecture to another as was illustrated by the switch from vector machines to modern architectures or more recently with the emergence of GPUs. Exascale platforms are expected around the 2020-2021 time frame, but developers will need months, if not years, to identify the best optimization strategies for these upcoming architectures.

In addition to programmability issues of current architectures, existing programming models tend to require a rewrite of large parts of the code for each new experiment with optimization strategies, making the identification of the best techniques and parameters all the more time-consuming. Preserving the code functional validity during this process is non-trivial. This imposes the burden of a deep understanding of the underlying domain algorithm on the optimization specialist. In addition, as the performance is one of the main aspects of simulation codes, they are challenging to read and/or modify. Indeed, highly optimized simulation codes largely come with sets of instructions related to parallelism and efficiency. Typical parallel consid-

erations include the management of the interconnected nodes constituting the architecture, through the use of approaches such as the MPI library, the parallelism exposed by the multiplication of cores in a processor, mentioned earlier, but also the parallelism offered by each core, designed as *Single Instruction, Multiple Data* computing units. Besides parallelism, simulation code developers have to take into account various architecture parameters mentioned earlier that leads to non-trivial nested loops and complex memory layouts to improve data locality, directives and low-level instructions to drive automatic optimizations done at compile-time. In addition, depending on the code, other concerns can appear such as load-balancing, *i.e.* how to schedule computations that do not last the same time of the available resources, or I/Os, *i.e.* how to efficiently write the results of a simulation. Not only are all these considerations excessively complex to read and understand but also they influence each other which make the simulation codes complicated. Hence, domain scientists have to become experts in the art of computer optimizations, in addition to their own domain of expertise.

As a matter of fact, simulation codes depend upon two highly knotty domains, namely computer engineering and domain science, and their development suffers from the incredible complexity that arises from the mix of these two. For this reason, the goal and the scope of this thesis were to improve readability, productivity, maintainability and portability of simulation codes at no performance cost, as to ease the development of such code. Hence, in this thesis, we have explored one of the possibilities to fulfill these objectives: the separation of concerns. That is to say the dissociation of domain science, focusing towards phenomena mathematical modeling, and optimization concerns, aiming efficient architecture usage. Therefore, this thesis exposes the following contribution. First, it extends available programming model for high-performance computation towards a paradigm that separates the simulation algorithm from architecture specific optimizations: the INdependent Kernel Scheduling (INKS) programming model. Then, it proposes an implementation of the INKS algorithm language, dedicated to express domain science concerns. It also proposes the description of several INKS optimization languages, aiming to specify the optimization related preoccupations, with one in particular

capable of taking into account a wide range of optimization choices. This thesis presents an application of the InKS programming model to plasma physics simulations.

2.1 Outline

The thesis presents our contribution towards a simplification of high performance numerical simulation codes and is organized into the following chapters:

Background and related work where we identify, present and analyze the methods proposed by the state-of-the-art in terms of programming models and approaches to effectively improve application developers productivity as well as how they limit this separation of concerns. In this analysis of related works, we also present the Polyhedral model, as it is at the foundation of our proposition.

The In*dependent Kernel Scheduling programming model* where we propose a programming model to improve developers productivity while overcoming the separating issues faced by other approaches: InKS. We also present the InKS **P**latform-**I**ndependent **A**lgorithm ($\text{InKS}_{\mathsf{PIA}}$) language; the InKS language to express algorithmic concerns.

Expressing optimization choices in the InKS programming model in which we expose all the developed methods to take into account the optimization concerns. In particular, we focus on the InKS **P**latform-**S**pecific **O**ptimization ($\text{InKS}_{\mathsf{PSO}}$) language. This second language focuses on expressing all kinds of optimization choices based on an algorithm specification described in $\text{InKS}_{\mathsf{PIA}}$.

Application and evaluation of the InKS programming model which evaluates both the InKS programming model proposition and implementation. In this evaluation we present a real-world application targeting a plasma physics simulation, the 6D Vlasov-Poisson system, implement it using the InKS model and evaluate our approach

through four angles: its gain in productivity, its generality, its simplicity of use and its performance.

The contributions presented in the different chapters of this thesis were published in several scientific articles: [Ejjaaouani et al., 2017], [Ejjaaouani et al., 2018], [Ejjaaouani et al., 2019b] and [Ejjaaouani et al., 2019a] Finally, we conclude this manuscript and propose numerous future works for our programming model.

Chapter 3

Background and related work

Contents

In this chapter, we expose the state-of-the-art in proposed languages and frameworks which are used to develop numeric simulation codes in the domain of the high-performance computing. In Section 3.1, we address some general-purpose languages widely used in simulation codes while Section 3.2 describes a set of tools used in conjugation of these languages. Section 3.3 describes some *Domain Specific Languages* (DSL). It follows with Section 3.4 which analyzes these different approaches and discusses how they compare to our approach and achieve our double objectives: separating algorithmic and optimization concerns while offering the best possible performance. Note that the approaches described in this Chapter are classified by what we consider to be their main user interface. For instance, *OpenMP* is classified as a directive based language even though it comes with an API and relies on a runtime. Finally, in Section 3.5, we present the Polyhedral model and its fundamental notions, as it is at the core of our works.

3.1 General-purpose languages for HPC

Computer science and scientific computing share a common past with mutual benefits. As a result, since the dawn of information technology, general-purpose languages were developed with the aim of potentially using them in scientific computing. These languages provide users with generality and fine control over the program. *Fortran*, *C*, *C++* and *Python* are the most common general-purpose languages used nowadays in the HPC community. In this section we present some of these languages.

3.1.1 *Fortran*

Fortran (FORmula TRANSlator) is an imperative compiled programming language developed especially for numeric computations and scientific computing in 1954 [Backus, 1954]. Since its first release, *Fortran* has dominated this area of programming and continue to be used in computationally intensive program such as weather prediction, finite element simulation or fluid dynamics. Indeed, *Fortran* provides flexibility and control over the

implementation of simulation codes and their optimization. Its popularity in the scientific community has made Fortran the most widely used high-performance computing language. Despite being one of the most ancient high-level programming language, it is still largely used because it was enhanced, over the years, by a wide variety of libraries and efficient compilers. Moreover, a lot of simulation codes were written in Fortran and the porting to a more modern language is judged too expensive.

3.1.2 *C* and *C++*

C is an imperative compiled general-purpose programming language developed in the early 1970s [Ritchie, 1993]. It was designed as a single interface for all the existing assembly languages, at a time when each was dedicated for a specific processor, complexifying program development. It provides low-level access to memory and constructs that map efficiently to typical machine instructions, such that it has been largely used in applications that used to be developed in assembly languages. It goes from the development of operating systems to embedded systems through scientific computing. In addition to its fine control over the code, *C* is cross-platform, and therefore can be compiled on a wide variety of computers and operating systems. Being straightforward and popular in the Computer science community, *C* has largely been used in the high performance computing community for a long time now.

Similarly, *C++* was originally designed as an extension of the *C* language in 1983 [Stroustrup, 1997]. Nowadays, it is a multi-paradigm language offering both performance, with low-level instruction, and productivity through object-oriented and generic programming features. For the same reasons as the *C* language, *C++* is widely used by the computer scientist community, and therefore, is as the root of a lot of libraries and tools for high performance computing as well as simulation codes themselves.

Although *Fortran*, *C* and *C++* were developed with different objectives by disparate communities, over time, they all converged to provide the sim-

ilar basic constructs greatly adapted to the high performance computing, namely low-level arrays and efficient nested loops. First, scientific measures distributed over a spatial grid are represented using multidimensional arrays, with each cell holding a numerical representation of the measure intensity. Then, nested loop, *i.e.* the repeated pattern of a loop within the body of another loop, are used to traverse the arrays and apply some computations. A simulation programs are roughly composed of the same three parts: the allocation part, in which all arrays are allocated; the initialization which sets all arrays initial values; and the time loop controls the progress of the simulation and contains nested loops, each applying a calculation to the arrays. In addition to computational nested loops, the time loop also contains diagnostic phases, where values are written to hard drives for their later analysis. Another major concern not adduced is the parallelization of the code. This typically influences the allocation part, where many arrays are distributed over the cluster of processors, and the time-loop, where each processor communicates values with its neighbors.

To illustrate the usage of general-purpose language in the numerical simulation, a sequential code simulating the distribution of heat over time in a solid material is presented in Listing 3.1. It is written in *C* but, as mentioned earlier, *Fortran* or *C++* version would look similar. First, the allocation part takes place on Line 8, where two arrays, `temperature_t` and `temperature_tp1`, are allocated. While the former holds temperature value at a given time-step `t`, the latter stores the one at the next time-step, `t+1`. Then, Line 13, the initial values of temperature are set. Finally, the time-loop starts on Line 22 and in particular, the heat distribution is computed in the nested loop Line 25: each `temperature_t` cell is read to update the one of `temperature_tp1`. Hence, by the end of the time-loop, `temperature_t` holds the values of the heat distribution after 100 time-steps.

3.2 Optimization tools for HPC

Although general-purpose languages for HPC are flexible and efficient, they are still complex and verbose to use in a lot of situations in scientific computing. A wide variety of tools offer to encode a set of reusable optimiza-

```c
#define SIZEX 1000
#define SIZEY 10000
#define NB_TIME 100
#define INDEX(x, y) ((x) + (y) * SIZEX)

int main(int argc, char **argv){

  /*...Allocations...*/
  size_t size = SIZEX * SIZEY;
  double* temperature_t = malloc(sizeof(*temperature_t) * size);
  double* temperature_tp1 = malloc(sizeof(*temperature_tp1) * size);

  /*...Initialization...*/
  for(int y=0; y<SIZEY; y++){
    for(int x=0; x<SIZEX; x++){
      temperature_t[INDEX(x, y)] = sin(y) + sin(x);
      temperature_tp1[INDEX(x, y)] = sin(y) + sin(x);
    }
  }

  /*...Time-loop...*/
  for(int t=0; t<NB_TIME; t++){

    /*...Computational nested loop...*/
    for(int y=1; y<SIZEY-1; y++){
      for(int x=1; x<SIZEX-1; x++){
        temperature_tp1[INDEX(x, y)] =
          0.1 * (
            temperature_t[INDEX(x-1, y)] +
            temperature_t[INDEX(x, y+1)] +
            temperature_t[INDEX(x+1, y)] +
            temperature_t[INDEX(x, y-1)]
          ) + 0.6 * temperature_t[INDEX(x, y)];
      }
    }

    swap(temperature_t, temperature_tp1);
  }
}
```

Listing 3.1 – Example of a simulation code: *C* implementation of the heat equation.

tions. Moreover, as these approaches target a specific set of optimizations, they offer dedicated constructs that enable users to express concerns that are not straight supported by general-purpose languages. These constructs provide a large speedup thanks to knowledge that users could not effectively otherwise use. In the end, the goal is to boost productivity with little to no impact on performance. Since these approaches are largely used by that the HPC community, we present some of them in this section.

3.2.1 Directives based languages

A first approach to propose tools for the HPC community comes in language extensions through compiler directives. It proposes to add information on top of an existing code, limiting its modification, to automatically manage some features, typically parallelism.

OpenMP *OpenMP* is a directives based language and an application programming interface (API) developed by a nonprofit technology consortium, *OpenMP Architecture Review Board* [Chandra et al., 2001]. It supports multi-platform shared memory multithreading programming in *Fortran*, *C* and *C++*. It provides a set of directives, functions and environment variables multi-platform that organize threads behavior at runtime.

Listing 3.2 illustrates the usage of *OpenMP* in a *C* code. In particular, it offers the `omp parallel for` construct which instructs the compiler to distribute the iterations of the following loop between available threads. For the sake of the explanation, let's consider that two threads, `T1` and `T2`, are accessible, the loop iterations Line 6 would be split between these. Hence, `T1` would run all iterations between `0` and `5000` while `T2` would execute the remaining ones. In summary, *OpenMP* eases writing shared memory parallel code with features such as the specification of independent loop iterations that can be executed in parallel.

```
1  int main(int argc, char **argv){
2    int a[100000];
3
4    //loop iterations are divided among available threads
5    #pragma omp parallel for
6    for (int i = 0; i < 100000; i++)
7      a[i] = 2 * i;
8
9    return 0;
10 }
```

Listing 3.2 – Demonstration of an *OpenMP* work-sharing construct.

XMP *XcalableMP* (*XMP* for short) is a directives based partitioned global address space (PGAS) language developed at the RIKEN Center for Computational Science in Japan [Lee and Sato, 2010]. It simplifies

the handling of distributed memory by presenting it to the developer as a single global space. *XMP* provides a minimal set of directives to manage distributed memory parallelism, among which directives to distribute an array over a set of nodes or to manage inter-nodes communications.

Listing 3.3 presents a simple *XMP* code. The code specifies data mapping of an array `A` (10 elements) among 2 nodes (5 elements per node). First, the `nodes` directive declares a node set `p` of size 2. Then, the `template` directive declares a template `t` of size 10, distributed by block among the node set `p`. The `align` directive maps array elements to template elements, and therefore transmits the distribution. After completing the data distribution phase, Line 6, *XMP* enables users to read, write and communicate the array elements as well as share work between the node in a node set. For instance, Line 8 shares the loop iteration according to the mapping of the template `t` elements to the node set `p`. Therefore, Line 10, the `A` array is written in parallel. In summary, *XMP* simplifies the writing of distributed parallelism applications, handling both data and computations distribution.

```
1  int main(){
2  #pragma xmp nodes p[2]
3  #pragma xmp template t[10]
4  #pragma xmp distribute t[block] onto p
5    int A[10];
6  #pragma xmp align A[i] with t[i]
7
8  #pragma xmp loop on t[i]
9    for(int i=0;i<10;i++){
10     A[i] = i+1;
11     printf("%d\n", A[i]);
12   }
13
14   return 0;
15 }
```

Listing 3.3 – Demonstration of *XMP* data and loop sharing directives.

3.2.2 Embedded *Domain Specific Languages*

Other approaches extend the general-purpose languages, typically C++ using the meta-programming approach, with domain specific features mostly related to data management and work-sharing parallelism.

Kokkos As supercomputer architectures become more and more varied, performance portability arises as a major challenge faced by the HPC community. *Kokkos* is a C++ template meta-programming based library which offers this performance portability [Edwards et al., 2014]. It provides an abstraction in which one expresses fine grain of parallelizable operations, gathered in parallel patterns and adaptable data structures. *Kokkos* can then map this work onto memory and threads, either on CPUs, GPUs and, most probably, in the future, FPGAs.

As demonstrated on Line 2 of the Listing 3.4, users can pass from CPU to GPU implementation and vice versa by modifying a few template parameters. Then, at compile-time, *Kokkos* automatically manages the memory layouts and the underlying parallel constructs to fit the target architecture.

```
1  //Could have chose OpenMP instead of Cuda
2  using ExecutionSpace = Cuda;
3
4  int main (int argc, char* argv[]) {
5    Kokkos::initialize (argc, argv);
6
7  /*
8    The following parallel_for would look like this
9    if we were using OpenMP instead of Kokkos:
10
11   #pragma omp parallel for
12   for (int i = 0; i < 10; ++i) {
13     printf ("Hello from i = %i\n", i);
14   }
15 */
16
17   Kokkos::parallel_for (
18   //We use the defined execution space
19     Kokkos::TeamPolicy<ExecutionSpace>(10,Kokkos::AUTO),
20     KOKKOS_LAMBDA (const int i) {
21       printf ("Hello␣from␣i␣=␣%i\n", i);
22     }
23   );
24
25   Kokkos::finalize ();
26 }
```

Listing 3.4 – Demonstration of *Kokkos* parallel for construct parameterization.

HPX *HPX* is a C++ Standard Library for concurrency and parallelism developed by the Stellar Group at the Louisiana State University [Kaiser et al., 2019]. *HPX* aims to extend the parallel notions introduced in

C++ 11 [Thoman et al., 2015] with keywords answering questions such as whether works can be done in parallel regarding thread-safety or wherever this work should be executed. It enables to write fully asynchronous code, ready for the exa-scale architecture, while providing a unified syntax and semantics for both intra and inter-node operations.

Armadillo *Armadillo* [Sanderson and Curtin, 2016] is a C++ library dedicated to linear algebra. It provides an efficient and easy-to-use interface combined with machine-dependent optimizations and inter-node support through *LAPACK* [Anderson et al., 1990]. In addition, it offers a high-level syntax derived from the Matlab one.

3.2.3 Runtimes

Runtime based approaches assist developers by taking care automatically of the optimization process. Users express its program using a dedicated API while, at runtime, the tool is responsible for the management of various optimization concerns, such as the data and tasks distribution. These approaches are especially useful in simulation code facing load-balancing issues.

StarPU *StarPU* [Augonnet et al., 2011] is a library which supports the many-tasks paradigm on hybrid architectures. In *StarPU*, the user expresses a set of tasks with data dependencies. At execution, the *StarPU* runtime translates these tasks to a directed acyclic graph (DAG) that is automatically scheduled on the available resources. Hence, *StarPU* handles parallel runtime concerns such as task dependencies, heterogeneous scheduling, optimized data transfers among computing processing units and cluster communications.

Legion Legion [Bauer et al., 2012] is a task-based programming model for heterogeneous architecture that focuses on data structures. It proposes an API to declare sets of data (regions), explicitly specify their properties (readable, writable, atomic access, etc.) and what are the tasks that use them. As the placement and movement of data have a crucial role in a sim-

ulation application performance, the *Legion* runtime automatically extracts parallelism and manages data movement in accordance with *Legion* code, simplifying code development, program performance tuning and porting of applications to new architectures.

3.2.4 Optimization tools based on the Polyhedral model

The Polyhedral model offers a precise symbolic representation particularly suited to describe nested loops which enables advanced code analysis and transformations at compile-time. It is described with more details in Section 3.5. With the emergence of this model and its possible use, many scientists proposed a set of tools with the same goal as general optimization tools: improving productivity of application developers and easing the writing of complex but efficient optimization strategies.

Pluto *Pluto* is an automatic parallelization tool based on the Polyhedral model [Bondhugula et al., 2008]. It first converts a *C* sequential code to its Polyhedral representation and then apply a set of transformation, typically to improve data locality, reduce cache misses and expose parallelism. Eventually, it generates *C* code exploiting work-sharing parallelism using *OpenMP* directives.

XFor *XFor* [Fassi and Clauss, 2015] proposes a loop structure based on the Polyhedral model. It offers programmers to express statements in a set nested loops with their relative execution order. It then provides a source-to-source compiler which takes this specification and a set of potential predefined optimizations the user wants to apply, and generates a *C* nested loops. In *XFor*, the Polyhedral model enables to encode reusable complex loop transformations.

3.2.5 Aspect-Oriented based languages

Aspect-Oriented Programming (AOP) is a programming paradigm that proposes to separate the different concerns constituting a code. It achieves

this goal by adding behavior to existing code without modifying the code itself. Instead, one identifies regions of code to modify using a set of rules, named *pointcut*, such as "any loop that contains exactly one instruction". Then the user provides an *action* to operate on the identified regions, such as "log the number of iterations that was executed". Considering a region of code, these *actions* either add code in or around the region or transform it.

AOP have been a paradigm recently explored by the HPC community to separate domain related code from architecture-specific optimization with the final goal to improve productivity.

LARA *LARA* is an Aspect-Oriented Programming approach that conveys domain knowledge and non-functional requirements to optimizer and mapping tools [Cardoso et al., 2012]. It proposes the *LARA* language to identify pointcuts as well as actions. While the *LARA* language is extensible, it already contains pointcuts and actions typically used in simulation code, such as mapping a task to a specific core or transforming a loop to improve cache use. An action can also be influenced by the targeted architecture, that is, it is possible to derive multiple versions of optimization depending on the underlying machine.

LoopsAJ In [Harbulot and R. Gurd, 2006], *Harbulot et al.* extend AspectJ, a Java implementation of the AOP paradigm, with *LoopsAJ*. It aims to add pointcut to loops as they are key places in computing intensive applications such as simulation codes. In respect to the AOP paradigm, *LoopsAJ* proposes to add codes around identified loops or to transform these loops into parallel or cache efficient ones.

3.3 *Domain Specific Languages* for HPC

General-purpose languages and tools, in the shape of libraries or languages extensions, are limited to the underlying programming paradigm, that is, the imperative paradigm. Although this showed its effectiveness, other approaches based their expressiveness on different paradigm; and

therefore proposed *Domain Specific Languages* (*DSL*) to handle this concern. These follow one of two possible paths. The first path is close to the one followed by HPC tools: their goal is to propose a more fitting way to express optimization concerns in general cases, compared to general-purpose languages. The approaches following the second path renounce to the generality. Indeed, by limiting their possibility to a very specific matter in simulation codes, such as stencil computations and inter-nodes communications, the approaches following this second path are capable of hiding the complexity of HPC systems and boost programmers productivity. Most of these tools propose a source-to-source compiler that translates the *DSL* to traditional *C/Fortran* code. Additionally, HPC tools can be complex to implement since they may rely on compiler extensions. A way to alleviate this issue is to use template meta-programming, as done in embedded *DSLs* such as *Kokkos* or *Armadillo*. A more language-independent answer is to propose a *DSL* and a source-to-source compiler.

3.3.1 Data-flow Languages

Data-flow based programming is a general approach proposed as an alternative to the standard message passing programming model used in simulation codes. This paradigm proposes to describe the program as a directed graph of the data flowing between operations. Several *DSL* based approaches explore this programming paradigm with the aim of overcoming the complexity in programmability of traditional approaches regarding the exploitation of the most recent heterogeneous architectures. Data-flow *DSLs* are accompanied by a runtime that makes the smart decisions in terms of task and data management at execution.

PaRSEC/PTG *PaRSEC* [Bosilca et al., 2013] is a task-based runtime for distributed heterogeneous architectures. It manages tasks and memory placements as well as communications. *PaRSEC* comes with several *DSLs*, each providing a specific programming model used to describe efficiently, to the runtime, tasks and their dependencies. PTG (Parameterized Task Graph) is a paradigm proposed by *Cosnard et al.* [Cosnard and Jeannot, 1999] in which user describes all data flow, that is, expresses all the tasks

that exist with the set of data they read and write. The PTG paradigm is at the root of the *PTG PaRSEC* DSL [Danalis et al., 2014].

DFL *DFL* [Fernández et al., 2014] is a data-flow *DSL* based on *OmpSs* [Duran et al., 2011]. It aims to, firstly, handle intra-node parallelism efficiently and code execution on heterogeneous computer units; and, secondly, exploit easily distributed systems. While its first goal is achieved using *OmpSs*, *DFL* proposes high-level operations to hide the complexity of inter-node communications.

3.3.2 Algorithmic Skeleton

Algorithmic skeletons are a high-level programming model focusing on parallel aspects. It leverages common programming patterns to hide the complexity of parallel and distributed applications. Algorithmic skeletons approaches, such as *SkePU* [Ernstsson et al., 2018], propose a limited set of patterns, called skeletons, such as *map*, *reduce* or *divide*. These high-level operations can then be combined to create more complex patterns. Algorithmic skeletons, by having the knowledge of each communication and synchronization implied by each pattern, offer several advantages. The two most important of them are that combination of patterns can be optimized statically by a compiler and it reduces errors compared to lower-level parallel programming models.

Lift Although algorithmic skeleton is a promising approach to alleviate distributed parallelism in HPC codes, compiling a high-level program based on patterns into an efficient low-level parallel code is challenging. *Lift* [Steuwer et al., 2017] is an algorithmic skeleton approach that stands out by the use of an innovative compiler targeting GPUs. The *Lift* compiler relies on an intermediate representation using functional patterns encoding OpenCL constructs. This functional nature simplifies the exploration of optimizations and enables *Lift* to generate efficient low-level code.

SkeTo *SkeTo* [Tanno and Iwasaki, 2009] proposes an algorithmic skeleton approach based on C++ and *MPI*. This approach differs by the use of

variable-length lists of elements that enables the solving of a wide range of problems. *SkeTo* provides patterns that dynamically and automatically change lists' length and manage load balancing. This strategy ensures good performance on distributed architectures as well as low memory footprint.

3.3.3 Stencil-specific DSLs

One of the most common operations in scientific computing is the stencil computation, that is, numerical approximations of a value localized in a geometrical space depending on its neighbors values. Being ordinary and potentially complex to express optimally in terms of performance, many approaches, especially in the shape of *DSLs*, proposed to handle stencil computations.

Pochoir *Pochoir* [Tang et al., 2011] is a stencil-specific *DSL* with which users specify a stencil, that is, the computation kernel and neighbor access patterns, as well as boundary conditions and a space-time domain on which to apply the stencil. The *Pochoir* compiler, a source-to-source compiler, then takes care of the code generation, performing numerous optimization strategies. This includes cache oblivious algorithm, to improve cache use, and intra-node parallelism based on Intel Cilk [Robison, 2013]. Using a minimal description of the stencil, *Pochoir* can provide a highly efficient code to handle stencil computation.

Listing 3.5 illustrates a stencil computation applied to a 2D domain and expressed using *Pochoir*. In particular, lines 12 and 23 respectively specify the stencil pattern and operation of the program.

Halide *Halide* is a language and compiler dedicated to image processing, capable of generating efficient parallel [Ragan-Kelley et al., 2017]. It provides heuristics for stencil applications that search for a compromise between data locality, parallelism, and redundant computations. First, one uses the *Halide* language to express its stencil operations as well as a scheduling. In a second step, the *Halide* compiler can use this specification to offer trade-offs between each critical optimization aspect.

```
1  //Value to return if elements outside the domain are accessed
2  Pochoir_Boundary_2D(heat_bv, u, t, x, y)
3    return 100 + 0.2*t;
4  Pochoir_Boundary_End
5
6  int main(int argc char** argv){
7    const int X = atoi(argv[1]);
8    const int Y = atoi(argv[2]);
9    const int T = atoi(argv[3]);
10
11   //Stencil shape definition
12   Pochoir_Shape_2D 2D_five_pt[] = {{1,0,0}, {0,0,0},{0,1,0},
13     {0,-1,0}, {0,0,-1}, {0,0,1}};
14
15   Pochoir_2D heat(2D_five_pt);
16
17   //2D Pochoir array of size X*Y declaration
18   Pochoir_Array_2D(double) u(X, Y);
19   u.Register_Boundary(heat_bv);
20   heat.Register_Array(u);
21
22   //Specification of the computation
23   Pochoir_Kernel_2D(heat_fn , t, x, y)
24     u(t+1, x, y) = u(t, x+1, y) - 2*u(t, x, y) + u(t, x-1, y) +
25            u(t, x, y+1) - 2*u(t, x, y) + u(t, x, y-1) +
26            u(t, x, y);
27   Pochoir_Kernel_End
28
29   /*...Initialization of u...*/
30
31   //Run the stencil on T time-steps
32   heat.Run(T, heat_fn);
33
34   return 0;
35 }
```

Listing 3.5 – Demonstration of the *Pochoir* stencil language.

PATUS Similarly to Pochoir, *PATUS* [Christen et al., 2011], for Parallel AutoTUned Stencils, proposes a *DSL* which aim to express a stencil computation. However, their optimization strategy greatly vary. While *Pochoir* relies on a compiler to decide automatically of the optimization, *PATUS* proposes its users to select predefined optimization strategies. These strategies can be parameterized by the user or left to the care of the PATUS compiler. This enables the tuning of code depending on the targeted architecture while minimizing the need of user intervention.

3.4 Discussions

Over the years, the domain of the high-performance computing drastically changed; from the development based on general and complex imperative languages to the widespread use of more and more specialized libraries, languages extensions and *DSLs*. This trend is accelerated by changes just as drastic of the high-performance computer architectures; from tens of nodes composed of single CPUs to thousands of nodes accompanied by multiple CPUs, each of them constituted of tens of computing cores, along with GPUs and FPGA accelerators. That is not to mention the profusion of kinds of memories and interconnections, despite playing an essential role in a program efficiency.

These languages and tools were successfully proposed with the aim of easing the development of numeric simulation codes, being such performance critical codes, compared to general-purpose languages for HPC, difficult to use in real-world simulation programs. This main goal is achieved by two general ways: **1)** Proposing a programming model to hide the complexity of writing optimization strategies by encoding them in reusable manners; hence easing the development and performance tuning of a wide variety of applications. **2)** Relying on a dedicated language and an adapted compiler to make the most efficient choices in terms of optimization; and therefore, boost even more the gain of productivity but in more restricted kinds of applications. This last strategy often draws on a programming model that either targets a specific type of application or enables users to give more information to the underlying compiler. In the set of approaches we have presented, directives-based languages, such as *OpenMP* and *XMP* and embedded *DSLs*, including *Kokkos* and *Armadillo*, fit in this first category. As illustrated in Listing 3.2, *OpenMP* hides the complexity of shared memory parallelism by offering adapted constructs. However, these parallel patterns heavily depend on parts of the code surrounding it, typically, memory layouts and loops. Similarly, *XMP* and *Kokkos*, presented respectively in Listings 3.3 and 3.4, uses the same strategy, but on different optimization concerns. Several approaches, such as *DFL*, *StarPU* or *HPX*, follow the same trend: these are general approaches that hide the complexity of par-

ticular optimization concerns through languages or libraries, even though they rely on compilers or runtimes for smart optimization choices. All are general approaches, usable in conjugation with a general-purpose language. On the contrary, automated tools such as *Pochoir*, *Pluto* or *Lift* lie in the second categories. For instance, as shown in Listing 3.5, *Pochoir* proposes a language and a compiler well suited to stencil operation expression. As it restricts its support to this domain, the compiler comes with complex and efficient optimization techniques.

In summary, what is common to all of these existing works is that they lie on a spectrum from very general approaches, where the optimization process is manual to more and more domains specific ones where the optimization process can be automated. On the one hand, the more general-purpose approaches support a large range of optimizations and application domains, but incur high implementation costs and low separation of concerns and portability. On the other hand, the more automated approaches reduce implementation costs and offer good separation of concerns and portability but restrain the range of supported domains and optimizations. This continuum is illustrated by Figure 3.1.

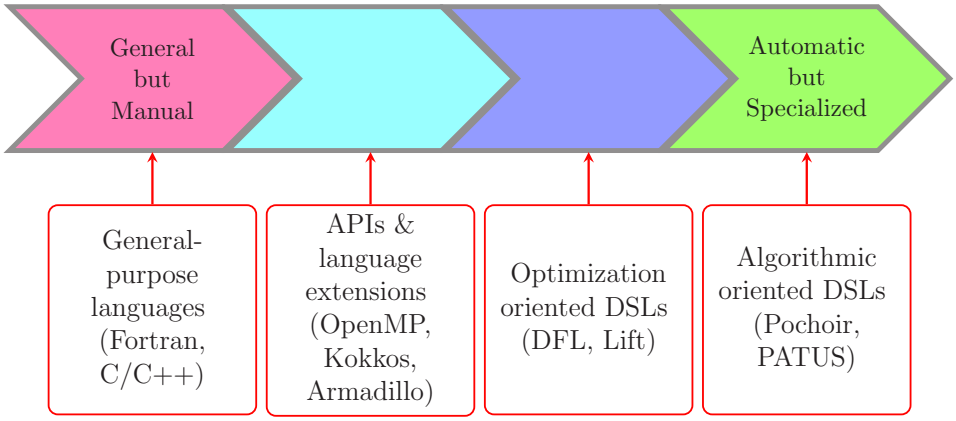


Figure 3.1 – Classification of the state-of-the-art approaches on a *"General & Manual"/"Automatic & Specialized"* axis.

This analysis and presentation have highlighted two major remarks. Firstly, no automatic, general and efficient approach have been proposed by the HPC community to specify simulation codes: all of them are either general but manual or automatized but specialized. An automatized

and general method would very likely be not efficient, and therefore, not considered for simulation. Secondly, there is a strong connection between the generality of an approach and the need for users to express various concerns manually. As our objective is to provide a programming model general enough to express a wide variety of numerical simulation programs, heading towards manual approaches seems like an efficient method. This ensures both performance and generality.

However, for now, we have compared existing works following a single axis: *"Manual & General" / "Automatic & Specialized"*. However, as varied these are, different perspectives can be considered. For instance, note that we did not discuss Aspect-Oriented approaches, with *Lara* or *LoopsAJ*. These approaches are very close to general-purpose languages: completely manual but general approaches. However, they offer a separation between domain science and optimization concerns. Typically, AOP approaches are often liked because they improve productivity, readability and maintainability of codes. Therefore, as these qualities are interesting, another appealing aspect to analyze would be how well the concerns, between domain science and computer optimizations, are separated by the tools interface while offering good performance. Starting with existing AOP works in the HPC domain, as mentioned earlier, they offer a good separation, at first sight, but the *pointcuts* and *actions* they propose tend to limit the possibility of optimization, typically the modification of the memory layout. As a result, some of these non-supported optimizations may fall in the algorithm part, limiting the separation of concerns. Nevertheless, in general, all approaches offer a good level of efficiency, but the more specific an approach is, the more it offers a separation of concerns for this specific domain, and vice versa. For instance, *OpenMP* is applicable to many domains but only takes care of instruction parallelism. Hence, other optimization concerns such as data distribution or memory layouts are left to the users and mixed with domain science. Still, used with care, *OpenMP* does not impede performance in comparison to general-purpose approaches. At the other end of the spectrum, *Pochoir* focuses on stencil computations and provides a language dedicated to this concern while all optimization aspects are handled by a compiler. Although it considers only a restricted domain, this strategy

offers a good separation of concerns at the same time as ensuring that the generated code is efficient.

In terms of separation of concerns, three approaches we have previously presented stand out in their way to take into account the optimization part: *XFor*, *PATUS* or *Halide*. In each tool, firstly, one describes the invariant parts of the code, that is to say, the algorithm. While *XFor* considers statements partial ordering, *PATUS* and *Halide* focus on stencil expressions. Secondly, the approach does not rely upon an efficient compiler. Instead, it offers its users to describe the optimization part, that is, the architecture-specific parts of the code. *XFor* proposes various automatic affine transformation of loops, whereas *PATUS* provides a language designed for stencil optimization, based on predefined but configurable strategies. As for *Halide*, one can specify scheduling strategies that will guide a compiler during the optimization process. *XFor* limits its expression to loops, whereas *PATUS* and *Halide* are designed to handle stencil programs. Although this kind of approach effectively separates the simulation programs preoccupations, it does so for a limited scope, making its applicability for complete real-world application unlikely.

This second axis of analysis has stressed two meaningful facts. Firstly, *DSLs* offer a good separation of algorithmic and optimization concerns thanks to languages dedicated to the former and compilers managing the latter. Secondly, despite AOP approaches depend upon general-purpose languages, that offer no separation, they provide this separation by spiting physically the concerns while providing users with constructs enabling interactions between each part, in the form of *pointcuts* and *actions*. As our objective is to provide a programming model both efficient and general, it cannot rely on the *DSLs* strategy; however, mixing it with the AOP one appears to be an adequate scheme. To put it another way, this consists in proposing two languages, either adapted to algorithmic or optimization concerns, as it is done in *DSLs* strategy for the algorithmic parts, and providing the language towards optimization concerns with instructions to interact with the other language, as it is done AOP approaches. This construction ensures performance, generality and separation of concerns.

In summary, this presentation and discussion of the existing works pro-

posed by the HPC community to alleviate the difficulty to program simulation code have brought to light two decisive facets that will direct this thesis. Firstly, it revealed the lack of a programming model providing both generality and efficiency while offering a good separation of concerns. Secondly, it highlighted how to drive the construction of a programming model fulfilling these objectives. Indeed, such approach shall satisfy these two crucial prerequisites:

1. it shall rely on user intervention to ensure both performance and generality;

2. it shall provide two languages, each adapted to a specific concern, to offer the separation of concerns while not impeding the performance and generality.

In the next chapter, we will present the InKS programming model which adheres to these requirements and achieves these objectives; that is, being general enough to express a wide variety of program while offering an optimization language enabling users to obtain the best possible performance in most situations.

3.5 The Polyhedral model

In this thesis work, we intensively use the Polyhedral model framework to analyze and transform code. Thus, as it is at the core of our works, we now present this model.

The Polyhedral model [Feautrier and Lengauer, 2011] (earlier known as the polytope model [Lengauer, 1993, Feautrier, 1996]) is a mathematical abstraction to analyze programs composed of nested loops. It is named after its foundation: the model relies on polyhedra in arbitrary (but finite) multidimensional spaces. This section provides a basic overview of the Polyhedral model and its objects.

3.5.1 Static Control Parts

In [Feautrier, 1991], Feautrier proposes the Polyhedral model to represent a class of programs made up of nested and successive `for` loops and `if` conditionals only. In these programs, memory is accessed through arrays that never alias each other. With such assumptions, the model fits in the Presburger arithmetic [Stansifer, 1984], the first-order theory of the integer numbers. The Presburger arithmetic contains only equality, inequality and addition while omitting the multiplication operation. It has the major advantage of being a decidable theory. Therefore, one can perform most of the operations (*e.g.* analysis, transformations) using algorithms, even on a symbolic (*i.e.* with parameters) representation.

In the Polyhedral model, a program is represented as a computation graph. The nodes of the graph, each of which represents an iteration of a statement, are associated with a set of integer points. These points belong to polyhedra which are defined through a conjunction of inequalities. In turn, these polyhedra can be analyzed and transformed with the help of linear programming tools. This model is designed to analyze affine programs; programs where all loop bounds and array references are affine functions of the enclosing loop iterators and loop-invariant parameters. A parameter is a symbolic loop invariant; usually, the set of parameters bounds the problem size.

To summarize, a program fits in the Polyhedral model in the following situations.

- The controls are static: predicates, loop bounds, loop increments and array subscripts shall only depend on literal constants or a finite number of variables that are either constant known at runtime or iterators of enclosing loops.
- The controls are affines: predicates, loop bounds and array subscripts shall be multi-dimensional affine functions of the variables while loop increments must be literal constants.

Therefore, breaking the control flow with instructions such as `break`, `goto` or `return` is illegal in such program. It is common in the literature to

name this class of program a *Static Control Parts* (*SCoP*). To illustrate this concept, the C code presented in Listing 3.6 is a valid *SCoP*. On the contrary, the code in Listing 3.7 is **not** a valid *SCoP*. Indeed, both the first loop bounds and the second `C` array access are not affine functions.

```
double A[M][N], B[N][P], C[M][P];
for(int i=0; i<M; i++)
  for(int j=0; j<P; j++){
    C[i][j] = 0; //Statement S0
    for(int k=0; k<N; k++)
      C[i][j] = A[i][k] + B[k][j]; //Statement S1
  }
```

Listing 3.6 – Example of a valid Static Control Parts.

```
double A[M], C[M*N];
  for(int i=0; i<M*N; i++)
    C[i] = 0; //Statement S0
    for(int j=0; j<M-1; j++)
      C[(i/M)*j] += A[j] * A[j+1]; //Statement S1
  }
```

Listing 3.7 – Example of an invalid Static Control Parts.

3.5.2 Statement vector and iteration domain

In the Polyhedral model, a statement is represented by a vector of the values of all its enclosing loop indexes. Thus, an occurrence of a statement, or instance, at loop depth n is represented by a statement vector of size n. In addition, the statement domain is a compact way to represent all the instances of a given statement. It represents the set of all possible values of the statement vector.

```
for(int i=1; i<M; i++)
```

```
  for(int j=1; j<N; j++)
    S(i, j);
```

Listing 3.8 – Affine Static Control Parts example - 1.

```
for(int i=1; i<M; i++)
  #pragma omp parallel for
  for(int j=1; j<N; j++)
    S(i, j);
```

Listing 3.9 – Affine Static Control Parts example - 2.

For the sake of explanation, let's consider the C code presented in Listing 3.8. It is composed of a two-dimensional loop nest containing a single statement, S, which depends on the values of i and j. Enumerating all the S instances is not possible because of the N and M parameters, but if we tried, it would look like the following set:

$$\begin{array}{cccc} (1,1) & (1,\,2) & \dots & (1,\,\mathrm{N}) \\ (2,1) & (2,\,2) & \dots & (2,\,\mathrm{N}) \\ \dots & \dots & \dots & \dots \\ (\mathrm{M},1) & (\mathrm{M},\,2) & \dots & (\mathrm{M},\,\mathrm{N}) \end{array}$$

With conciseness, the Polyhedral model proposes to represent instances of S by a two-dimensional vector and its domain, $\mathcal{D}_S$, by the set of values this vector can take. Mathematically, this is expressible as is:

$$\mathcal{D}_S(M,N) = \{(i,j) \in \mathbb{Z}^2 \mid (1 \leq i < M) \wedge (1 \leq j < N)\} \tag{3.1}$$

$$\begin{cases} i-1 & \geq 0 \\ j-1 & \geq 0 \\ -i+M & \geq 0 \\ -j+N & \geq 0 \end{cases} \Rightarrow \mathcal{D}_S(M,N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \middle| \begin{pmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ M \\ N \\ 1 \end{pmatrix} \geq 0 \right\} \tag{3.2}$$

To express an iteration domain, the bounds are extracted to form a system of inequalities. The constraint matrix encodes a canonical form of the

affine inequalities of a Static Control Parts, as illustrated on Equation (3.2). Each row corresponds to a single inequality. Polyhedral tools have a more suited means to express these polyhedra. Indeed, in the Polyhedral model, parameters as well as all points in a polyhedron always belong to the set of integers. For instance, the *isl* library notes Equation (3.1) as is:

$$[M, N] \rightarrow \{S[i, j] : 1 \leq i < M \text{ and } 1 \leq j < N\}$$

In the literature, this is often illustrated using a graphical representation, as done in Figure 3.2. Every dot in the figure represents an integer point; hence a S instance.

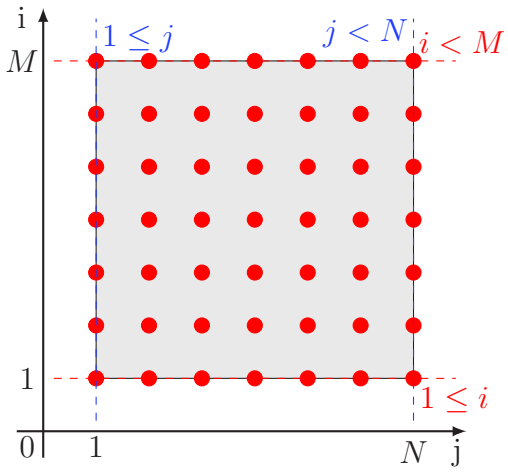


Figure 3.2 – Graphical representation of the polyhedron described in Equation (3.1). Each red dot represent a couple of integer (i, j).

3.5.3 Scattering functions

While the iteration domain contains the set of instances of the program statements, the scattering functions add constraints on the ordering of these instances. These are relations that maps a statement vector to a time vector. This last vector represents a virtual multi-dimensional space over which instructions are executed in the lexicographic order. The lexicographic order [Baader and Nipkow, 1998, Robbiano, 1985] is a generalization, to

vectors, of the way words are ordered based on the alphabetical order of their component letters. For instance, a vector of dimension d, v_1^d, is said lexicographically equal to a vector v_2^d if their d elements are equal. On the contrary, v_1^d is said lexicographically greater if, considering n elements, such that $0 \leq n < d$, v_1^d and v_2^d are lexicographically equal, while the $n + 1^{th}$ element of v_1^d is greater than the $n + 1^{th}$ element of v_2^d.

To illustrate scattering functions, let's consider the Listing 3.8 again. It presents a program calling a single statement, S. We defined the iteration domain of S as Equation (3.1). Loops are arranged such that each instance is called following the i then j values. In the Polyhedral model, such ordering can be defined by the following scattering function:

$$\theta_S = \{(i, j) \rightarrow (i, j)\} \tag{3.3}$$

Hence, the $S(4, 4)$ instance is performed at the $(4, 4)$ logical time; and therefore is executed before $S(4, 5)$ which is performed at the $(4, 5)$ logical time. Scattering functions can also describe parallel executions. Considering the C code presented in Listing 3.9 which differs from the previous listing by a parallel execution of the most inner loop. Since, for a fixed value of i, all S instances can be executed in parallel, the scattering function does not need to consider the j value. This can be expressed by the following scattering function:

$$\theta_S = \{(i, j) \rightarrow (i, 0)\} \tag{3.4}$$

For a fixed value of i, all time vectors are lexicographically equal; and therefore, can be executed in parallel. Such scattering function defines a partial order, in comparison to a total order, in which all time vectors are lexicographically different. The scattering functions presented in Equations (3.3) and (3.4) can be respectively noted in the Polyhedral model, in *isl* form, as is:

$$\begin{aligned} &[M, N] \rightarrow \{S[i, j] \rightarrow [i, j]\} \\ &[M, N] \rightarrow \{S[i, j] \rightarrow [i, 0]\} \end{aligned} \tag{3.5}$$

Mathematically, the Polyhedral model defines that an instance u is executed before another instance v, that is, the time vector of u is lexicographically lesser than the time vector v, using the order comparator, $\prec$, noted: $u \prec v$.

3.5.4 Access functions

An Access function maps a statement vector to an array locations accessed, as read or write, by the statement. Note that a statement may access multiple cells in several arrays, and therefore; maybe be accompanied by multiple access functions. Considering the Listing 3.6, the statement named *Statement S1* reads in the `A` and `B` arrays while writes in the `C` array. We can note its set of access functions as is:

$$\begin{cases} Access^{A}_{\mathcal{R}_{S1}} = \{(i,j,k) \rightarrow (i,k)\} \\ Access^{A}_{\mathcal{W}_{S1}} = \{(i,j,k) \rightarrow \emptyset\} \\ Access^{B}_{\mathcal{R}_{S1}} = \{(i,j,k) \rightarrow (k,j)\} \\ Access^{B}_{\mathcal{W}_{S1}} = \{(i,j,k) \rightarrow \emptyset\} \\ Access^{C}_{\mathcal{R}_{S1}} = \{(i,j,k) \rightarrow \emptyset\} \\ Access^{C}_{\mathcal{W}_{S1}} = \{(i,j,k) \rightarrow (i,j)\} \end{cases} \tag{3.6}$$

We can note $\mathcal{R}_{S1}$ and $\mathcal{W}_{S1}$, the set of array locations respectively read from and written to by the statement $S1$. The set of access functions presented in Equation (3.6) is expressed in the Polyhedral model, and more specifically by the *isl* library, as is:

$$\begin{aligned} \mathcal{R} &:= [M,N] \rightarrow \{S1[i,j,k] \rightarrow A[i,k]; S1[i,j,k] \rightarrow B[k,j]\} \\ \mathcal{W} &:= [M,N] \rightarrow \{S1[i,j,k] \rightarrow C[i,j]\} \end{aligned} \tag{3.7}$$

Two instances u and v are in dependence, noted δ, if there is an intersection between the set of locations read and written by u and the same sets of v. Excluding the read dependence, which does not modify the state of the memory, this is noted:

$$u \ \delta \ v \Leftrightarrow (\mathcal{R}_u \cap \mathcal{W}_v) \cup (\mathcal{W}_u \cap \mathcal{R}_v) \cup (\mathcal{W}_u \cap \mathcal{W}_v) \neq \emptyset \tag{3.8}$$

3.5.5 Implementation of the Polyhedral model

There are several libraries available that can be used to manipulate a polyhedral model representation of a program, including the *Omega library* [Kelly et al., 1995], *PolyLib* [Loechner et al., 1999], *PPL* [Bagnara et al., 2008] and *isl* [Verdoolaege, 2010]. Script languages also exist and some, such as the *iscc* calculator based on *isl*, are available online [Verdoolaege, 2014].

This thesis work relies on the *isl* library. *isl* is a C library offering data structures representing parameterized sets and relations of integer points bounded by linear constraints. It also comes with a wide range of functions to manipulate these structures, including intersection, union, set difference, projection, transitive closure, Cartesian product or lexicographic optimization. It also offers more complex functionalities such as dependence analysis or statements scheduling in accordance with dependence. *isl* is still an active project, developed by Inria researchers and used by well-known compiler environments, such as *GCC* or *LLVM*. It is part of the *Polyhedral.info* organization and is available online at Integer Set Library website.

iscc is a script language offering to access easily the *isl* features. The language is presented in [Verdoolaege, 2007] and is adapted to the expression of sets and supports most *isl* features. Some parts of this thesis proposes an *iscc* description of programs.

Chapter 4

The Independent Kernel Scheduling programming model

Contents

As explained in Section 3.4, most existing approaches tend to lie on a spectrum from general but manual approaches to more and more domain specific and automatic approaches. While the first approaches support a large range of optimizations and application domains, they are implementation costly, do not support separation of concerns and make portability difficult. On the contrary, automated approaches reduce implementation costs and offer good separation of concerns and portability but restrain the range of supported domains and optimizations. Our analysis has highlighted that there are no existing approach that provides generality, perfor-

mance and separation of concerns while proposing two guidelines to design it: relying on user contributions, instead of a compiler, and proposing two concern-specific languages.

Thereby, to reuse and combine the advantage of existing approaches, we propose the *Independent Kernel Scheduling* (InKS) programming model. It separates algorithmic and optimization concerns in numeric simulation codes with the aim of improving both developers productivity and codes readability, as well as easing application portability. It comes with the InKS$_{\mathsf{PIA}}$ language to express the simulation algorithm, with no concerns for the performance, and the InKS$_{\mathsf{PSO}}$ language to describe the optimization part, based on the algorithm description.

This chapter presents the core of this thesis work: the InKS programming model. In Section 4.1, we start by classifying algorithmic and optimization concerns in a numeric simulation code using a basic stencil code as an example. Using this preliminary analysis as a keystone, in Section 4.2 we propose the InKS programming model. In Section 4.3, we present the InKS$_{\mathsf{PIA}}$ language which intends to express all algorithmic aspects while leaving optimization choices unspecified. Finally, in Section 4.4, we conclude the chapter and discuss the choices we made for the implementation of the InKS programming model.

4.1 Classification of concerns

In this section, we categorize the various aspects interleaved in simulation codes as either algorithmic or optimization concerns. This classification is at the root of the motivation behind the InKS programming model. We define the simulation *algorithm* as the parts of code that will never change no matter the architecture executing the code; in other words, the *platform independent* code. That corresponds to domain related code, namely mathematical solving derived from a fixed numerical scheme. The algorithm aims to describe the simulation and is developed mostly by domain scientists. On the contrary, we define as *optimization choices* any instruction that is not essential to the application validity, *i.e.* the resulting values for a given input, but responsible for its good performance; in other words,

the *platform specific optimizations*. Optimization choices, generally written by optimization specialists, must respect the simulation algorithm. That is, any valid optimization choices combined with its algorithm shall give the exact same values as result for a given set of inputs.

We categorize these concerns by analyzing multiple C implementations of a 7-points finite difference method 3D heat equation solver, based on the implementation of S. Kamil [Kamil, 2012]. Listing 4.1 shows the simplest of those implementations, based on a double-buffer strategy. Another implementation, presented in Listing 4.2, provides cache blocking over two of the three space dimensions that can be specifically tuned for the machine cache size. A third implementation, shown in Listing 4.3 uses recursive function calls to implement a cache oblivious method with implicit blocking in four dimensions (3 in space, plus time).

In these three examples, linearized arrays (Lines 7 and 8 of Listing 4.1) store the temperature values. The `i3D` macro (Line 1) maps from the 3D space coordinate of the mesh to the linear memory space. The time coordinate accessible in the arrays evolves during the simulation but slightly differently depending on the code version. In the version presented in Listing 4.1, `T1` contains the values for the current time-step while `T2` contains values remaining from the previous time-step mixed with values being computed for the next time-step. This is also true for the cache blocking version. However, in the cache oblivious version (Listing 4.3), the time-blocking aspect requires a different memory storage. The `T1` and `T2` arrays are stored inside an array of arrays that makes it possible to access one array or another using a modulo operation. One array contains values from odd time-steps only while the other contains values from even time-steps only. Many distinct time-steps are, however, stored inside each array at any given time depending on the space coordinate. Figure 4.1 illustrates the cache oblivious strategy on a 2D example. Note that the set of values computed along the simulation, in the 3D space + 1D time coordinate system, is the same for all versions of the code; that is, every coordinate in the convex 4D hyper-rectangle bounded by `0` and the `nx`, `ny`, `nz` and `nt` parameters. Being independent to the implementation version, we classify this set of values as part of the algorithmic concern in a simulation code.

```
1  int gnx, gny;
2  #define i3D(i, j, k, t) ((i)+(gnx)*((j)+(gny)*(k)))
3
4  void heat_equation(double** Res, int nx, int ny, int nz, int nt){
5    gnx = nx; gny=ny;
6    const size_t size = nx * ny * nz;
7    double* T1 = malloc(sizeof(double) * size);
8    double* T2 = malloc(sizeof(double) * size);
9
10   for (int k = 0; k < nz; k++)
11     for (int j = 0; j < ny; j++)
12       for (int i = 0; i < nx; i++){
13         T1[i3D(i, j, k)] =
14           i*(nx-i-1)/nx +
15           j*(ny-j-1)/ny +
16           k*(nz-k-1)/nz;
17         T2[i3D(i, j, k)] = T1[i3D(i, j, k)];
18       }
19
20   for (int t = 1; t < nt; t++){
21     for (int k = 1; k < nz - 1; k++){
22       for (int j = 1; j < ny - 1; j++)
23         for (int i = 1; i < nx - 1; i++)
24           T1[i3D(i, j, k)] =
25             T2[i3D(i, j, k + 1)] + T2[i3D(i, j, k - 1)] +
26             T2[i3D(i, j + 1, k)] + T2[i3D(i, j - 1, k)] +
27             T2[i3D(i + 1, j, k)] + T2[i3D(i - 1, j, k)] -
28             6 * T2[i3D(i, j, k)];
29     }
30     swap(T1, T2);
31   }
32
33   free(T1);
34   *Res = T2;
35 }
```

Listing 4.1 – C implementation of the 3D finite difference heat equation solver using a double-buffer strategy.

On the contrary, the memory layout to store and read these values differs from an implementation to another. Still, the chosen memory layout shall respect the simulation algorithm; and therefore, being able to store the set of considered values. Hence, we catalog the memory layout as part of the optimization choices.

All studied implementations of the heat equation solver use loops. The body of these loops is made of computations that operate on the arrays content (Lines 24 to 28 of Listing 4.1) and is very similar from one version of the code to the other apart from indexing issues previously discussed. On the other hand, the control part of the loops that gives values to indexes and schedules computations inside the loops differs from one implementa-

```
1  int gnx, gny;
2  #define i3D(i, j, k, t) ((i)+(gnx)*((j)+(gny)*(k)))
3
4  void heat_equation(double** Res,
5    int nx, int ny, int nz, int nt, int TI, int TJ){
6    gnx = nx; gny = ny;
7    /*... Initialization ...*/
8
9    for (int t = 1; t < nt; t++){
10     for (jj = 1; jj < ny-1; jj += TJ) {
11       for (ii = 1; ii < nx-1; ii += TI)
12         for (k = 1; k < nz-1; k++)
13           for (j = jj; j < MIN(jj+TJ, ny-1); j++)
14             for (i = ii; i < MIN(ii+TI, nx-1); i++)
15               //Computation
16     }
17     swap(T1, T2);
18   }
19
20   free(T1);
21   *Res = T2;
22 }
```

Listing 4.2 – C implementation of the 3D finite difference heat equation solver using a 2D cache blocking strategy.

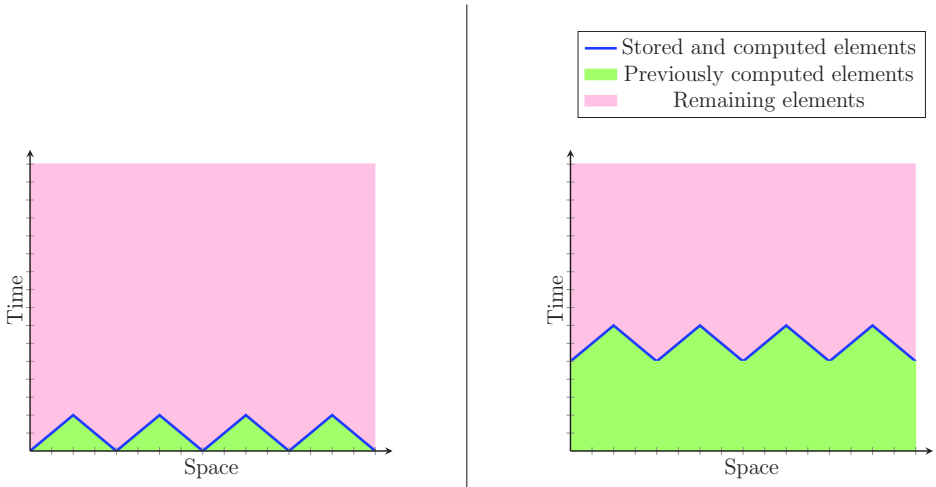
tion to the other. In the example from Listing 4.1 the loops iterate in a pretty straightforward order whereas those from Listing 4.2 used for cache blocking are more complex. In the cache oblivious examples, the loops are different again and the iterations depend on parameters of the recursive function calls. Although, these schedules differ both in terms of expression and performance, they all respect ordering constraints, namely that any value has to be written to memory before it is first read and that its storage space in memory must not be reused for another value before it is last read. Moreover, note that all schedules traverse the same set of coordinates; that is, the bounded 4D polyhedra described earlier. Therefore, the content of the loops, the set of traversed coordinates as well as these ordering constraints thus constitute parts of the simulation algorithm, while the choice of a specific schedule that respects the constraints is an optimization choice.

All the examples end with the values from the target time-step in the `T2` array. Among all the computed values, we can consider this subset as the *result* of the program. In addition, note that the values at the initial time-step, instead of being generated inside this part of the code, could come from the caller function. Hence, similarly to the result of the program, a

```
int gnx, gny;
#define i3D(i, j, k) ((i)+(gnx)*((j)+(gny)*(k)))

void F(double* A[], int t0, int t1, int x0, int dx0, int x1, int dx1,
int y0, int dy0, int y1, int dy1, int z0, int dz0, int z1, int dz1) {

  int dt = t1-t0;
  if (dt == 1 || (x1-x0)*(y1-y0)*(z1-z0) < 4096) {
    for (int t=t0;t<t1;t++)
      for (int z=z0+(t-t0)*dz0;z<z1+(t-t0)*dz1;z++)
        for (int y=y0+(t-t0)*dy0;y<y1+(t-t0)*dy1;y++)
          for (int x=x0+(t-t0)*dx0;x<x1+(t-t0)*dx1;x++)
            A[(t+1)%2][i3D(x,y,z)] = A[t%2][i3D(x+1,y,z)]
            + A[t%2][i3D(x-1,y,z)] + A[t%2][i3D(x,y+1,z)]
            + A[t%2][i3D(x,y-1,z)] + A[t%2][i3D(x,y,z+1)]
            + A[t%2][i3D(x,y,z-1)] - 6*A[t%2][i3D(x,y,z)];
  }else if (dt > 1) {
    if (2* (z1-z0) + (dz1-dz0) * dt >= 4 * ds * dt){
      int zm = (2* (z0+z1) + (2*ds+dz0+dz1) * dt) / 4;
      //Recursive calls
      F(A,t0,t1,x0,dx0,x1,dx1,y0,dy0,y1,dy1,z0,dz0,zm,-ds);
      F(A,t0,t1,x0,dx0,x1,dx1,y0,dy0,y1,dy1,zm,-ds,z1,dz1);
    }else if (2*(y1-y0) + (dy1-dy0)*dt >= 4*ds*dt){
      int ym = (2* (y0+y1) + (2*ds+dy0+dy1) * dt) / 4;
      //Recursive calls
      F(A,t0,t1,x0,dx0,x1,dx1,y0,dy0,ym,-ds,z0,dz0,z1,dz1);
      F(A,t0,t1,x0,dx0,x1,dx1,ym,-ds,y1,dy1,z0,dz0,z1,dz1);
    } else {
      int s = dt/2;
      //Recursive calls
      F(A,t0,t0+s,x0,dx0,x1,dx1,y0,dy0,y1,dy1,z0,dz0,z1,dz1);
      F(A,t0+s,t1,x0+dx0*s,dx0,x1+dx1*s,dx1,y0+dy0*s,dy0,
      y1+dy1*s,dy1,z0+dz0*s,dz0,z1+dz1*s,dz1);
    }
  }

}

void heat_equation(double** R, int nx, int ny, int nz, int nt){
  gnx = nx; gny = ny;
  double* T1 = malloc(sizeof(double) * nx * ny * nz);
  double* T2 = malloc(sizeof(double) * nx * ny * nz);
  /*... Initialization ...*/
  double* A[2] = {T1, T2};
  F(A, 1, nt, 1, 0, nx-1, 0, 1, 0, ny-1, 0, 1, 0, nz-1, 0);
  *R = A[nt%2];
}
```

Listing 4.3 – C implementation of the 3D finite difference heat equation solver using a cache oblivious strategy.

subset of the existing values can be used as the input of the program. This subset is invariant no matter the implementation version; despite this set being empty in our example. These two sets also constitute a part of the application algorithm.



(a) Element status at $t1$ (b) Element status at $t2$

Figure 4.1 – Graphical representation of the *cache oblivious* strategy applied on a 2D array. Two consecutive instants, $t1$ and $t2$, are represented. At each instant of the simulation, values of several time-steps are stored inside the array (blue area).

To summarize, we have identified five concerns that form the simulation algorithm of the 3D heat equation solver, and more generally algorithm of numerical simulation codes. That is, the values that exist during the execution, the computations done inside the loops, the set of coordinates to traverse, the constraints on computation order and the subsets of values used as input or expected as result. We have also identified two types of optimization choices: the mapping in memory of the existing values and the specific scheduling of computations. All these concerns are cataloged in Table 4.1. In other examples, more optimization choices could appear if we consider parallel execution. For instance, distributed memory parallel versions of the code, where choices related to the distribution of data on nodes and communications would have to be made. Nevertheless, if the simulation algorithm contains enough information to derive a sequential version of the code, a parallel version will not require more information.

	Algorithm	**Optimization**
Data concerns	Values existence Simulation input & output	Memory layout
Computation concerns	Computation expression Constraint on order Traversed coordinates	Computation ordering

Table 4.1 – Classification of simulation program concerns as related to either algorithmic or optimization.

4.2 The InKS programming model

In the previous section, we have identified a specific set of concerns that relates either to the simulation algorithm or the optimization choices. We now describe the core and motivations behind the INdependent Kernel Scheduling (INKS) programming model. As a reminder, we have defined the simulation *algorithm* as a code related to the simulated domain, that is mathematical solving based on a fixed numerical scheme. On the contrary, *optimization choices* refer to all instructions responsible for the application good performance.

All in all, the examples presented in the previous section and written in C interleave algorithmic and optimization concerns. A complete new code is written to assay each distinct optimization. On the contrary, one would like a programming model that clearly separates these two aspects. This separation comes with, on one side, the simulation algorithm as the invariant part of the code, and on the other side, a set of optimization choices derived from this algorithm. In the middle, a compiler takes care of merging the algorithm with optimization choices.

Just as existing works, our main objective is to ease the development of numeric simulation codes, in comparison to the complexity of general-purpose languages such as C or Fortran. However, we differ from existing works in the way followed to achieve this goal. We propose to limit the cost of tuning an application not by encoding reusable sets of optimization strategies or by counting on an effect compiler or runtime; but by relying on an invariant part of the code: the simulation algorithm. Setting the simulation algorithm exhibits several advantages compared to traditional

approaches.

Firstly, it eases the collaboration between specialists of the simulated domain and specialists of computer optimizations. Each can focus on the parts of code that depend upon its knowledge without being forced to apprehend concepts outside of his domain of competence.

Secondly, similarly to *DSL* approaches such as algorithmic skeletons or stencil-specific *DSL*, an algorithm dedicated language offers users an abstraction to convey their actual objectives to the underlying compiler. Hence, a compiler can use these pieces of information to automatically optimize parts of the code, for example by getting facts related to data alignment or vectorization capabilities that can be transferred into the generated code. Contrariwise, general-purpose approaches ordinarily fail in understanding their users actual goals, forcing optimization specialists to manually specify such optimizations.

Finally, and more importantly, it limits the parts of code that must be rewritten to implement new optimization strategies. This is especially interesting with upcoming architectures, on which optimization tools may not be efficient. Indeed, the bleeding edge of supercomputers architectures may be equipped with computing units and hierarchical memories for which the best optimization strategies are currently unknown. Narrowing the part of code to rewrite diminishes implementation costs; and therefore, makes possible a more extensive exploration of the best optimization strategy for each architecture, especially the upcoming ones. In addition, it eases the maintaining of multiple versions of code, since they are all based on a single algorithm. Indeed, while some algorithm modifications (*e.g.* constraints on order) may require adaptations of the optimization choices, others, such as the actual computations, are transparent. For instance, let's consider an algorithm and three versions of its optimization choices. Using a general-approach, one may have three independent codes, each repeating the algorithm part. Hence, fixing the forgetting of a minus sign in one of the version, does not prevent the need to repeat the bug fix in the two other versions. On the contrary, with InKS, as this operation fix is part of the algorithm, it would be automatically affect all versions of the code.

Even if we believe this separation of concerns is an important goal of such

programming model, this shall, however, not come at the cost of benefits offered by existing works. Therefore, such programming model shall not increase the complexity of specifying the simulation algorithm. Similarly, it shall also be possible to specify any optimization while the specification of these choices shall not be much more complex than it currently is in existing approaches. It shall be possible to express a wide range of different problems in this language so as to cover as many simulation domains as possible and to include this inside another program written in existing languages, such as C or Fortran for example, to make the progressive adoption of the programming model possible. Finally, since the whole simulation algorithm is available, a compiler should enable users to test the algorithm without having to specify complex optimization choices. In summary, our approach shall accomplish all the following objectives:

1. Separates algorithmic and optimization concerns;
2. Provides generality;
3. Being as efficient as existing approaches;
4. Being not more complex than existing approaches;
5. Supports algorithm testing.

This section presents the InKS programming model which aims to fulfill such objectives.

The InKS programming model, illustrated in Figure 4.2, proposes two distinct languages to separate the expression of the algorithm and optimization choices. Relying on two languages enables us to separate the concerns both syntactically, in distinct set of files, and semantically, with distinct notions and concepts adapted to the aspects the users need to express. The first one is the InKS platform-independent algorithm ($\mathbf{InKS_{PIA}}$) language to enable the full expression of the simulation algorithm independently of any concern for optimization choices. That includes the aspects identified in the previous section:

- the set of values computed along the program execution;

- the subset of values used as input and output of the program;
- the set of computations that composes the program;
- the set of coordinates at which each computation is applied, and
- the constraints on the ordering of the computation.

This description is primarily produced by domain scientists.

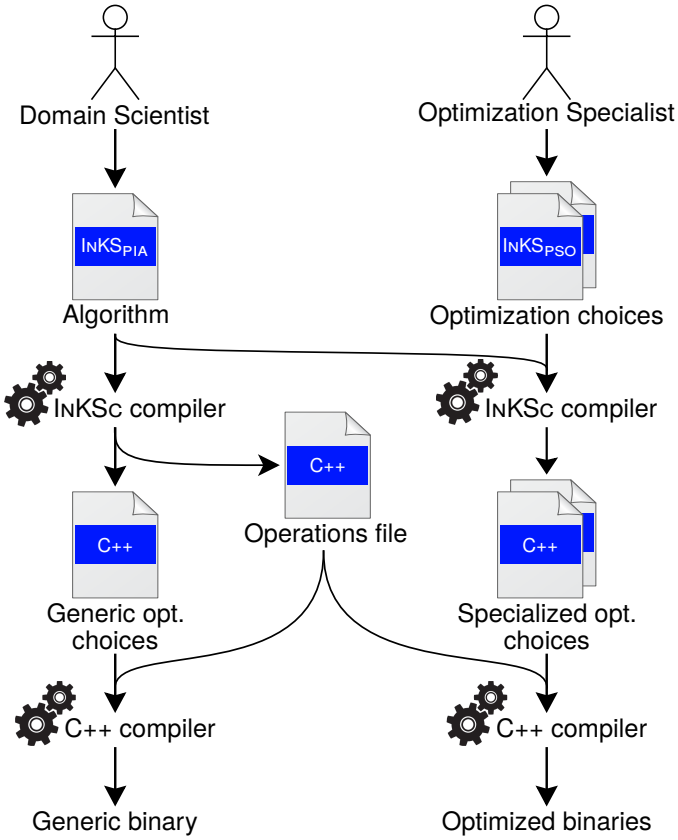


Figure 4.2 – The INKS programming model

The INKS programming model also provides the INKS platform-specific optimization (**InKS$_{PSO}$**) language. INKS$_{PSO}$ depends upon an INKS$_{PIA}$ algorithm to express the optimization choices part of that algorithm. This concern relates in particular to memory layouts as well as computation scheduling, as identified in Section 4.1. Thereby, this includes the computing unit selected for each computation, the total, or parallel, ordering of these computations, the memory location for each value, etc. All other

pieces of information are gathered from the InKS$_{\mathsf{PIA}}$ algorithmic code. Optimization choices, generally written by optimization specialists, must respect the simulation algorithm. In particular, the chosen computation scheduling must respect the data dependence expressed in the InKS$_{\mathsf{PIA}}$ code while the memory layouts must enable computations to access the data they specified. Thereby, any valid optimization choices combined with its algorithm shall give the exact same values as results, for a given input. Yet, these results may slightly vary because of rounding errors for floating point operations; although this can be generally addressed with compilation options. Note that domain scientists and optimization specialists need not necessarily be different persons. Even in this case, the separation of concerns minimizes the code rewriting for each new optimization.

A InKS program consists of the combination of a simulation algorithm, written in InKS$_{\mathsf{PIA}}$, and one of its varied optimization versions, based on InKS$_{\mathsf{PSO}}$. In between, the InKS programming model comes with a compiler that takes care of merging both aspects. Such program supports the C calling convention, and therefore, can be called from a C, C++ or Fortran program for instance.

4.3 InKS$_{\mathsf{PIA}}$: expressing the algorithmic concern in the InKS programming model

In this section, we present the InKS$_{\mathsf{PIA}}$ language of the InKS programming model. InKS$_{\mathsf{PIA}}$ intends to describe algorithmic concerns with no regards for their latter optimization. In Chapter 3, we have presented several approaches among which three focused, at least partially, on achieving such goal: *XFor* [Fassi and Clauss, 2015], *PATUS* [Christen et al., 2011] and *LARA* [Cardoso et al., 2012]. While *XFor* considers loop statements partial ordering, *PATUS* focuses on stencil expressions. In both approaches, optimization choices are derived from this algorithm description in a second step. Although these approaches share similarities with the objective of the InKS$_{\mathsf{PIA}}$ language, they focus on too specific concerns, namely loop

ordering and stencil optimization. However, it is interesting to notice that *PATUS* proposes a Domain Specific Language to express the stencil description. Similarly, *XFor* relies on a *DSL* to express the invariant part of the program. On the contrary, *LARA* is a general-purpose proposition based on the Aspect-Oriented Programming. Thereby, in a first step, C is used to describe domain science while in a second step, optimization specialists can express code transformation using the *LARA* language. AOP approaches achieve the two main goals of the InKS programming model, namely generality and separation of concerns, however, it comes at performance cost. Indeed, it offers only a limited set of transformations, preventing a wide range of optimizations.

At first glance, all approaches follow the same strategy, proposing a language to express the specific algorithmic concerns they handle. However, *XFor* actually relies on a *DSL* that targets the Polyhedral model. The Polyhedral model offers to describe a program as a collection of statements gathered in polyhedra and a set of relations describing which statement shall be executed before which other one. Affine transformations can then be applied to this representation at compile-time to modify the program while preserving its correctness, *i.e.* results do not differ for a given input. For instance, such transformations can expose parallelism possibilities or improve data locality. The Polyhedral model is presented with more details in Section 3.5. As a matter of fact, *XFor* uses this model and adheres to its concepts, proposing predefined transformations which, once combined with the algorithm description originally provided, produces efficient loop statement scheduling. With this strategy, *XFor* separates the loop description from its optimization. Although InKS$_{\mathsf{PIA}}$ aims for generality, in order to express complete simulation code, we propose to follow the same strategy. That is, we provide InKS$_{\mathsf{PIA}}$ as a *DSL* that targets the Polyhedral model and which is adapted to express a complete simulation algorithm. In a second step, we can propose transformation of the program while preserving its correctness. This approach presents three major advantages. First, it enables the InKS$_{\mathsf{PIA}}$ language to express a wide variety of programs; that is, the one supported by the Polyhedral model: *Static Control Parts* (*c.f.* Section 3.5.1). Secondly, InKS$_{\mathsf{PIA}}$ code representation is com-

patible with transformations expressible in the Polyhedral model. This includes various tools based on this model developed by the scientific community (*e.g. Pluto* [Bondhugula et al., 2008], *PPCG* [Verdoolaege et al., 2013] or *CHiLL* [Basu et al., 2017]), but also potential implementations of the InKS optimization language, InKS$_{\mathsf{PSO}}$. Finally, the Polyhedral model enables static analysis and helps us to insure the code against potential implementation errors.

Thereby, the InKS programming model reuses and combines advantages of the existing works:

- the possibilities of fine and manual optimizations offered by both general imperative languages and dedicated tools which is found in InKS$_{\mathsf{PSO}}$;
- the ease of optimization offered by dedicated tools through an InKS$_{\mathsf{PSO}}$ interface and;
- the separation of concerns and performance portability offered by *DSLs* thanks to the double InKS languages;
- all this while offering a substantial level of generality through the Polyhedral model.

4.3.1 The InKS$_{\mathsf{PIA}}$ concepts

InKS$_{\mathsf{PIA}}$ is a purely declarative language which aims to describe a simulation algorithm while *not* limiting the set of potential optimization choices. The type of choices left unspecified in InKS$_{\mathsf{PIA}}$ include for example the order of execution of operations (provided data dependencies are met), the memory layout or the memory reuse along execution. A InKS$_{\mathsf{PIA}}$ code is composed of three parts. The first one is a set of *logical arrays* in which each cell can be written at most once. The second part consists of very fine grain *operations* containing some mathematical operations, without any loops. These operations read from and write to cells in logical arrays. These reading and writing constitute data dependencies. The last part declares for each operation a set of coordinates for which the operation can be

executed. Ultimately, an InKS$_{\text{PIA}}$ code specifies a directed graph of operations working at very fine grain on logical arrays. Each operation depends on or is necessary to the execution of other operations. This corresponds to a parameterized task graph (PTG) [Cosnard and Loi, 1995, Cosnard and Jeannot, 1999]: a directed acyclic graph of tasks (here operations) in a compact representation, independent of problem size. To summarize, InKS$_{\text{PIA}}$ expresses two aspects of the algorithm: the existence of values, using logical arrays, and computations and their partial ordering, using operations and their data dependencies and domain.

InKS$_{\text{PIA}}$ is used to express the aspects related to the simulation algorithm, identified in Section 4.1:

- the set of values computed along the program execution;
- the subset of values used as input and output of the program;
- the set of computations that composes the program;
- the set of coordinates at which each computation is applied, and
- the constraints on the ordering of the computation.

The first objective of the language is to express the set of values computed along the simulation without hard-coding a memory layout. To achieve such goal, InKS$_{\text{PIA}}$ separates physical and logical memory spaces by providing infinite multidimensional logical arrays based on dynamic single assignment (DSA) [Vanbroekhoven, 2002] to store values. DSA form qualifies a program that does not perform any destructive update of scalars and array elements; that is, each element is written at most once. A logical array can be seen as a pure function: a given input always gives the same output. That is to say, a logical array coordinate corresponds to a unique value. In the case of an algorithm with a time evolution for example, a dimension must be added to arrays to represent time as the same coordinate cannot be reused for values of different time-steps. These logical arrays are used to address the data manipulated by the algorithm but do not impose memory storage. Therefore, memory placement of each coordinate is left unspecified.

Logical arrays are defined in INKS$_{\text{PIA}}$ using the constructs presented in Listing 4.4. In addition, the logical array construct specifies the subset of values used as input and output of a computation. These subsets are defined as polyhedra, *i.e.* a union of polyhedron, which INKS$_{\text{PIA}}$ construct is presented in Listing 4.5. In INKS$_{\text{PIA}}$, a n dimensional polyhedron is defined by n successive constraints, the n^{th} bounding the n^{th} dimension of the polyhedron. Hence, the number of constraints (`cstr` in Listing 4.5) specify the number of dimensions of the polyhedron. A constraint is a range which each boundary can either be set to an affine function of the existing parameters or left unspecified. In the latter case, the boundary is set to its default value, *i.e.* $-\infty$ for the left boundary and $+\infty$ for the right one.

```
<str:data_type> <str:array_name>(<int:dim_number>)
(
  ([in|out]: <polyhedra:in_out_data>)
    |
  (in: <polyhedra:in_data> | out: <polyhedra:out_data>)
)?
```

Listing 4.4 – The INKS$_{\text{PIA}}$ logical array construct.

```
val := [id | int]
expr := [val | expr+expr | expr-expr | -expr | int*expr]
min, max := expr
cstr := [: | expr | min: | :max | min:max | cstr%int]
polyhedron := (id=[val:val[)* (cstr*)
polyhedra := (id=[val:val[)* { polyhedron+ }
```

Listing 4.5 – The INKS$_{\text{PIA}}$ polyhedra construct.

Similarly, to express computations without hard-coding a scheduling, INKS$_{\text{PIA}}$ offers two main constructs: *operation* procedures and *simulation* kernels. The operation procedure represents a fine grain computation. Each operation procedure (illustrated in Listing 4.7) takes as parameters logical coordinates (Line 1: `x`, `t`) and logical arrays (Line 2: `my_array`). It specifies

the coordinates readable and written in each array relative to the coordinates taken as parameters (Line 2: `(x,t-1)` and `(x,t)`). It defines a C++ implementation which specifies a fine grain computation that can access the logical arrays values using the parenthesis operator (Line 5). The operation procedure construct is illustrated in Listing 4.6.

```
op <str:op_name>(<str:coord_name>*)(
  <logical_array:used_logical_array>+
)
#CODE (C++)
  <C++ code>
#END
```

Listing 4.6 – The INKS$_{PIA}$ operation procedure construct.

```
1 op square(x, t) : (
2   double my_array {in: (x, t-1) | out: (x, t) }
3 )
4 #CODE (C++)
5   my_array(x,t) = my_array(x, t-1) * my_array(x, t-1);
6 #END
```

Listing 4.7 – Example of INKS$_{PIA}$ operation procedure.

The simulation kernel is the entry point of the simulation (illustrated in Listing 4.9). It takes as parameters integers unknown at compile-time, but invariant during execution (Line 1: `X`, `T`) that typically define the dimensions of the problem. It declares the logical arrays that exist in the simulation, their type and number of dimensions (Line 2: `double full_array(2)`). Its input and outputs are specified as regions in the logical arrays that must be provided for the algorithm to run (Line 2: `(0:X,0)`) and that are made available after its execution (Line 2: `(0:X,T-1)`). Its implementation in a dedicated language specifies the domains of application of operation procedures (Line 5): coordinates where the operation can generate values in the logical arrays. Hence, operations execution order is left unspecified. On the contrary, constraints on the execution order are automatically deducted by analyzing data dependencies between operations. More complex usage of INKS$_{PIA}$ are presented in Section 6.2. The simulation kernel construct is

Algorithmic concerns		**InKS$_{\text{PIA}}$ construct**
Values existence	$\mapsto$	Simulation logical arrays
Simulation input & output	$\mapsto$	Simulation subset of logical arrays
Computation expression	$\mapsto$	Operation C++ code
Constraint on order	$\mapsto$	Operation subset of logical arrays
Traversed coordinates	$\mapsto$	Operation domain

Table 4.2 – Correspondence between algorithmic concerns and InKS$_{\text{PIA}}$ construct.

presented in Listing 4.8. Table 4.2 presents the correspondence between the algorithmic concerns identified in Section 4.1 and the InKS$_{\text{PIA}}$ constructs used to express them.

```
simulation <str:sim_name>(<str:parameter_name>*)(
  <logical_array:existing_logical_array>+
)
#CODE (InKS)
  (<str:op_name> <polyhedra:domain> :
     (<str:array_name>+))*
#END
```

Listing 4.8 – The InKS$_{\text{PIA}}$ simulation kernel construct.

```
1 simulation my_simulation(X, T) : (
2   double full_array(2) {in: (0:X, 0) | out: (0:X, T-1)}
3 )
4 #CODE (INKS)
5   square(0:X, 1:T) : (full_array)
6 #END
```

Listing 4.9 – Example of InKS$_{\text{PIA}}$ simulation kernel.

4.3.2 Illustrative example

We now use the 3D heat equation solved using the finite difference method to illustrate the InKS$_{\text{PIA}}$ syntax and concepts. Listing 4.1 presents the C implementation whereas Listing 4.10 shows the InKS$_{\text{PIA}}$ implementation.

```
1  operation Boundary(x, y, z, t) : (
2    double H {in: (x, y, z, t-1) | out : (x, y, z, t)} )
3  #CODE (C)
4    H(x, y, z, t) = H(x, y, z, t-1);
5  #END
6
7  operation Inner(x, y, z, t) : (
8    double  H {in: (x-1:x+2, y, z, t-1) ; (x, y-1:y+2, z, t-1);
9          (x, y, z-1:z+2, t-1) | out : (x, y, z, t)})
10 #CODE (C)
11   H(x, y, z, t) = H(x+1, y, z, t-1) + H(x-1, y, z, t-1)
12   + H(x, y+1, z, t-1) + H(x, y-1, z, t-1) + H(x, y, z+1, t-1)
13   + H(x, y, z-1, t-1) - 6.0 * H(x, y, z, t-1);
14 #END
15
16 operation Init(i, j, k, X, Y, Z, t) : (
17   double  H {out: (i, j, k, t)}
18 )
19 #CODE (C)
20   H(i, j, k, t) = i*(X-i-1)/X + j*(Y-j-1)/Y + k*(Z-k-1)/Z;
21 #END
22
23 simulation inks_heat(nx, ny, nz, nt) : (
24   double Heat(4) {out: (0:nx, 0:ny, 0:nz, nt-1) }
25 )
26 #CODE (INKS)
27   Boundary it=[1:nt[ {
28         iy=[0:ny[ iz=[0:nz[ {(0, iy, iz, it); (nx-1, iy, iz, it)};
29         ix=[0:nx[ iz=[0:nz[ {(ix, 0, iz, it); (ix, ny-1, iz, it)};
30         ix=[0:nx[ iy=[0:ny[ {(ix, iy, 0, it); (ix, iy, nz-1, it)}}
31       : (Heat),
32   Inner (1:nx-1, 1:ny-1, 1:nz-1, 1: ) : (Heat),
33   Init (0:nx, 0:ny, 0:nz, nx, ny, nz, 0) : (Heat)
34 #END
```

Listing 4.10 – INKS$_{PIA}$ implementation of the 3D finite difference heat equation solver.

Lines 1, 7 and 16 define three operations while Line 23 declares the simulation. First, we focus on the operations, and more specifically on the `Inner` one, Line 7, which expresses the 7-points stencil computation. It uses a four dimensional logical array containing double precision floating-point values, named `H`, and a tuple of four integers, `x`, `y`, `z` and `t`. Between curly brackets, Line 8, the algorithm specifies that, for a given tuple (x, y, z, t) as parameter, the operation uses values as input, marked `in`, and generates some as output, marked `out`. More specifically, it reads seven values of the `H` logical array: $(x-1:x+2, y, z, t-1); (x, y-1:y+2, z, t-1); (x, y, z-1:z+2, t-1)$ and write a single value (x, y, z, t). Note that the colon operator, `':'`, expresses a set of values, from its left term included to its right term excluded. Finally, lines 10 to 14, the mathematical expression of

the computation is expressed in plain C++. In accordance with the input and output specification, the cells of the logical array `H` are accessed using parentheses and a tuple of integers.

Line 23 declares the simulation. It comes with a tuple of four integers that parameterize the simulation: `nx` the number of cells in the first space dimension, `ny` the number of cells in the second space dimension, `nz` the number of cells in the third space dimension, and `nt` the number of time-steps. It also declares the `Heat` 4D logical array in which a region is expected as output of the program; that is, all values in spacial dimension at the last time-step, `nt-1`. It is followed by the set of operations valid to execute to obtain such result. For instance, Line 32 defines the valid set of parameters of the `Inner` operation. The first set of parentheses defines the valid values for each integer parameter. For instance, the first integer parameter of the `Inner` operation, `x`, could be equal to any integer between `1` and `nx-1`. This set of parenthesis specify an hyperrectangle, more specifically, a four dimensional polyhedra. Note that it is a set of *valid* values; that is, they are not necessarily executed to produce the result. Therefore, the four dimension of the polyhedra is not bounded: the `t` can be equal to any value from `1` to $+\infty$. Despite the lack of upper bound, the `Inner` operation will not be executed for every `t` value, only until the data expected as output of the program are generated. Hence, in this example, this dimension could have been reduced from `"1:"` to `"1:nt"`. In the second set of parenthesis, it expresses that it shall be executed on the `Heat` logical array.

4.3.3 InKS$_{\text{PIA}}$ completeness analysis

Let us now roughly demonstrate that this constitutes the whole algorithm of a program; *i.e.* that the InKS$_{\text{PIA}}$ language carries enough information to execute it. This constructive proof also forms the first part of our compiler algorithm. If this proof fails, it means that the InKS$_{\text{PIA}}$ code is invalid (*e.g.* an operation may read values that were never generated).

The pieces of information available in the InKS$_{\text{PIA}}$ language are:

- the set of integer parameters $\mathcal{P}$;

- the set of logical arrays $\mathbb{A} = \{a_1, ..., a_n\}$;
- the dimension $dim(a) \in \mathbb{N}$ of each logical array $a \in \mathbb{A}$;
- the set of operations $\mathbb{K} = \{k_1, ..., k_n\}$;
- the validity domain $dom(k)$ of each operation $k \in \mathbb{K}$ with its dimension $n = dim(k)$, $dom(k) = \{x \in \mathbb{Z}^n \mid \mathcal{P}x + b \leq 0,\ b \in \mathbb{Z}\}$;
- the dependencies (inputs I_k and outputs O_k) of each operation k;
- the set of subset of logical arrays available at the beginning of the simulation $D_i = \{a_{i1}, ..., a_{in}\}$;
- the set of subset of logical arrays expected as result of the simulation $D_o = \{a_{o1}, ..., a_{on}\}$.

We call *operation instance* the association of an operation k with a coordinate from its domain $dom(k) \subseteq \mathbb{Z}^{dim(k)}$ and we denote K the set of all operation instances.

$$k \in \mathbb{K} : dom(k) \to K$$

$$K = \bigcup_{k \in \mathbb{K}} \left(\bigcup_{i \in dom(k)} k(i) \right)$$

Similarly, we call *data instance* the association of a logical array a with a coordinate from its domain $\mathbb{N}^{dim(a)}$. We denote D the set of all data instances and D_n the union of D and the coordinate that represents the lack of data instance $D_\emptyset$.

$$a \in \mathbb{A} : \mathbb{N}^{dim(a)} \to D$$

$$D = \bigcup_{a \in \mathbb{A}} \left(\bigcup_{i \in \mathbb{N}^{dim(a)}} a(i) \right)$$

$$D_n = D \cup D_\emptyset$$

The inputs I_k and outputs O_k dependencies of an operation k map each instance of this operation to the data it reads or writes. We denote I and O

the general input and output relations formed as the union of all operation dependencies that map operation instances to data.

$$O_k : K \to D$$

$$I_k : K \to D_n$$

$$I = \bigcup_{k \in \mathbb{K}} I_k, O = \bigcup_{k \in \mathbb{K}} O_k$$

In order for the program to be well-formed, a given data instance should only be produced by a single operation instance. That is, the intersection of the output relation applied to two distinct operation instances should always be empty. On the other hand, multiple operation instances can take the same data instance as input.

$$\forall k_1, k_2 \in K,\ k_1 \neq k_2 \Rightarrow O(k_1) \cap O(k_2) = \emptyset$$

Each subset of logical array a_o expected as result of the simulation is a subset of the data that the computation must generate, we denote T_o the union of all these data. Similarly, we denote T_i the union of all pieces of data available at the beginning of the simulation.

$$a_i \in D_i : a_i \subseteq D$$

$$T_i \subseteq D = \bigcup_{a_i \in D_i} a_i$$

$$a_o \in D_o : a_o \subseteq D$$

$$T_o \subseteq D = \bigcup_{a_o \in D_o} a_o$$

The algorithm describes a program that computes T_o. Consequently, for the program to be well-formed, T_o must be the output of some operation. We denote K_T this set of operation instances which generate the result data.

$$K_T \subseteq K, \left(\bigcup_{k \in K_T} O(k) \subseteq T_o \right)$$

In addition, a operation instance k_1 must be computed before an instance k_2, denoted $k_1 \prec k_2$ if k_1 generates data as output that k_2 accesses as input.

$$\forall k_1, k_2 \in K,\ O(k_1) \cap I(k_2) \neq \emptyset \iff k_1 \prec k_2$$

For the program to be well-defined, there must be no loop in its dependencies, and we can define $\leq$ the transitive closure of $\prec$ that constitutes a partial order relation on K. The set of operation instances K_x that must be executed to produce T_o is the set of all instances that come before at least one instance in K_T, in addition to K_T itself.

$$K_x = \{k \in K | \exists k_t \in K_T, k \leq k_t\} \cup K_T$$

The data D_x that will have to be allocated at some point for the execution of all K_x is the data that is the input or output of at least one such operation instance. For the program to be valid, all the data instances accessed as input of an executed operation instance must be part of the output of another operation instance, one can therefore define the allocated data from the output only.

$$D_x = \bigcup_{k \in K_x} I(k) \cup \bigcup_{k \in K_x} O(k)$$

To be valid, the program shall access data instances that either come from previous computations or from the data instances available at the beginning of the simulation.

$$\bigcup_{k \in K_x} I(k) \subseteq D_i \cup \bigcup_{k \in K_x} O(k)$$

One can augment the executed operation instances with instances representing the allocation $ka(d)$ and deallocation $kd(d)$ of each data instance d.

The order relation can also be defined on these operation instances by taking into account that a data instance must be allocated before it is written and deallocated after it is last accessed.

$$K_x^+ = K_x \cup \bigcup_{d \in D_x} \{ka(d), kd(d)\}$$

$$\forall (k,d) \in K_x \times D_x,\ d \in I(k) \iff ka(d) \prec k$$

$$\forall (k,d) \in K_x \times D_x,\ d \in O(k) \iff k \prec kd(d)$$

To summarize, the InKS$_{\text{PIA}}$ language provides enough information to construct the K_x^+ set that specifies all memory allocations and deallocations and operation instances to execute with a partial order relation that determines the constraints on the scheduling of these operations. Using Polyhedral tools such as *isl*, this set is the only piece of information required to generate a valid scheduling and memory layout. An implementation of a compiler using this information and *isl* is presented in Section 5.1.

4.3.4 Illustrative example analysis

In Section 4.3.2, we have illustrated InKS$_{\text{PIA}}$ usage on the 3D heat equation solved using the finite difference method. We now use this example, which InKS$_{\text{PIA}}$ code is shown in Listing 4.10, to picture the algorithm presented in the previous section. We implement this algorithm using the *iscc* calculator [Verdoolaege, 2014] and can be implemented with the *isl* library.

The first step consists in parsing the InKS$_{\text{PIA}}$ code to obtain the *isl* structures used as input of the algorithm. Note that in the simulation kernel, the operations use the `Heat` logical array. Hence, we represent the input and output relations of the operations as directly working on this logical array, instead of the non-specific parameter `"H"`. In addition, we use several times the `coalesce` function, which simplifies the sets and relations, in order to reduce compilation time. The parsing of Listing 4.10 gives us the following iscc code:

```
#Simulation validity domains => Operation instances
K_init := [nx, ny, nz, nt] -> {Init[i, j, k, X, Y, Z, t]:
  0<=i<nx and 0<=j<ny and 0<=k<nz and t=0 and X=nx and Y=ny and Z=nz};
```

```
K_inner := [nx, ny, nz, nt] -> {Inner[x, y, z, t]:
  1<=x<nx-1 and 1<=y<ny-1 and 1<=z<nz-1 and 1<=t};
K_boundary := [nx, ny, nz, nt] -> {Boundary[x, y, z, t]:
  1<=t<nt and exists(ix, iy, iz: 0<=ix< nx and 0<=iy<ny and 0<=iz<nz and
    ((y = iy and z = iz and (x=0 or x=nx-1)) or
    (x = ix and z = iz and (y=0 or y=ny-1)) or
    (x = ix and y = iy and (z=0 or z=nz-1))) )};

#Operation input/output relations
I_init := [nx, ny, nz, nt] -> {
  Init[i, j, k, X, Y, Z, t] -> Heat[i, j, k, t] : false};
O_init := [nx, ny, nz, nt] -> {
  Init[i, j, k, X, Y, Z, t] -> Heat[i, j, k, t]};
I_inner := [nx, ny, nz, nt] -> {
  Inner[x, y, z, t] -> Heat[x, y, o2, -1 + t] : -1 + z <= o2 <= 1 + z;
  Inner[x, y, z, t] -> Heat[x, o1, z, -1 + t] : -1 + y <= o1 <= 1 + y;
  Inner[x, y, z, t] -> Heat[o0, y, z, -1 + t] : -1 + x <= o0 <= 1 + x};
O_inner := [nx, ny, nz, nt] -> {Inner[x, y, z, t] -> Heat[x, y, z, t]};
I_boundary := [nx, ny, nz, nt] -> {Boundary[x, y, z, t] -> Heat[x, y, z, t-1]};
O_boundary := [nx, ny, nz, nt] -> {Boundary[x, y, z, t] -> Heat[x, y, z, t]};

#Simulation input/output
In_i := [nx, ny, nz, nt] -> {};
In_o := [nx, ny, nz, nt] -> {
  Heat[i, j, k, nt-1]: 0<=i<nx and 0<=j<ny and 0<=k<nz};
```

We can then construct K, I, O, T_i and T_o, respectively the set of all operation instances, the set of all data instances,, the input and output relations of all operations and the simulation's input and output:

```
K := K_init + K_inner + K_boundary;
I := coalesce((I_init + I_inner + I_boundary) * K);
O := coalesce((O_init + O_inner + O_boundary) * K);
T_i := In_i;
T_o := In_o;
```

The following step consists in computing D_i, the data instance that may be read during the execution, D_o, the data instance that may be written, K_T, the operation instances generating the simulation's output, and $\prec$ (`Order_relation`), the partial order relation between each operation instance:

```
D_i := coalesce(I(K));
D_o := coalesce(O(K));
K_t := coalesce((O^-1)(T_o));
#Transitive closure of the composition of O and reverse I
Order_relation := coalesce(coalesce(O . (I^-1))^+);
#Intersection with the kernel that may be executed
```

```
Order_relation := coalesce(Order_relation * ((O^-1)(D_i * D_o)));
```

With the partial order relation, it is possible to compute the set of operations to execute in order to generate the data required by the operations that will generate the simulation output:

```
K_x := (K_t + (Order_relation^-1)(K_t));
```

Finally, with K_x and the partial order relation, we can generate a valid scheduling:

```
codegen K_x using Order_relation;
```

As for the memory layout, we can naively allocate the entire logical array or rely on Polyhedral techniques to reuse memory, such as the one developed by *Isoard et al.* [Darte et al., 2016] and presented in Section 5.1.

4.4 Conclusion and discussions

In this chapter, we have classified the concerns of a simulation code as related to algorithm or optimization choices. From this classification, we have proposed the InKS programming model with the aim of separating these two concerns by depending upon two concern dedicated languages. The first one, $\text{InKS}_{\mathsf{PIA}}$, is in charge of the full expression of the simulation algorithm; that is, parts of code that will never change no matter the architecture you are running the code on; in other words, the platform independent code. The second language, $\text{InKS}_{\mathsf{PSO}}$, takes care of the optimization part; namely the code that is not essential to the application correctness but responsible for its good performance; in other words, the platform dependent code. Although the algorithm expression is free, the optimization choices are specified in accordance with its algorithm. Hence, for a given program, the simulation algorithm remains identical while distinct versions of optimization choices, typically one per architecture, can be written concisely, each reusing pieces of information gathered from the simulation algorithm. Finally, the InKS model provides a InKS compiler, InKS_{C}, responsible for the translation from $\text{InKS}_{\mathsf{PIA}}$ and $\text{InKS}_{\mathsf{PSO}}$ code to a valid *C++* application. After the InKS model description, we have

presented the InKS$_{\text{PIA}}$ language as well as the main pieces of information contained in an InKS$_{\text{PIA}}$ program in the last part of this chapter.

As a first remark, we can note that, even though there are many differences between the InKS programming model and other approaches, the InKS$_{\text{PIA}}$ language shares similarities with others *DSLs*. For instance, the PTG DSL of *PaRSEC* [Danalis et al., 2014] is also a declarative language of operations that is scheduled using a minimal set of data dependencies. However, while InKS$_{\text{PIA}}$ proposes to describe the algorithm with no concern for performance, PTG counts on users to describe optimization in these operations, which are not necessarily small, and relies on users to fix memory layouts and on a compiler to derive an efficient parallel scheduling. Another language is close to InKS$_{\text{PIA}}$: *PIPES* [Kong et al., 2016]. It proposes a language to describe the algorithm in terms of operations with data dependencies and logical arrays, targeting the Polyhedral model; however, it chose a completely different path for the expression of optimization choices. Firstly, in the algorithm languages, users can express hints and optimizations which break the separation of concerns, but more importantly, all optimization choices are taken into account by a compiler which relies on Polyhedral optimization techniques. As discussed in Section 3.4, such automatic tools are often highly efficient in a lot of situations but will not provide the best performance in cases that are clearly not rare in numerical simulation. For instance, *PIPES* relies on *Barvinok* [Verdoolaege et al., 2007] to compute an efficient memory layouts for each logical arrays. However, the complexity of the algorithm to compute the best possible layout is huge. For this reason, they count on heuristics to determine a good approximation; and as any approximation, it may work great in various situations and not so great in others.

Proposing InKS$_{\text{PIA}}$ as a *DSL* that targets the Polyhedral model exhibits several advantages. This strategy enables the InKS$_{\text{PIA}}$ language to express all classes of programs handled by this model, namely the *Static Control Parts* (*SCoP*) defined in Section 3.5, while making it compatible with various optimization transformations and static analysis. However, compared to general-purpose approaches, such as C or Fortran, it limits the supported class of programs. Indeed, *SCoP* are Parameterized Task Graph (PTG):

directed graph of tasks dependent on invariant integer parameters set at execution. Therefore, in these programs, dependencies known only at runtime, such as indirections, are not expressible. Similarly, since domain sizes are parameterized by constants known at runtime, mesh modifications during the algorithm implementation execution are not valid. Still, while PTGs cover many classes of programs, the InKS programming model is usable in conjunction with traditional programming models, such as C or Fortran. Although, the invariant integer parameters must be set at the algorithm implementation execution, it is possible to call multiple times, from a general C program, the algorithm with different parameters. Such strategy enables users to handle method such as mesh refinement outside of the InKS program while using the InKS model for a given mesh. In addition, it is possible to express computations that would not fit the Polyhedral model inside the InKS_{PIA} operation. For instance, one can use conditionals depending on logical arrays' values with C++ in the operations, even though it is not possible in InKS_{PIA}. This creates a compromise between the separation of concerns and the language capability. This compromise offers more complex operations but may use over-constraining the dependence, limiting the possible range of optimizations.

Instead of proposing a new language, InKS_{PIA}, it is conceivable to use a subset of the C language as the algorithm language. Then, translating tools, such as Clan [Bastoul et al., 2012] or pet [Verdoolaege and Grosser, 2012], can convert such C code to a Polyhedral representation. However, InKS_{PIA} offers three major advantages over such method. First, it enables a simple and fast analysis of the code. InKS_{PIA} and its compiler are designed to support the algorithm expression. On the contrary, C and C++ support much more instructions that translating tools may have issues to parse correctly. Secondly, we would have to specify which C set of codes is valid in the model. That may be complex for us to express such restrictions and detect them. Moreover, users may struggle to know what is exactly possible or not, compared to plain C. Finally, and maybe more importantly, C comes with underlying optimization. As we presented in Section 4.1, C comes with a total operation ordering and a specific memory layout. Although we can hope that translating tools are efficient enough to retrieve both

the partial ordering and the Dynamic Single Assignment form, it may be more complex for optimization specialists. Indeed, to implement a version of the optimization choices, they first need the minimal constraints on the ordering and the DSA form; that is, the algorithm. Hence, they would have to mentally undo optimizations carried by the C in order to get the algorithm. All that with the aim of knowing if that loop can be broken to expose vectorization or this array can be expanded to add data parallelism. For all these reasons, InKS$_{\mathsf{PIA}}$ is more suited as the algorithm language, compared to C. Besides, we believe that InKS$_{\mathsf{PIA}}$ is not more complex than the C language.

Another preoccupation relates to the C++ used in operation expression. Since, in practice, any C++ code is valid in an InKS operation, understanding the semantics of the operation expression is beyond the scope of the InKS$_{\mathsf{PIA}}$ compiler or the Polyhedral model knowability. While it enables users to use external libraries, such as *FFTW* for Fast Fourier Transform, these black-boxes limit the compiler comprehension of the algorithm, and therefore, its latter optimization. For instance, an operation that manages a Dirichlet boundary condition (copy of the boundary from a time-step to another) can be eliminated in some situation, depending on the underlying memory layout. With this limitation, such optimizations are not possible in the InKS programming model. However, a possibility to address this issue is to support predefined basic operation in the InKS$_{\mathsf{PIA}}$ language. Each comes with an implicit C++ implementation as well as a set of dependencies. These operations could include copies, but also additions, multiplications, etc. Thus, copy could be eliminated while dependencies would be implied by the basic operation: an addition or a multiplication would require two inputs and generate one output, a copy one of each, etc.

Chapter 5

Expressing optimization choices in the InKS programming model

Contents

The InKS programming model proposes to separate algorithmic and optimization concerns in numeric simulation codes in order to improve both developers productivity and codes readability, as well as easing application

portability. In the previous chapter, we have presented the INKS$_{\mathsf{PIA}}$ language which aims to express the simulation algorithm. As a description of the simulation, the algorithm left memory placement and operations execution order unspecified. The second step for the INKS programming model is to provide a way to express optimization using the pieces of information contained in the algorithmic part. A possibility would be to use plain C++, calling INKS$_{\mathsf{PIA}}$ operations and manually writing memory layouts and operations ordering. Although, this would offer very good performance, it would not rely on INKS$_{\mathsf{PIA}}$ information. Therefore, users would have to repeat all information, including iteration domains, already expressed in INKS$_{\mathsf{PIA}}$ *validity domains*, or making sure that constraints implied by the algorithm are respected.

To address this issue, we propose, in the INKS programming model, an optimization language for numerical simulation code: INKS$_{\mathsf{PSO}}$. It aims at handling general optimization choices while relying on pieces of information gathered from the INKS$_{\mathsf{PIA}}$ code. Proposing a language that can compete with C or Fortran in terms of performance while integrating optimization tools largely developed for these languages is a challenging project. Therefore, we first conceive a compiler which automatically generates valid memory layouts and operations scheduling, in accordance to a given INKS$_{\mathsf{PIA}}$ algorithm. In a second step, we propose two preliminary optimization languages that aim to wrap an existing optimization tool or strategy: INKS$_{\mathsf{Loop}}$ and INKS$_{\mathsf{XMP}}$. Compared to their original counterpart, these languages minimize code writing to the optimization part only as every other pieces of information are gathered from the INKS$_{\mathsf{PIA}}$ code. Here, the main objective is mainly to separate the concerns which, in itself, provides many advantages, discussed in Section 3.4, among which potential performance improvements. In a third step, by using a set of guidelines identified during the development of these two preliminary optimization languages, we propose a first implementation of the INKS$_{\mathsf{PSO}}$ language. This implementation is general enough to handle a wide variety of optimization choices while using INKS$_{\mathsf{PIA}}$ information for the missing parts.

In this chapter, we focus on optimization approaches offered by the INKS programming model to set optimization choices, in accordance with

the algorithm. In Section 5.1, we start by presenting a fully automatic compiler based method to generate valid optimization choices. After this first step, that enables us to prove that all minimal pieces of information are contained in the InKS$_{\mathsf{PIA}}$ algorithm, we focus on the optimization languages. Hence, Section 5.2 exposes and evaluates InKS$_{\mathsf{Loop}}$, a first approach to express optimization choices: a *DSL* that focuses on optimization strategies related to nested loops. Then, we show and evaluate a second approach in Section 5.3, InKS$_{\mathsf{XMP}}$, a wrapper to XcalableMP, a PGAS directive based programming model.

Developing and evaluating these first two optimization languages enable us to propose a more general approach to describe both memory layouts and computation scheduling. This approach, namely InKS$_{\mathsf{PSO}}$, is presented in Section 5.4. Its source-to-source compiler is then presented in Section 5.5. Finally, in Section 5.6, we conclude the chapter and discuss the limitations of the InKS$_{\mathsf{PSO}}$ language as well as the choices we made at the implementation of optimization languages. The InKS programming model compilers, as well as some examples, are available at `https://github.com/Armassarion/InKS`.

5.1 A fully compiler-based approach for memory layouts and operations scheduling

In Section 4.3, we have presented the InKS$_{\mathsf{PIA}}$ algorithmic language of the InKS programming model. It is designed to express algorithmic concerns only while leaving optimization choices unspecified. More specifically, in Section 4.3.3, we demonstrated that InKS$_{\mathsf{PIA}}$ code contained all information necessary to express valid optimization choices. Hence, in this Section, we present InKSc, a compiler for InKS$_{\mathsf{PIA}}$ code which derives from the algorithm a set of valid optimization choices; that is, memory layouts for each logical arrays as well as operations ordering.

As we discuss in Section 3.4, it seems difficult to an automatized approach to handle both a wide variety of programs and offer the best possible

performance for each. Such approach offers a trade-off between generality and performance. Although, this strategy delivered great performance in various situations, we do not target this objective. Thus, we do not propose an automatized approach which relies on smart methods to achieve, in certain situation, the best performance possible. Instead, our compiler achieves another objective: it enables users to test the algorithm; that is, one is able to verify that their algorithm generates correct output for small test cases. In addition, the compiler also translates INKS$_{\text{PIA}}$ operations to valid C++ functions, for their latter call in the scheduling set by the optimization choices, as illustrated on the *Operation files* in the Figure 4.2 (Page 86).

We rely on the *Integer Set Library* [Verdoolaege, 2010], *isl*, to encode the INKS$_{\text{PIA}}$ information in a polyhedral model representation. Roughly, this model offers to describe a program as a collection of statements gathered in polyhedra and a set of relations described which statement shall be executed before which other one. It is presented with more details in Section 3.5. Using this library, we are able to express, in the polyhedral model form, the two main objects presented in the Section 4.3.3; that is, K_x, the set of operation instances that must be executed to produce the result of the simulation, and $\leq$, the partial order relation on K_x. Hence, K_x is represented in the form of a union of polyhedra, each integer point representing an operation instance. $\leq$ is represented by a union of polyhedral relations, mapping instance of K_x that must be executed before other instances.

First, *isl* offers to generate a valid scheduling which traverses all integer points of the union of polyhedra representing K_x in accordance with $\leq$. In addition, it is possible to drive more precisely the scheduling by providing additional relations. One of them is the *proximity* relation, which encourages the *isl* schedule algorithm to map domain elements i to domain elements that should be scheduled either before i or as early as possible after i. Hence, we use as proximity relation an unbounded version of the partial order relation $\leq$. This leads to a scheduling that minimizes each data instance lifetime: whenever an operation produces a data instance, *isl* will try to schedule all operations that read this data instance as soon as

possible. The *isl* scheduling, Sch, is in the form of a scattering function. That is to say, a relation mapping an operation instance to a time vector, representing a virtual multi-dimensional space executed in the lexicographic order. In addition, *isl* also provides a function to generate an Abstract Syntax Tree (AST) from a scheduling. This AST, composed of loops, branches and operation calls, can be traverse to generate the C code.

Then, the second step is to handle the memory layouts. Although INKS$_{\text{PIA}}$ logical arrays are infinite in each dimension, only a subset of these cells are effectively accessed depending on K_x; that is, D_x, the set of data instance accessed by K_x at some point of the execution. Still, this subset is in the DSA form; that is, there is absolutely no memory reuse. Although, the compiler aims to generate a valid set of optimization choices for test purposes only, not reusing memory is not reasonable and drastically limits the set of executable test-cases. Hence, the compiler needs to implement a heuristic to generate memory layouts that reuse memory, and for that, it is essential to rely on the computation scheduling. Indeed, memory layouts and computation scheduling are tightly bound: a modification on one may make the second invalid. Therefore, we base our memory layouts generation on the scheduling computed by *isl*: Sch.

In [Darte et al., 2016], *Isoard et al.* propose a two-step method to produce memory layout based on a technique called lattice-based memory allocation [Darte et al., 2005]. This strategy is a generalization of different strategies based on affine mappings with foldings by modulo operations (called modular mappings), formalized with integer lattices. A modular mapping $(M, \vec{b})$, defined by a $p \times n$ integer matrix M and a positive integral vector $\vec{b}$ of dimension p, maps the coordinate $\vec{i}$ of a n-dimensional array to $\sigma(\vec{i}) = M\vec{i}\ mod\ \vec{b}$ (the modulo is applied component-wise) in a p-dimensional array of shape $\vec{b}$. Such mappings reuse memory and are compatible with the Polyhedral model. The first step uses Sch to compute the `Conflict Set`; *i.e.* the set of pairs of data instances that should not be mapped to the same location. Minimizing data instances lifetime, as we did in the scheduling computation, limits the size of the `Conflict Set`. The second step uses the `Conflict Set` to compute a modular mapping. Both steps are described in details in Isoard thesis [Isoard, 2016]. The modular mapping enables the

compiler to generate a memory layout that reuses memory cells, mapping multiple logical coordinates to the same memory location.

In addition to these two steps, we add a third one which greatly speeds up index computations at execution. This optimization is possible when $p = n$ and M is a permutation matrix, *i.e.* an identity matrix in which each row was permuted. In our situation, it appears that these conditions are fulfilled most of the time. In addition, this optimization relies on D_x, the subset of data instances, identified by a coordinate vector, that are effectively accessed during the simulation execution.

Considering all the data instance D_a of a particular logical array A of dimension $dim(A) \in \mathbb{N}$ accessed during the simulation $D_a \subseteq D_x$. We have $(M_a, \vec{b_a})$ the modular mapping of A such that, from a logical coordinate $\vec{l_a} \in D_a$, the physical coordinate being accessed is $\vec{p_a} = \sigma(\vec{l_a}) = M_a\vec{l_a}\ mod\ \vec{b_a} \mid \vec{l_a}$. Using the Polyhedral model, for each dimension i of A, it is possible to obtain the maximum of the i^{th} component of all logical coordinates, $\vec{l_a}$, of A being accessed, l_{mi}:

$$\forall i \in \{1..dim(A)\}, \exists l_{mi} \in \mathbb{N}, \forall \vec{l_a} \in D_a \mid l_{ai} \leq l_{mi}$$

Since M is a permutation matrix, the i^{th} component of the logical coordinate $\vec{l_a}$, l_{ai}, is simply permuted to a j^{th} component of the physical coordinate $\vec{p_a}$, p_{aj}, before applying the component-wise modulo operation, here b_{aj}:

$$p_{aj} = l_{ai}\ mod\ b_{aj}$$

If $l_{mi} \leq b_{aj}$, the modulo operation is not necessary. Indeed, for a considered dimension, if the maximal logical coordinate value is inferior or equal to the modulo we have to apply, the modulo is not necessary. This strategy enables us to remove the usage of modulo operations in dimensions where there is no memory reuse, typically all dimensions non-related to time.

5.1.1 Evaluation of the compiler-based approach

We now evaluate our compiler-based approach for optimization choices in the InKS programming model. This evaluation relies on two benchmarks. The first algorithm is the 3D finite difference heat equation solver, presented in Listing 4.10. The second one is the one dimensional version of this problem. For each benchmark, three versions of the optimization choices were used. The first version of each benchmark is simple and manually implemented. It uses a double-buffer method to store temperature values and relies on the most trivial nested loop possible; that is, in a *time then space* computation ordering. More specifically, it is similar to Listing 4.1 with the exception of executing InKS$_{\text{PIA}}$ operations instead of plain computations. The second versions of optimization choices are generated automatically by our compiler. Finally, the third version, only available for the 3D heat equation benchmark, is a manually modified version of the generated code. All codes were compiled with Intel 18 compiler (`icpc` with `-O3 -xHost -ip -ipo` compilation options) and executed on a single core of the *Irene* cluster ([TGCC, 2019], France), each computing node being equipped with 192 GB RAM and two *Xeon Platinium 8168* CPUs. Execution times are presented in Table 5.1.

	Time/iteration (second)				
Benchmark	**Manual**	**Compiler**	**Diff.**	**Compiler optimized**	**Diff.**
Heat 1D	1.89 (±19.36%)	1.72 (±17.44%)	9.59%	N/A	N/A
Heat 3D	3.24 (±12.52%)	55.63 (±0.42%)	-94.18%	2.81 (±9.61%)	15.12%

Table 5.1 – Comparison of manual C++ implementation and InKS compiler generated code. Time/iteration (median) and maximal relative change of the C++ and compiler-based code of the 3D and 1D heat equation solver, size (1024^3) with 15 time-steps (after 5 time-steps of warm up). The maximal relative change r to the median m of a set of n values V is defined as $r = \frac{m}{max_{i=0..n}(|m-V_i|)}$.

As shown in Table 5.1, for the 1D heat equation benchmark, the compiler generates a code efficient enough to match the performance of a naive manually written code. However, in the case of the 3D heat equation benchmark, the generated code is much slower than the version "manual" version. There are two major causes to this lack of performance. First, as mentioned

in Sections 4.3 and 4.4, operations are black-boxes for the compiler and in this situation, this is an issue. Indeed, the compiler is forced to schedule the `Boundary` operation even though it is a copy from the previous time-step. In the "manual" version, this operation is plainly eliminated. Secondly, *isl* schedules the loops in a correct but not optimal order. In this situation, the index used for the smaller stride of our array is not managed by the innermost loop, breaking possible vectorization. For the sake of explanation, Table 5.1 also presents a manually altered version of the 3D heat equation generated code (*Compiler optimized* column). In this version, we have reordered the loops and removed the execution of the `Boundary` operation. Combined with the `#pragma ivdep` directives automatically added by the compiler in the generated code, it effectively speeds-up the computation to the point it improves the manual version. Also, note that all three versions have the same memory footprint. Indeed, *Isoard* algorithm and modular mapping lead to a double-buffer memory layout, equivalent to the one implemented manually. In the end, although the automatic version is not as efficient as a manually written one, it is generated quickly from the InKS$_{\text{PIA}}$ algorithm. Indeed, the compilation takes seconds in this case and up to a few minutes for very complex cases (*e.g.* tens of operations on eight-dimensional logical arrays), mostly because of the transitive closure computation (*c.f.* Section 4.3.3). Moreover, this automatically generated code assures the correct results. In comparison, errors can be introduced in the manual version from the InKS$_{\text{PIA}}$ code which complicates the debugging: is the algorithm wrong or is it the manual implementation? Having a completely automatized version guarantees that all errors necessarily come from the algorithm. Moreover, the generated code is fast enough for test purposes.

In summary, using *isl*, modular mappings and our heuristic, the compiler is able to generate a valid scheduling as well as reasonable memory layouts, both in terms of memory reuse and indexes computation efficiency. Although, it fulfills its objective, *i.e.* generating a valid code for test purposes; such approach is not efficient enough for real application runs. The heuristic for the scheduling is trivial while memory layouts heavily rely on inefficient modulo operations. Instead, once the algorithm is validated by

the compiler-based approach, numerical simulation codes require performance. Therefore, optimization must rely on either a smarter compiler, enhanced with more complex strategies, or a language dedicated to optimization expression. As discussed in Section 3.4, we believe that although the smart compiler approach gives tremendous results in very specific domains, it is not adapted to the optimization of arbitrary complex applications.

For this reason, in the next sections, we focus on optimization languages that offer specialists the ability to express themselves the set of optimization choices for an application. Using these languages, experts only specify optimizations while a compiler retrieve missing pieces of information from INKS$_{\text{PIA}}$ code. This strategy offers a true separation of concerns by limiting the aspects to repeat in the optimization part.

5.2 InKS$_{\text{Loop}}$: a *DSL* for nested loops description and optimization

Implementing a complete optimization language for the INKS programming model is a long-term objective. For this reason, we focused on various optimization languages, each offering to describe a specific set of optimization choices. The global objective is to highlight guidelines for the latter construction of the INKS optimization language. In this Section, we present one of them: INKS$_{\text{Loop}}$, an optimization *DSL* to describe the order of nested loops surrounding an INKS$_{\text{PIA}}$ operation.

INKS$_{\text{Loop}}$ offers to manually specify tightly loop nests, *i.e.* loop nests containing instructions only in the innermost loop, and more specifically, a single INKS$_{\text{PIA}}$ operation. The INKS$_{\text{Loop}}$ compiler generates a C++ function containing the nested loops. In addition, for all other optimization choices (*e.g.* memory layouts, non tightly nested loops), plain C++ is usable in combination with INKS$_{\text{Loop}}$. An INKS$_{\text{Loop}}$ program is roughly composed of four parts. Firstly, one defines the INKS$_{\text{PIA}}$ operation at the core of the nested loops. Secondly, one sets the iteration domain of the surrounding loops. Thirdly, the user declares an order between each iteration

indexes. The final step consists of applying optimization transformations to the nested loops.

Listing 5.1 illustrates the InKS$_{\text{Loop}}$ usage on the 3D heat equation solver, which algorithm is presented in Listing 4.10 (Page 94) using InKS$_{\text{PIA}}$. As a reminder, the `Inner` operation defined in the InKS$_{\text{PIA}}$ code is shown again in the InKS$_{\text{Loop}}$ code. Firstly, the `loop` keyword introduces a nest optimization with a name, the list of parameters from the algorithm on which the loop bounds depend and a reference to the operation at the center of the nested loops. On Listing 5.1, we call the nested loops `stencil_loops`, parameterize it by the size of each dimension, *i.e.* `nx`, `ny`, `nz` and `nt`, and specify that it calls the `Inner` InKS$_{\text{PIA}}$ operation.

```
1  /*** PIA CODE
2
3  operation Inner(x, y, z, t) : (
4    double  H {...}
5  )
6
7  PIA CODE ***/
8
9  loop stencil_loops(t, nx, ny, nz, nt) : Inner {
10   Order : z, y, x;
11   Block : 128, 128;
12 }
13
14 loop stencil_loops_3DBlock(t, nx, ny, nz, nt) : Inner {
15   Order : z, y, x;
16   Block : 128, 128, 128;
17 }
18
19 loop stencil_loops_no_block(t, nx, ny, nz, nt) : Inner {
20   Order : z, y, x;
21 }
```

Listing 5.1 – InKS$_{\text{Loop}}$ implementation of the 3D finite difference heat equation solver using three strategies a three loops nest, a 2D blocked three loops nest 2D and a 3D blocked three loops nest.

Secondly, loop bounds correspond to the *validity domain* of the InKS$_{\text{PIA}}$ operation; and therefore, are automatically extracted from the InKS$_{\text{PIA}}$ code. However, these bounds can be restricted in two ways. Either with the `Set` keyword, which defines a new iteration domain using the syntax used for `validity domains`, or by fixing some of the integer parameters of the InKS$_{\text{PIA}}$ operation. Listing 5.1 presents the usage of the second strategy. It fixes `t` by adding it to the `stencil_loops` parameters. It means that the

```
1 template <typename T0>
2 void stencil_loops_no_block(T0& H, int t, int nx, int ny, int nz, int nt){
3   if (t >= 1 && nt >= t + 1)
4     for (int c1 = 1; c1 < nz - 1; c1 += 1)
5       for (int c2 = 1; c2 < ny - 1; c2 += 1)
6         #pragma ivdep
7         for (int c3 = 1; c3 < nx - 1; c3 += 1)
8           Inner(H, fac, c3, c2, c1, t);
9 }
```

Listing 5.2 – C++ generated code for a three loops nest specified in INKS$_{\text{Loop}}$ (Line 19 to 21 of Listing 5.1).

iteration domain corresponds to the *validity domain* in which the value of `t` is fixed by the caller of `stencil_loops`.

Thirdly, the `Order` keyword specifies the iteration order on the dimensions named according to the INKS$_{\text{PIA}}$ code, minus the ones set by the method previously described. In Listing 5.1, loops are ordered following the `z`, `y` and finally the `x` indexes.

Finally, INKS$_{\text{Loop}}$ comes with two keywords to apply some common optimization techniques. First, the `Block` keyword enables the user to implement blocking. It takes as parameters the size of block for the loops starting from the innermost one. If there are fewer block sizes than loops, the remaining loops are not blocked. This strategy is presented in Listing 5.1 on Line 11: the two innermost loops are blocked by a size of 128. We also test the same order with a 3D block, presented Line 16 of the same Listing. According to the INKS$_{\text{PIA}}$ code, the loops iterate over an array containing double-precision floating-point values. Hence, the resulting block is of size $128\,kB$ and fits in the L2 cache of most modern processors. Then, the `Buffer` keyword (not present in the Listing) proposes to copy array data in a local buffer before the innermost loop, containing the computations. This buffer is then used during the computations and copied back to the array after, at the end of this innermost loop. In the case of computations performed on a large array, in which each data access is far from contiguous, this strategy ensures data locality and enables vectorization possibilities. Listing 5.3 presents a manual implementation of this optimization strategy.

During both the order and optimization phases, the compiler uses data dependencies from the INKS$_{\text{PIA}}$ code to check the validity of the loops order.

```
/*** Non optimized version ***/
double A[Z][Y][X];
double A_res[Z][Y][X];

for(int k=0; k<Z; k++)
  for(int j=0; j<Y; j++)
    for(int i=0; i<X; i++)
      //Work on non contiguous elements
      // => no vectorization + bad cache use
      A_res[k][j][i] = A[k-1][j][i] + A[k][j][i] + A[k+1][j][i];

/*** Optimized version ***/
double A[Z][Y][X];
//Memory footprint reduced
double buffer_in[Z];
double buffer_out[Z];

for(int j=0; j<Y; j++)
  for(int i=0; i<X; i++){
    //Copy A -> Buffer
    for(int k=0; k<Z; k++)
      buffer_in[k] = A[k][j][i];
    //Work on small, contiguous buffers
    // => vectorization + better cache use
    for(int k=0; k<Z; k++)
      buffer_out[k] = buffer_in[k-1] + buffer_in[k] + buffer_in[k+1];
    //Copy buffer => A
    for(int k=0; k<Z; k++)
      A[k][j][i] = buffer_out[k];
  }
```

Listing 5.3 – Two versions of a 1D stencil computation applied on the last dimension of a 3D array.

It also uses this information to generate C++ directives whenever possible, typically related to vectorization, to help the future C++ compiler in its automatic optimization stage. Listing 5.2 shows the C++ generated code for the `stencil_loops_no_block` InKS$_{\text{Loop}}$ code of the Listing 5.1.

5.2.1 InKS$_{\text{Loop}}$ algorithm

In Section 5.1, we have presented *isl* and its scheduling feature. As a reminder, from a set of operations to perform and a partial order between them, *isl* can generate a scattering function. That is, a relation mapping an operation instance to a time vector, representing a virtual multi-dimensional space executed in the lexicographic order. InKS$_{\text{Loop}}$ is actually a DSL to propose such scattering function. Indeed, firstly, the iteration domain and order define a first scattering function. Secondly, we apply polyhedral

transformations to this function in order to obtain a new one which adds the blocking and the call to the copies. For instance, the `loop` and `Order` keywords of the INKS$_{\texttt{Loop}}$ code presented in Listing 5.1 imply the following scattering function, written in *isl* form:

$$\begin{aligned}[X,Y,Z,T,t] \rightarrow \{Inner[i,j,k] \rightarrow [k,j,i] : \\ 0 < i < X-1 \ and \\ 0 < j < Y-1 \ and \\ 0 < k < Z-1 \\ \}\end{aligned} \tag{5.1}$$

Then, the `Block` keyword applies the following transformation to the scattering function:

$$\begin{aligned}\{[k,j,i] \rightarrow [k,jj,ii,j,i] : \\ ii \leq i < ii+128 \ and \\ ii \ mod \ 128 = 0 \ and \\ jj \leq j < jj+128 \ and \\ jj \ mod \ 128 = 0 \\ \}\end{aligned}$$

Therefore, the resulting scattering function is the composition of the original scattering function and the blocking transformation, that is:

$$\begin{aligned}\{[X,Y,Z,T,t] \rightarrow \{Inner[i,j,k] \rightarrow [k,jj,ii,j,i] : \\ ii \leq i < ii+128 \ and \\ ii \ mod \ 128 = 0 \ and \\ jj \leq j < jj+128 \ and \\ jj \ mod \ 128 = 0 \ and \\ 0 < i < X-1 \ and \\ 0 < j < Y-1 \ and \\ 0 < k < Z-1 \\ \}\end{aligned}$$

The last keyword, `Buffer`, adds copies and works differently. A copy is a statement, similarly to an operation. Therefore, the goal is to enhance the scattering function Sct mapping operation instances O of dimension $dim(o)$ to time vectors T of dimension $dim(t)$, with two more scattering functions. One for the copy from the array to the local buffer, *CopyIn*, and the copy back, *CopyOut*.

$$O : Z^{dim(o)}$$

$$T, T_{ci}, T_{co} : Z^{dim(t)}$$

$$Sct : O \to T$$

$$CopyIn : Z^{dim(o)-1} \to T_{ci}$$

$$CopyOut : Z^{dim(o)-1} \to T_{co}$$

CopyIn shall be scheduled just before the operation whereas *CopyOut* just after. That means the time vectors of *CopyIn* shall be lexicographically lower than the ones of the operation. However, this difference shall appear only for the last dimension of the time vectors. In other words, the time vector of the operation and *CopyIn* must be equal except for the last dimension, in which the *CopyIn* time vector must be inferior. The opposite for *CopyOut*. Mathematically, all four conditions shall be respected:

$$\forall \vec{t} \in T \begin{cases} \forall \vec{t_{ci}} \in T_{ci} \begin{cases} t_{cij} = t_j, \forall j \in \{1..dim(t)-1\} \\ \\ j = dim(t),\ t_{cij} < t_j \end{cases} \\ \forall \vec{t_{co}} \in T_{co} \begin{cases} t_{coj} = t_j, \forall j \in \{1..dim(t)-1\} \\ \\ j = dim(t),\ t_{coj} > t_j \end{cases} \end{cases}$$

In order to respect these conditions, we have to get the set of time vectors of O, $T_{min} \subset T$, such that there are no time vector $\vec{t} \in T$ such that $\vec{t}$ is lexicographically equal to $t\vec{1} \in T_{min}$ for the first $dim(t) - 1$ component while being lexicographically less for the last component. Similarly, the set

of time vectors of O, T_{max}, can be computed by modifying the last condition: the last component of $\vec{t}$ shall not be lexicographically greater than the one of $\vec{t1}$. Note that in our situation, all time vectors are necessarily different because InKS$_{\mathsf{Loop}}$ specifies a total order between operation instances. Mathematically, we can write:

$$T_{min} = \left\{ \vec{t_i} \in T \mid \nexists \vec{t_j} \in T, i \neq j \begin{cases} t_{ik} = t_{jk} \ \forall k \in \{1..dim(t) - 1\} \\ t_{ik} > t_{jk}, \ k = dim(t) \end{cases} \right\}$$

$$T_{max} = \left\{ \vec{t_i} \in T \mid \nexists \vec{t_j} \in T, i \neq j \begin{cases} t_{ik} = t_{jk} \ \forall k \in \{1..dim(t) - 1\} \\ t_{ik} < t_{jk}, \ k = dim(t) \end{cases} \right\}$$

Finally, we can compute T_{ci} and T_{co} by subtracting, respectively adding, one at the last component of each time vectors.

For the sake of explanation, let's consider the code presented in Listing 5.1 and the scattering function presented in Equation (5.1), mapping each instance (i, j, k) of $Inner$ to the logical time $T = (k, j, i)$. We want to perform a *CopyIn* right before the most-inner `i` loop. That is, we want the relation mapping each instance (j, k) of *CopyIn* to a time vector $(k_c, j_c, i_c) \mid \forall (k, j, i) \in T, \ k = k_c \wedge j = j_c \wedge i > i_c$. To schedule *CopyIn*, we first need the minimal time vectors of the operation, for any given `j` and `k` values. To get these time vectors, we must first obtain all time vectors at which the operation is scheduled: Dom. That is, the current domain of the scattering function presented in Equation (5.1). Secondly, we compute only the set of values of the `i` dimension of Dom. This can be done by using such relation:

$$Only\_i := \{Inner[i, j, k] \rightarrow [i]\}$$

Then, *isl* proposes lexicographic functions to obtain the minimal value of $Only\_i(Dom)$ Using the inverse relation $Only\_i^{-1}$ enables us to get the operation instance with the minimal time vector value, for any given `j` and `k` values. Applying the scattering function gives us the minimal time vectors of these instances.

The second step is to define the *Before* relation that maps a time vector $t1$ to another time vector $t2$ such that the first components of $t1$ and $t2$ are equal and the last one of $t2$ is inferior to the last one of $t1$. Applying such relations to the minimal time vectors of the *Inner* operations gives us the time vector of the *CopyIn* operation. Such computations can be done with the *iscc* calculator [Verdoolaege, 2014] with the following script:

```
#Inputs
Scattering := [X, Y, Z] -> {Inner[i, j, k] -> [k, j, i] :
     0<i<X-1 and 0<j<Y-1 and 0<k<Z-1};
Dom := dom Scattering;

#Relations
Only_i := {Inner[i, j, k] -> [i]};
Before := {[k, j, i] -> [k, j, i-1]};
CopyIn := {CopyIn[j, k] -> [k, j, i]};

#Computations
min_i := lexmin(Only_i(Dom));
min_s := (Only_i^-1)(min_i);
min_time := Scattering(min_s);
CopyIn := CopyIn ->* (Before(min_time));

#Result
Scattering := Scattering + CopyIn;
codegen Scattering; # <- prints the code
```

5.2.2 Evaluation of the InKS$_{\text{Loop}}$ approach

In order to evaluate the InKS$_{\text{Loop}}$ approach in terms of performance and usability, we have first implemented, using InKS$_{\text{PIA}}$, the algorithm of the 3D heat equation solved by finite difference method, presented in Listing 4.10, and two kernels of the well-known NAS benchmark [Bailey et al., 1991], `IS` and `MG`, using the C++ version as reference [Griebler et al., 2018]. We have then implemented an InKS$_{\text{Loop}}$ version of the optimization choices. For the heat equation, that corresponds to the implementation of the blocked loop nest presented in Listing 5.1, using the plain C++ version as reference, presented in Listing 4.2. As for the `IS` and `MG` NAS kernels, we have implemented parts of the optimization choices that involved nested loop, as it is the choices handled by InKS$_{\text{Loop}}$. All other optimization choices, including memory layouts were implemented using plain C++. Finally, we

	Execution time (second)		
Benchmark	**C++**	**InKS$_{\text{Loop}}$**	**Diff.**
NAS/IS	1.91 (±0.00%)	1.91 (±0.01%)	0.00%
NAS/MG	4.61 (±0.02%)	4.32 (±0.02%)	6.84%
Heat	39.51 (±0.78%)	39.49 (±0.53%)	0.05%

Table 5.2 – Comparison of C++ and INKS$_{\text{Loop}}$ implementations. Execution time of the C++ and INKS$_{\text{Loop}}$ implementations of 2 kernels of the sequential NAS benchmark (IS & MG), class B - Time/iteration of the 3D heat equation (7-point stencil) using a 2D blocking method, size (1024^3) with 10 time-steps. Median and maximal relative change of 10 executions.

compared the execution times between INKS$_{\text{Loop}}$ versions and plain C++ versions of optimization choices. *C.f.* 5.1.1 for compiler and architecture details. Results are presented in Table 5.2.

In terms of specification of optimization choices, INKS$_{\text{Loop}}$ enables the developer to specify optimization choices only while algorithmic information is extracted from INKS$_{\text{PIA}}$ code. This is illustrated by Listing 5.1 presenting the INKS$_{\text{Loop}}$ implementation of the loop nest around the stencil computation. Loop bounds are automatically retrieved from the algorithm while it enables the check of the specified order. Moreover, developers can easily test different optimization choices that would be tedious in plain C++. This is what we did with the 3D Heat solver based on the double-buffer technique. As shown in Listing 5.1, using INKS$_{\text{Loop}}$, we have implemented 3 versions of the loops: one with a naive loop nest without blocking, one based on a 2D cache blocking and a last one using a 3D cache blocking by modifying almost nothing in the code. Since INKS$_{\text{Loop}}$ is usable with C++, this optimization language does not restrict the expressible optimization choices: one can still implement optimizations not handled by INKS$_{\text{Loop}}$ in C++. The approach enables optimization specialists to focus on their specialty which make the development easier while offering a static check to verify that the optimization is valid with the algorithm. However, it handles only loop nests that call a single INKS$_{\text{PIA}}$ operation. Because of this limitation, it was not possible to implement non perfect nested loops (*i.e.* loops with a single operation) such as the one in the `EP` kernel of the NAS benchmark [Bailey et al., 1991]. Indeed, as shown on Listing 5.4, it

relies on a loop nest that performs multiple operations. Regarding performance, the InKS$_{\text{Loop}}$ approach has no overhead in comparison to a plain C++ approach. InKS$_{\text{Loop}}$ implementations of the optimizations choices of the InKS$_{\text{PIA}}$ code achieve performance similar to the reference as shown in Table 5.2.

```
1  for ( i = 1; i <= NK; i++) {
2    x1 = 2.0 * x[2*i-1] - 1.0;
3    x2 = 2.0 * x[2*i] - 1.0;
4    t1 = pow2(x1) + pow2(x2);
5    if (t1 <= 1.0) {
6      t2 = sqrt(-2.0 * log(t1) / t1);
7      t3 = (x1 * t2);     /* Xi */
8      t4 = (x2 * t2);     /* Yi */
9      l = max(fabs(t3), fabs(t4));
10     qq[l] += 1.0;     /* counts */
11     sx = sx + t3;     /* sum of Xi */
12     sy = sy + t4;     /* sum of Yi */
13   }
14 }
```

Listing 5.4 – One of the computation intensive parts of the `EP` kernel of the NAS benchmark (code from [Griebler et al., 2018]).

InKS$_{\text{Loop}}$ is a preliminary test before a real optimization language implementation. The goal was to demonstrate the feasibility of an approach handling the specification of a simple set of optimizations while retrieving the algorithmic information in InKS$_{\text{PIA}}$ code. This DSL enables us to highlight a set of guidelines for the full definition of the InKS$_{\text{PSO}}$ language.

First, this definition must be based on the concepts that InKS$_{\text{PSO}}$ must express, i.e. the optimizations related to memory (allocations and layouts) and to computations scheduling. InKS$_{\text{Loop}}$ focus on computations scheduling by adding strong constraints on the order of the computations. In the InKS$_{\text{PSO}}$ full implementation, adding such constraints may be a good idea to schedule precisely a set of operations.

Secondly, in order to express these optimization concepts only, it must be bound to its algorithmic counterpart, express in InKS$_{\text{PIA}}$. In InKS$_{\text{Loop}}$, the `Order` keyword refers to the structuring variables of a computational operation defined in InKS$_{\text{PIA}}$. Although it is quite simple, this strategy is severely limited and prevents the call of multiple operations in the loop nest, since they do not share structuring variables. Hence, InKS$_{\text{PSO}}$ should come

with its own loop indices which then may refer to the operations structuring variables.

Finally, we must work on the means to express this operation ordering with the most generality. For this part, a declarative language such as INKS$_{\text{Loop}}$ may not be the best strategy: to handle the interaction between memory and computation, an imperative language could be easier.

5.3 InKS$_{\text{XMP}}$: an XMP wrapper for memory management on distributed architectures

Parallel concerns are essential facets of optimization in most numerical simulation codes. Hence, after experimenting loop nest specification in the previous section, we explore a second aspect of the optimization choices: the parallel concerns. In this Section, we present INKS$_{\text{XMP}}$, an optimization language that wraps *XcalableMP*, a tool that handles parallel consideration in distributed memory environment.

XcalableMP (XMP for short) is a directives based partitioned global address space (PGAS) language developed at the RIKEN Center for Computational Science in Japan [Lee and Sato, 2010]. It simplifies the handling of distributed memory by presenting it to the developer as a single global space. XMP provides a minimal set of directives to manage distributed memory parallelism, among which directives to distribute an array over a set of nodes or to manage inter-nodes communications. INKS$_{\text{XMP}}$ is a directive based language that offers an interface to two XMP concerns: domain decomposition and a specific type of halo exchanges. Compared to plain C and XMP, INKS$_{\text{XMP}}$ asks for optimization choices only while other information is retrieved from the INKS$_{\text{PIA}}$ code. A source-to-source compiler then replaces directives with C and XMP code. As illustrated on Lines 1 to 3 and Line 11 to 12 of Listing 5.6, plain C and XMP are usable in combination with INKS$_{\text{XMP}}$ for all other optimization choices. INKS$_{\text{XMP}}$ comes with two directives: the `inks decompose` directive supports domain decomposition while the `exchange` directive supports halo exchanges.

```
1  #pragma xmp nodes p3d[PROC_Z][PROC_Y][PROC_X]
2  #pragma xmp template t3d[:][:][:]
3  #pragma xmp distribute t3d[block][block][block] onto p3d
4
5  void inks_heat(double* InOut_Heat, int nx, int ny, int nz, int nt){
6
7    #pragma xmp template_fix[block][block][block] t3d[nz][ny][nx]
8    double Heat[2][nz][ny][nx]
9    #pragma xmp align Heat[*][z][y][x] with t3d[z][y][x]
10   #pragma xmp shadow Heat[0:0][1:1][1:1][1:1]
11   /*...*/
12 }
```

Listing 5.5 – Plain XMP implementation of the 3D finite difference heat equation solver.

```
1  #pragma xmp nodes p3d[PROC_Z][PROC_Y][PROC_X]
2  #pragma xmp template t3d[:][:][:]
3  #pragma xmp distribute t3d[block][block][block] onto p3d
4
5  void inks_heat(double* InOut_Heat, int nx, int ny, int nz, int nt){
6
7    #pragma xmp template_fix[block][block][block] t3d[nz][ny][nx]
8    #pragma inks decompose % Heat(4*2, 3, 2, 1) with t3d
9    #pragma xmp shadow Heat[0:0][1:1][1:1][1:1]
10   /*Plain C is usable*/
11   for(int i=0; i<10; i++)
12     printf("%d\n", i);
13 }
```

Listing 5.6 – INKS$_{\text{XMP}}$ implementation of the 3D finite difference heat equation solver.

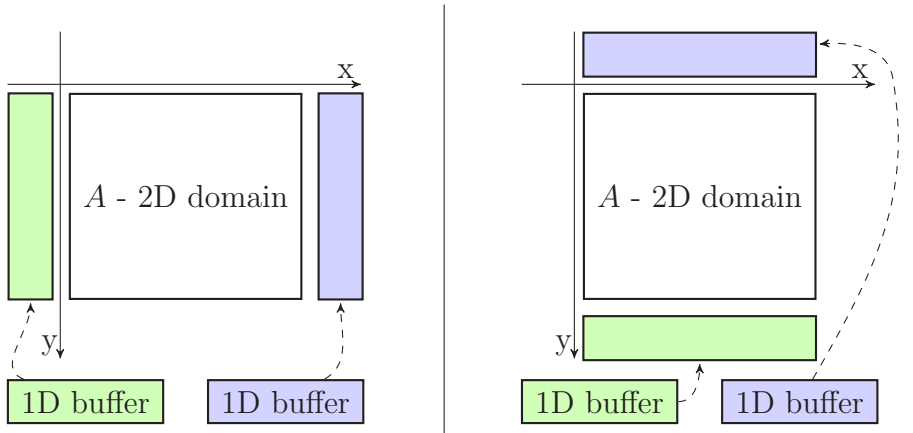
XMP proposes to allocate multidimensional arrays through two types of allocations: static and dynamic ones. Just as C, static allocations rely on multiple pairs of square brackets while dynamic allocations use an allocation function close to `malloc`. XMP static allocations are well-suited to most arrays; however, dynamic allocations are required for certain types of communications, such as the one used in the class of specific halo exchange we will present later. Once all allocations are described, XMP provides directives to allocate virtual arrays, named template and describe the decomposition of these virtual arrays over a set of nodes, namely an XMP topology. Finally, it provides a directive to map allocated arrays to these virtual arrays. The combination of the virtual decomposition and the physical to virtual mapping exposes the physical arrays decomposition over the XMP topology.

INKS$_{XMP}$ provides a directive that wraps XMP static and dynamic allocation of logical arrays described in the INKS$_{PIA}$ algorithm. The domain size is extracted from INKS$_{PIA}$ source and the user only has to specify its mapping onto memory. In INKS$_{PIA}$ code, there are no concerns for memory optimization such as dimension ordering or memory reuse. Therefore, INKS$_{XMP}$ supports dimension reordering and folding which consists in reusing the same memory address for subsequent indices in a given dimension. As in XMP, INKS$_{XMP}$ supports domain decomposition mapped onto an XMP topology. Listing 5.6 illustrates INKS$_{XMP}$ domain decomposition from the INKS$_{PIA}$ code presented in Listing 4.10 whereas Listing 5.5 presents the same domain decomposition expressed in plain XMP. As shown on Line 8, using INKS$_{XMP}$, the data type is retrieved automatically as well as the size of the array. Users focus on dimension reordering, expressed using integers. For instance, according to the algorithm, the first three dimensions of the `Heat` logical array are space dimensions, `x`, `y` and `z`, while the last one, `t`, is reserved for the time. In INKS$_{XMP}$, we reorder these dimensions to take advantage of the smaller stride for the space dimension: the fourth dimension of the algorithm, the time one, is placed on the left of the physical buffer; that is, the dimension with the biggest stride. Similarly, the `x` dimension is placed at the right, at the smallest stride dimension. In addition, despite the decomposition of the space dimension, we do not want to distribute the time dimension. Therefore, we replicate it by using the star, `'*'`, operator. Finally, we fold the time dimension: it is not necessary to hold values of all time-steps; a double-buffer strategy is enough. Therefore, we keep only two time-steps, expressed by the `2` following the star operator. On the contrary, in plain XMP, illustrated in Listing 5.5, users have to specify the dimension sizes manually as well as mapping the physical buffer to the virtual distributed array.

XMP also supports halo allocations and exchanges. It automatically allocates a larger array on each node, depending on the halo size provided by the user, and manages the index computation itself such that halo areas are completely transparent for the user. We call such halo areas *connected halo buffer*, as they are directly parts of the array. As an XMP domain can be specified as periodic, such halo areas can be handled accordingly.

On the contrary, another method to implement halo areas is to use *dynamic halo buffer*, *i.e.* to use independent buffers to store the halo values. Despite being less common, this strategy heavily reduces memory footprint for large multidimensional arrays when the halo region is read only in certain dimensions at a time. In particular, it gives great performance result in combination with the `Buffer` optimization strategy of the INKS$_{\text{Loop}}$ approach. For the sake of an example, let's consider a 2D domain $X \times Y$ named A and two stencil operators f_1 and f_2. A is decomposed in 4 perfect rectangles over 4 nodes, P_{11}, P_{12}, P_{21} and P_{22}. f_1 reads only values on the first dimension, x, of A whereas f_2 focuses on the second dimension, y. To perform the f_1 operation, each node needs a halo of size $X/2$. Then, for f_2, each needs a halo of size $Y/2$. With the connected halo buffer strategy, each node needs to allocate a halo region of $X/2 + Y/2$. However, to apply f_1, a halo area is necessary only to hold values of the x dimension. In the same way for f_2, a halo area may contain only values of the y dimension. Therefore, we can use the dynamic halo buffer and allocate a buffer of size $max(X/2, Y/2)$. This buffer shall be used as halo region for the x dimension during the f_1 operation and then be reused as halo region for the y dimension during the f_2 operation. Figure 5.1 illustrates the dynamic halo buffer strategy for one of the node on a 2D example. In some application, in which arrays are of dimension 5 or greater, such strategy can reduce memory footprint and make possible to run bigger cases. Unfortunately, as it is uncommon, this strategy is not directly supported by XMP and requires users to describe explicitly the allocations and communications for halo exchanges. In addition, the exchanged array shall be dynamically allocated.

INKS$_{\text{XMP}}$ supports the dynamic halo buffer pattern. Once a logical array has been dynamically allocated using the INKS$_{\text{XMP}}$ directive for that purpose, the user can specify which dimension of that array should be exchanged. In addition, one should add the halo size, the operation that will be executed after the exchange, or both. If the operation is specified, the INKS$_{\text{XMP}}$ compiler can automatically compute the halo size and if it was not provided, use this size for the allocations, or verify that the specified halo size is large enough. This calculation is made using the INKS$_{\text{PIA}}$ operation data inputs and outputs. Finally, the INKS$_{\text{XMP}}$ compiler generates the C



(a) Halo buffers in 1st dimension (b) Halo buffers in 2nd dimension

Figure 5.1 – Dynamic halo exchange representation on a 2D domain. The blue and green buffers of (a) are reused in (b).

and XMP code that perform the actual distributed halo allocations and exchanges.

5.3.1 Evaluation of the InKS$_{XMP}$ approach

To evaluate the InKS$_{XMP}$ approach in terms of performance and usability, we have first implemented, using InKS$_{PIA}$, the algorithm of the 3D heat equation solved by finite difference method, presented in Listing 4.10, and one kernel of the well-known NAS benchmark [Bailey et al., 1991], `EP`, using the C++ version as reference [Griebler et al., 2018]. We have then implemented an InKS$_{XMP}$ version of the optimization choices. For the heat equation, that corresponds to the domain decomposition presented in Listing 5.6. As for the `EP` NAS kernel, we have implemented a domain decomposition of all the buffers. All other optimization choices, including loop scheduling were implemented using plain C and XMP. Finally, we have compared the execution times between InKS$_{XMP}$ versions and plain C and XMP versions of optimization choices. All codes were compiled with the XMP compiler, *XMPCC*, (`-O3 -xHost -ip` compilation options) and executed on up to 4 nodes of the *Irene* cluster (*c.f.* 5.1.1 for architecture details). *XMPCC* was compiled with the Intel 18 compiler and based on IntelMPI 2018. Only one core per node was used, as to evaluate the remote communications. Results are presented in Table 5.3.

	Execution time (second)		
Benchmark	**C+XMP**	**InKS$_{\text{XMP}}$**	**Diff.**
NAS/EP	14.01 (±0.16%)	14.00 (±0.13%)	0.07%
Heat	422.21 (±4.46%)	421.93 (±4.65%)	0.07%

Table 5.3 – Comparison of C+XMP and InKS$_{\text{XMP}}$ implementations. Execution time of the C+XMP and InKS$_{\text{XMP}}$ implementations of the `EP` NAS kernel, class B - Time/iteration of the 3D heat equation (7-point stencil), size (1024^3) with 10 time-steps. Median and maximal relative change of 10 executions on 4 nodes (1 core per node).

In terms of specification of optimization choices, InKS$_{\text{XMP}}$ enables the developer to specify optimization choices only while algorithmic information is extracted from InKS$_{\text{PIA}}$ code. This is illustrated by Listing 5.6 presenting the InKS$_{\text{XMP}}$ 3D domain decomposition, in comparison to its plain XMP counterpart, presented in Listing 5.5. Both are equivalent, but the InKS$_{\text{XMP}}$ version expects only optimization choices parameters; that is, dimension reordering and folding as well as the distributed memory concerns which here corresponds to the non distribution of the time dimension. Hence, one can test another memory layout, such as a different dimensions ordering, by changing only a few parameters, while multiple directives must be modified in XMP. Note that since InKS$_{\text{XMP}}$ is a wrapper for XMP using InKS$_{\text{PIA}}$ to retrieve some information, it is usable in any codes that can be expressed using InKS$_{\text{PIA}}$ and optimized with XMP. Since InKS$_{\text{XMP}}$ is usable with C, it does not restrict the expressible optimization choices: one can still implement optimizations not handled by our wrapper using C and XMP. Moreover, operations such as halo size computation are automatized using the information gathered from InKS$_{\text{PIA}}$ code. Regarding performance, the InKS$_{\text{XMP}}$ approach has no overhead in comparison to a plain C and XMP approach. InKS$_{\text{XMP}}$ implementations of optimization choices of the InKS$_{\text{PIA}}$ code achieve performance similar to the reference as shown in Table 5.3. In the case of the heat equation solver, performance are rather terrible in both situations because of our XMP misuse. Current implementation of XMP supports only the C language while we rely on C++ to handle most of the memory accesses. For technical reasons, data access are managed in a way that is not intended by XMP, which

negatively impacts the performance. In addition, INKS$_{\mathsf{XMP}}$ and INKS$_{\mathsf{Loop}}$ are not compatible, as the latter uses C++. Although INKS$_{\mathsf{XMP}}$ and plain XMP achieve similar performance, our language comes with limitations. It offers distributed memory allocations but no control on the access to this memory, letting optimization specialists in charge of the good use of this memory.

Just as INKS$_{\mathsf{Loop}}$, INKS$_{\mathsf{XMP}}$ is a preliminary test before a full optimization language implementation. The goal was to propose a way to express a simple set of optimizations and retrieving the algorithmic information in INKS$_{\mathsf{PIA}}$ code. Compared to INKS$_{\mathsf{Loop}}$, which focuses on computation ordering, INKS$_{\mathsf{XMP}}$ targets memory and enables us to highlight a different set of guidelines for the full definition of the INKS$_{\mathsf{PSO}}$ language.

First, with INKS$_{\mathsf{XMP}}$ we have tried some tests on the memory aspects, providing a directive to allocate a logical array and to reorder and fold its dimensions. As mentioned earlier, once this memory is allocated, the mapping of values to the memory is left to the user. Proposing a way to reorder and fold the logical dimensions while specifying how memory cells correspond to logical values seems like an efficient strategy to handle the memory layout concern in INKS$_{\mathsf{PSO}}$.

Then, the domain decomposition directives make explicit reference to logical arrays described in INKS$_{\mathsf{PIA}}$, making possible the computation of the size of each dimension and the halo size depending on the data being accessed. It is clear that, in INKS$_{\mathsf{PSO}}$, memory layout shall refer to the logical arrays described in the algorithm. Such strategy makes possible the use of INKS$_{\mathsf{PIA}}$ information to verify that a memory layout is in accordance with the following computations or even to automatically guess these computations.

Finally, we must work on the means to express these optimization concepts with the most generality. For this part, we think that working with existing tools is probably mandatory for most complex strategy, such as XMP for the domain decomposition. Therefore, supporting at least the most common optimization tools, such as OpenMP and MPI, is a requirement to propose an optimization language usable in most types of codes.

5.4 InKS$_{\mathsf{PSO}}$: a language to express general optimization choices

With the InKS$_{\mathsf{PIA}}$ language of the InKS programming model, it is possible to express all aspects of simulation algorithm. As a description of the simulation, the algorithm leaves memory placement and operations execution order unspecified. In this chapter, we have proposed three approaches to set such choices: the InKS$_{\mathsf{C}}$ compiler and both InKS$_{\mathsf{Loop}}$ and InKS$_{\mathsf{XMP}}$ optimization languages. However, one may notice that the first approach is fully automatic and not designed to reach the best possible performance, while the two later target a very specific kind of optimization choices. For the InKS programming model to be considered an interesting approach, it is essential to come up with means to express all optimization choices; that is, both memory placement and operations ordering. Compared to the previous optimization approaches of the InKS model, it shall be general enough to express an entire program while being capable of reaching the best possible performance. However, these previous approaches bore guidelines for the specification of this general optimization language. In particular, the language shall be imperative and focus on memory layouts while offering instructions that constraint the operation ordering. Hence, this section is dedicated to InKS$_{\mathsf{PSO}}$, an optimization language of the InKS programming model with the aim of achieving such goals.

InKS$_{\mathsf{PSO}}$ is an imperative optimization language of the InKS programming model with the aim of expressing both memory layouts and operation scheduling by reusing the pieces of information contained in the algorithm. Just as InKS$_{\mathsf{PIA}}$, InKS$_{\mathsf{PSO}}$ optimization choices expression is mostly based on the Polyhedral model. An InKS$_{\mathsf{PSO}}$ code is always derived from an InKS$_{\mathsf{PIA}}$ algorithm specification and respects its statements, especially in terms of ordering constraints and execution results. If the InKS$_{\mathsf{PSO}}$ code does not respect its algorithm, the code is invalid and, thanks to the Polyhedral model, it can be detected.

In a typical InKS$_{\mathsf{PSO}}$ code, one first allocates multidimensional buffers. Then one defines links between the cells contained in these buffers and the cells in logical arrays described in InKS$_{\mathsf{PIA}}$. This step's objective is to

exhibit a correspondence between the logical and physical memory spaces. Computations are handled by requesting the update of a buffer content with the values of the logical array region mapped in it. Knowing the logical regions mapped in each buffer, the compiler can automatically extract from the INKS$_{PIA}$ algorithm the sequence of operations to execute for an update. Finally, one can reuse memory by redefining the links such that buffers' regions correspond to new logical arrays regions. An INKS$_{PSO}$ program is a sequence of those statements whose execution in order must result in up-to-date buffers containing the output of the algorithm. Indeed, as we have highlighted in Section 5.2.2, proposing an imperative language is more suited for our optimization approach.

To express these concepts, INKS$_{PSO}$ defines two sets of instructions. The first set of instructions (illustrated in Listing 5.7) is related to the memory layout management:

- `var` declares a buffer variable that may be allocated later.
- `alloc` allocates a contiguous multidimensional physical *buffer*. It takes as parameters the type and size of the buffer.
- `free` releases the memory space occupied by a buffer.
- `map` registers a mapping between cells from a physical buffer and cells from a logical array declared in INKS$_{PIA}$. `map` takes as parameter a coordinate mapping function.
- `unmap` unregisters a map from the set of defined ones.

```
1  var double buffer(2) //2D buffer declaration
2  alloc double buffer(2, X) //2D buffer allocation
3
4  //Link between "buffer" physical cells and "full_array" logical cells
5  map i=[0:X[ buffer(0, i) = full_array(i, 0)
6
7  /*...Computations...*/
8
9  /*Remap the same physical region to a new logical one*/
10 map i=[0:X[ buffer(0, i) = full_array(i, 1)
```

Listing 5.7 – Example of INKS$_{PSO}$ `alloc` and `map` instructions.

A logical coordinate can be mapped to multiple physical coordinates in order to store a single value in multiple locations. On the opposite, a physical coordinate can only be mapped to a single logical coordinate at a given point of execution that represents the value currently stored in this location. Hence, the `map` instruction implies an `unmap` of the physical region being mapped. A buffer that contains values but is not mapped to a logical array is considered empty. Remapping a buffer region makes the values it contains inaccessible. InKS$_{\text{PSO}}$ accepts any affine function for the map statement which makes it possible to use simple memory layouts like column or row major, but also much more convoluted ones (*e.g.* blocked mapping). Like in InKS$_{\text{XMP}}$, it is possible to fold or reorder any dimension.

Then, operations ordering is specified by one of the four variations of the `update <region> using <strategy>` construct. The `<region>` parameter defines the region where to update values and shall be a logical region. The `<strategy>` can be one of the following:

- `<operations[,...]>` updates the values in the specified region at the exclusion of any other using the specified operation procedures, all required dependencies must already be up-to-date in other buffers;
- `auto` is similar but updates any intermediate value required as long as it is already mapped in a buffer;
- `copy` expects a physical region and copies the values from one or more buffer that must be up-to-date and mapped to the same logical area;
- `for` introduces a loop where the sequence of intermediate statements to execute for the update is manually specified.

During an `update` statement, the values are written to the buffer most recently mapped while they are read from the buffer most recently mapped that is up-to-date. Hence, the operations execution order is driven by data accesses. Associated with the memory statements, this strategy enables users to identify easier how the vectorization will be implemented or how the cache will be used.

The `for` loop (illustrated in Listing 5.8) iterates over one or multiple indices. Each index accepts a fortran-like iteration domain specified using

the syntax `{start:end[:step]}`. In addition, each index can have options specified. Currently, INKS$_{PSO}$ provides two options:

- `OMP_FOR` keyword to request a parallel execution of the loop iterations, equivalent to a `#pragma omp parallel for`;
- `OMP_TASK` to put each iteration in an OpenMP task, executed by the available computing units.

In addition, one can specify the OpenMP schedule or collapse strategies to the `OMP_FOR` option. The support of the OpenMP Task paradigm in INKS$_{PSO}$ is a preliminary work, fruit of a collaborative effort involving Jérôme Richard, an expert in the task paradigm. This exploratory study is described with details in Appendix C.

```
1 update full_array(0:X, T-1) using for(t{1:T}){
2   //t is available in the region
3   map i=[0:X[ buffer(t%2, i) = full_array(i, t)
4   update full_array(i, t) using square
5 }
```

Listing 5.8 – Example of INKS$_{PSO}$ `for` instruction.

Regions are defined by a range in each of their dimensions. The boundaries of the range can linearly depend on the value of previous dimensions. The `dom` keyword is used to define named ranges that can then be reused whenever needed.

An INKS$_{PSO}$ function defines the signature of the generated C++ function (illustrated in Listing 5.9). Its parameter list contains integers that match the parameters of the implemented algorithm and buffers where to store the inputs and outputs. Each buffer can be marked as either allocated by the caller or in the function. The mapping of the input and output regions specified in the algorithm into these buffers is also specified.

5.4.1 Illustrative example

We now use the 3D heat equation solved using the finite difference method to illustrate the INKS$_{PSO}$ syntax and concepts. Listing 4.10 shows the INKS$_{PIA}$ implementation of the simulation algorithm while Listing 5.10

```
/* InKSpso interface */
my_simulation(
  /* Specify the allocation and map status of the
   * physical buffer "buffer" expected
   * at the end of the program
   */
  give double buffer(2, X)
    {out: i=[0:X[ H(T-1 % 2, i) = full_array(i, T-1)},
  X, T
){
  /* InKSpso code */
}

/* Equivalent C++ interface */
double* my_simulation(){
  double* buffer = new double[2*X];

  //*** Updating buffer such that it contains the full_array
  //*** values at the last time step

  return buffer;
}
```

Listing 5.9 – Example of InKS$_{\text{PSO}}$ interface and its C++ counterpart.

illustrates the InKS$_{\text{PSO}}$ implementation of the optimization choices. This InKS$_{\text{PSO}}$ code implements the same optimization choices as the C version in Listing 4.1, namely the double-buffer strategy.

```
inks_heat(
  give double H(2, nz, ny, nx) {out: iz=[0:ny[ iz=[0:ny[ ix=[0:nx[
      H(nt-1 mod 2, iz, iy, ix) = Heat(ix, iy, iz, nt-1)},
  nx, ny, nz, nt
){
  dom iz=[0:ny[ iz=[0:ny[ ix=[0:nx[

  alloc double H(2, nz, ny, nx)
  map H(0, iz, iy, ix) = Heat(ix, iy, iz, 0)

  /*Initialization*/
  update Heat(ix, iy, iz, 0) using Init

  /*Timeloop*/
  update Heat(ix, iy, iz, nt-1) using for(t{1:nt}){
    map H(t mod 2, iz, iy, ix) = Heat(ix, iy, iz, t)
    update Heat(ix, iy, iz, t) using Boundary, Inner
  }
}
```

Listing 5.10 – InKS$_{\text{PSO}}$ implementation of the 3D finite difference heat equation solver using a double-buffer strategy (InKS$_{\text{PIA}}$ algorithm is available in Listing 4.10).

The entry point of the simulation is defined in the InKS$_{\text{PIA}}$ code in the

first part of the `simulation` kernel. Therefore, the INKS$_{\mathsf{PSO}}$ code starts to express the memory layout of the output logical array as well as the property of the buffer (lines 2 and 3). Here, `give` means that the caller will pass a `H` pointer and expects that, by the end of the INKS$_{\mathsf{PSO}}$ code, it will be allocated. On Line 8, the `H` pointer is allocated and points to a buffer of size $2 \times nx \times ny \times nz$. This buffer is usable as a 2D array for the double-buffer strategy. The next line maps this buffer to a subset of the `Heat` logical array. More precisely, each cell identified by the coordinate `(0, iz, iy, ix)` in the `H` buffer corresponds to the cell `(ix, iy, iz, 0)` in the logical `Heat` array, for any `ix, iy, iz` between 0 and respectively `nx, ny, nz`. Then, it updates every value of the space dimension at the first time-step of the `Heat` logical array, using the `init` operation procedure. This area being mapped to the `H` buffer, as mentioned earlier, it is written during the computation. Finally, Line 15 illustrates the `update <region> for` construct with an index `t` iterating from `1` to `nt`. It introduces the time-loop containing the core of the computation: the 3-point stencil and boundaries computations. The region `Heat(ix, iy, iz, nt-1)` corresponds the logical region that will be up-to-date by the end of the loop. Inside the loop, the `t` index is used to:

1. put in place the double-buffer strategy by remapping `H` to `Heat` at each iteration (Line 15);
2. update all the values in the space dimension at the current time-step of the `Heat` logical array using the `Boundary` and `Inner` operation procedures (Line 16).

At the end of the program, the `H` buffer holds all values in the space dimension at the two last time-steps of the `Heat` logical arrays, as expected by the algorithm described in INKS$_{\mathsf{PIA}}$ in Listing 4.10.

Following the specification of a compiler to handle automatically the optimization choices (Section 5.1), the INKS$_{\mathsf{Loop}}$ language to express nested loops (Section 5.2) and the INKS$_{\mathsf{XMP}}$ language which takes care of domain decomposition (Section 5.3), this Section has finally presented the INKS$_{\mathsf{PSO}}$ language. Unlike others, INKS$_{\mathsf{PSO}}$ is capable of specifying general optimization choices, including memory layouts and operations scheduling. However,

just as the other optimization languages, it uses the InKS$_{\mathsf{PIA}}$ algorithm to ease the specification of the optimizations. The following part will describe the implementation of our InKS$_{\mathsf{PIA}}$ + InKS$_{\mathsf{PSO}}$ compiler.

5.5 The source-to-source InKS$_{\mathsf{PSO}}$ compiler

We now present the InKS$_{\mathsf{PSO}}$ source-to-source compiler. The InKS$_{\mathsf{PSO}}$ compiler combines an InKS$_{\mathsf{PIA}}$ algorithm with InKS$_{\mathsf{PSO}}$ optimizations to generate C++ code. The compilation process consists of four steps. First, it analyzes the InKS$_{\mathsf{PIA}}$ code and encodes it using the Polyhedral model. At this point, the compiler has access to the set of logical cells that will be accessed during the simulation as well as the set of operation instances, their inputs and outputs. This step is presented in Section 4.3.3. As a reminder, here are the main pieces of information available:

- the set of integer parameters $\mathcal{P}$;
- the set of logical arrays $\mathbb{A} = \{a_1, ..., a_n\}$;
- the dimension $dim(a) \in \mathbb{N}$ of each logical array $a \in \mathbb{A}$;
- the set of operations $\mathbb{K} = \{k_1, ..., k_n\}$;
- the set of data instances allocated at some point of the execution D_x;
- the set of operation instances executed at some point of the execution Kx;
- the relations mapping operation instances to their data dependencies (inputs I and outputs O);
- the order relation, mapping an operation instance to another operation instance that must be executed later, $\prec$.

In addition, we denote P_x the set of physical cells that can be allocated at any time along program execution (*i.e.* the computer physical memory).

A second step analyzes the InKS$_{\mathsf{PSO}}$ statements and checks that they are consistent with the InKS$_{\mathsf{PIA}}$ algorithm. For instance, it verifies that

the logical array specified in a `map` or `update` statement actually exists in the INKS$_{\mathsf{PIA}}$ code and that the number of dimensions specified effectively matches.

The third compilation phase obtains the required information for the code generation. We illustrate this phase with the INKS$_{\mathsf{PSO}}$ code presented on Listing 5.12, corresponding to the optimization choices of the INKS$_{\mathsf{PIA}}$ algorithm presented on Listing 5.11. During this stage, each statement is subdivided into one or more *"actions"*. The actions associated to the various INKS$_{\mathsf{PSO}}$ statements are as follows:

- each `alloc` statement allocates the provided physical region (*alloc action*),
- each `take` or `give` physical region property specified in the INKS$_{\mathsf{PSO}}$ interface is considered an `alloc` statement,
- each `free` statement both deallocates (*free action*) and unmaps (*unmap action*) the provided physical region,
- each `unmap` statement removes the mapping associated to the provided physical region (*unmap action*),
- each `map` statement first unmaps (*unmap action*) any pre-existing mapping of the mapped physical region, as specified in the INKS$_{\mathsf{PSO}}$ definition, then it applies the provided mapping (*map action*) on the provided region,
- each `update <operation...>` statement first the required reads logical region (*read action*) to write the provided logical region (*write action*). It reads logical regions depending on the operations being called (*i.e.* their input relation specified in INKS$_{\mathsf{PIA}}$).
- each `update auto` statement first reads the required logical region (*read action*) to write the provided logical region (*write action*). It reads and writes logical region, as well as all intermediate logical region that would be necessary, depending on the operations being called (*i.e.* their input relation specified in INKS$_{\mathsf{PIA}}$).

- each `update copy` statement first considers the logical region that is mapped to the provided physical region, then it reads physical regions that are up-to-date and mapped to the same logical region (*read action*), and finally, it writes this logical region to the provided physical region (*write action*),
- each `in` mapping property specified in the InKS$_{\text{PSO}}$ interface is considered a `map` statement and a *write action* of the provided logical region.

```
1  kernel inner(x, t) : (
2    double R {in: (x, t-1); (x-1, t-1); (x+1, t-1)| out: (x, t)}
3  )
4  #CODE(C)
5    R(x, t) = 0.5 * R(x, t-1) + 0.25 * (R(x-1, t-1)+R(x+1, t-1));
6  #END
7
8  kernel bord(x, t):(
9    double B {in: (x, t-1) | out:(x, t)}
10 )
11 #CODE (C)
12   B(x, t) = B(x, t-1);
13 #END
14
15 simulation inks_heat(X, T):(
16   double Heat(2) {in: (0:X, 0) | out: (0:X, T-1)}
17 )
18 #CODE (inks)
19   inner (1:X-1, 1:T)      : (Heat),
20   bord it=[1:T[ { (0, it); (X-1, it) } : (Heat)
21 #END
```

Listing 5.11 – InKS$_{\text{PIA}}$ implementation of the 1D heat equation solved with finite difference.

Each action is associated to a time vector $\vec{t} \in T$. A time vector is a vector of integers which represents the logical time at which a statement is executed. As explained in more details in Section 3.5.3, time vectors are sorted according to the lexicographic order. We set the vector equals to [1] as the smallest time vector associated to an action. This way, while they may differ in length two vectors can always be compared. However, because of implementation choices in *isl*, in order to compare two time vectors, they must have the same length. For this reason, we fill the shorter time vectors with zeros. We define the time vector $\vec{t}_{max} \in T$ such that $\nexists \vec{t} \in T \mid \vec{t}_{max} < \vec{t}$. Therefore, $\forall \vec{t} \in T$, $[1] \leq \vec{t} \leq \vec{t}_{max}$

```
1  //actions' time vector are presented in commentary
2  inks_heat(
3    take double H(2, X) //A->[1,0,0]
4      {in : x=[0:X[ H(0, x) = Heat(x, 0) //U->[2,0,0] + M->[3,0,0] + W->[4,0,0]
5        | out: x=[0:X[ H(T-1, x) = Heat(x, T-1)},
6    X, T
7  )
8  {
9    update Heat(0:X, T-1) using for(t{1:T})
10   {
11     map x=[0:X[ H(t mod 2, x) = Heat(x, t) //U->[5,t,1] + M->[5,t,2]
12     update Heat(0:X, t) using inner, bord //R->[5,t,3] + W->[5,t,4]
13   }
14   //tmax -> [6,0,0]
15 }
```

Listing 5.12 – INKS$_{\text{PSO}}$ implementation of the optimization choices (double-buffer) of the 1D heat equation (Algorithm on Listing 5.11).

The time vector $\vec{t_s} \in T$ of the action s is defined as follow:

1. a constant integer that identifies the action rank in the top level function block;
2. then, for each surrounding `for` loop:
   a) as many integers as the loop has indices, each integer taking the value of the corresponding indice,
   b) a constant integer that identifies the action rank in the loop block.

Each action is represented by a relation that maps the action time vector $\vec{t} \in T$ to the action effect:

- $A : T \to \mathcal{P}(P_x)$ associates to an action time vector, the physical buffer region this action allocates;
- $F : T \to \mathcal{P}(P_x)$ associates to an action time vector, the physical buffer region this action deallocates;
- $M : T \to \mathcal{P}(P_x \times D_x)$, maps an action time vector to a relation representing the mapping between physical and logical region this action introduces;

- $U : T \rightarrow \mathcal{P}(P_x)$ associates to an action time vector, the physical buffer region this action unmaps;
- $R : T \rightarrow \mathcal{P}(D_x)$ associates to an action time vector, the logical array region this action reads;
- $W : T \rightarrow \mathcal{P}(D_x)$ associates to an action time vector, the logical array region this action writes;

For example, Listing 5.12 shows how an action is associated to a time vector. The `take` property on Line 3 corresponds to an `alloc` statement, and therefore, an *alloc* action, associated to the time vector $[1, 0, 0]$. The `in` property on Line 4 is composed of three consecutive actions, *unmap*, *map* and *write*, respectively associated to the time vector $[2, 0, 0]$, $[3, 0, 0]$, $[4, 0, 0]$. Then, an `update for` statement is found: each action inside the loop will be associated to a time vector that contains the action numbering (4), the indice (`t`) and the action rank in the loop. Finally, Line 12, the `update` statement invokes the `inner` and `bord` operations which read values, producing a *read* action at date $[5, t, 3]$. Then, they write values, producing a *write* action at date $[5, t, 4]$.

In addition to the time vector, each action has an effect on the program. Hence, the *alloc* action on Line 3 considers allocated a physical buffer of size `X` $\times$ `2` named `H`. Line 4, the *unmap* action unmaps the physical cells `0` to `X` on the $\texttt{0}^{th}$ row of the `H` physical buffer. The following *map* action maps these cells to the cells of the `Heat` logical array, such that, for all `x` between `0` and `X`, the $\texttt{x}^{th}$ physical cell of the $\texttt{0}^{th}$ row corresponds to the $\texttt{0}^{th}$ logical cell of the $\texttt{x}^{th}$ row. As described in the InKS$_{\text{PIA}}$ code, the second dimension of the `Heat` logical array is considered as the time dimension. In order to obtain correct performance, it is mandatory to set the physical array's smaller stride dimension (the most right one) as the space dimension. That is why we specify a dimensions reordering in this mapping. And finally, this line introduces the *write* action, writing the logical region being mapped. Line 11, the *map* action changes the memory layout, based on a double-buffer strategy, using the modulo operator (`%` in C). Finally, according to the algorithm (Listing 5.11), the *read* action specified Line 12 reads values

of the previous time-step while the *write* action writes values of the current time-step `t` in the `Heat` logical array.

We can gather all these pieces of information to construct the A, U, M, R and W relation. Let $T, X, t \in N \mid 0 \leq t < T$, we can express all action relations as is in *isl* form:

$$A = \Big\{ [1,0,0] \rightarrow H[j,i] \mid 0 \leq i < X \wedge 0 \leq j < 2 \Big\}$$

$$U = \Big\{ [2,0,0] \rightarrow H[0,i] \mid 0 \leq i < X;$$
$$[5,t,1] \rightarrow H[tt,i] \mid 0 \leq i < X \wedge (tt+t)\, mod\, 2 = 0 \Big\}$$

$$M = \Big\{ [3,0,0] \rightarrow \{(H[0,i],\, Heat[i,0]) \mid 0 \leq i < X\};$$
$$[5,t,2] \rightarrow \{(H[tt,i],\, Heat[i,t]) \mid 0 \leq i < X \wedge (tt+t)\, mod\, 2 = 0\} \Big\}$$

$$R = \Big\{ [5,t,3] \rightarrow Heat[i,t-1] \mid 0 \leq i < X \Big\}$$

$$W = \Big\{ [4,0,0] \rightarrow Heat[i,0] \mid 0 \leq i < X;$$
$$[5,t,4] \rightarrow Heat[i,t] \mid 0 \leq i < X \Big\}$$

We define $\mathcal{M}_{never} \subset T^2 \times P_x \times D_x$, the set of tuples $(\vec{t}_m, \vec{t}_{max}, p, d)$ such that the physical cell p is mapped to d between $\vec{t}_m$ and $\vec{t}_{max}$. That it to say, all physical regions that were never unmapped after their mapping. $\mathcal{M}_{never}$ is computed by subtracting to M all physical regions mapped at a mapping dates $\vec{t}_m$ and unmapped at a unmapping date $\vec{t}_u \mid \vec{t}_m < \vec{t}_u$.

$$\mathcal{M}_{never} = \Big\{ (\vec{t}_m, \vec{t}_{max}, p, d) \in T^2 \times P_x \times D_x \mid (p,d) \in M(\vec{t}_m),$$
$$\nexists\, \vec{t}_u \in T,\ p \in U(\vec{t}_u),\ \vec{t}_m < \vec{t}_u \Big\}$$

Going back to our example (Listing 5.12), we can compute $\mathcal{M}_{never}$ with the M and U relations we have computed earlier. For the sake of the example, we will consider that there is at least one time-step, *i.e.* $T > 1$. In this example, the time vector $\vec{t}_{max}$ is equal to $[6,0,0]$. First, $\mathcal{M}_{never}$ contains the mappings such that they were never unmapped. We can compute it by removing from M the elements that have an unmapping date that appears after their mapping date. In this program, this corresponds to the element

$H[0,i]$ mapped at $[3,0,0]$ which has an unmapping date at $[5,t=2,1]$. Indeed, $H[tt,i] \mid (tt+t)\,mod\,2 = 0$ evaluates to $H[0,i]$ when $t=2$. Similarly, most elements have an unmapping date smaller than their mapping date, except for the last two columns of the `Heat` logical array, that is to say, the last two time-steps:

$$\mathcal{M}_{never} = \{[5,tt,2,6,0,0] \rightarrow \{(H[tt\,mod\,2,i], Heat[i,tt]) : 0 \leq i < X \wedge T-2 \leq tt < T\}$$

We define $\mathcal{M}_{between} \subset T^2 \times P_x \times D_x$, the set of tuples $(\vec{t_m}, \vec{t_u}, p, d)$ such that the physical cell p is mapped to d between $\vec{t_m}$ and $\vec{t_u}$. It is computed by taking all physical regions mapping dates from M. Then, we extract from U the unmapping of this physical region that happens after its mapping and we keep only the earliest one. Finally, we find which logical data is mapped in M.

$$\mathcal{M}_{between} = \Big\{(\vec{t_m}, \vec{t_u}, p, d) \in T^2 \times P_x \times D_x \mid (p,d) \in M(\vec{t_m}),\; p \in U(\vec{t_u}),\; \vec{t_m} < \vec{t_u},\; \nexists \vec{t_{u2}} \in T,\; p \in U(\vec{t_{u2}}),\; \vec{t_m} < \vec{t_{u2}} < \vec{t_u},\; \Big\}$$

$\mathcal{M}_{between}$ contains the mappings that are available at a given time of the simulation. Considering the time loop of Listing 5.12, there are two mappings available at each iteration. At the iteration `t=1`, two mappings are available because their unmapping date appears later:

$$\Big\{[3,0,0] \rightarrow \{(H[0,i], Heat[i,0]) : 0 \leq i < X\};\; [5,1,2] \rightarrow \{(H[1,i], Heat[i,1]) : 0 \leq i < X\}\Big\}$$

Then, because of the double-buffer strategy, as the loop moves forward, the `t%2`th row of `H` is unmapped and mapped to the next column of `Heat` (the `t`th). For instance, the mapping of `H[t mod 2, x]` $\rightarrow$ `Heat[x, t]` is valid until the iteration `t+2`, when it is mapped to another time-step of `Heat`. Figure 5.2 illustrates the displacement of the mapping depending on the time-step.

We define $\mathcal{M} \subset T^2 \times P_x \times D_x$, the union of $\mathcal{M}_{never}$ and $\mathcal{M}_{between}$. It

contains all tuples $(\vec{t_m}, \vec{t_u}, p, d)$ such that the physical cell p is mapped to d between $\vec{t_m}$ and $\vec{t_u}$. $\mathcal{M}$ of the program presented on Listing 5.12 contains the following elements:

$$\Big\{[3,0,0,5,2,1] \to \{H[0,i] \to Heat[i,0] : 0 \leq i < X\};$$
$$[5,t,2,5,t+2,1] \to \{H[t \bmod 2, i] \to Heat[i,t] : 0 \leq i < X \wedge t < T-2\}$$
$$[5,tt,2,6,0,0] \to \{H[tt \bmod 2, i] \to Heat[i,tt] : 0 \leq i < X \wedge T-2 \leq tt < T\}\Big\}$$

Note that the number of dimensions of the domain of each element is twice the size of a time vector because the first part corresponds to the mapping date while the second part gives the unmapping date.

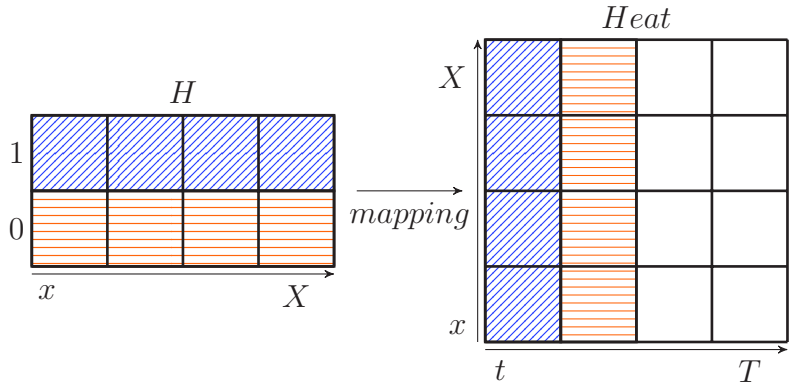
We denote $W_x \subset T^2 \times P_x \times D_x$, the set of tuples $(\vec{t_w}, \vec{t_u}, p, d)$ such that the value of the logical cell d is written in the physical cell p at the date $\vec{t_w}$ until the unmapping date $\vec{t_u}$. We obtain the mappings available at the time of writing $\vec{t_w}$ from $\mathcal{M}$ and keep only the one mapped the latest.

$$\begin{aligned}\mathcal{W}_x = \Big\{(\vec{t_w}, \vec{t_u}, p, d) \in T^2 \times P_x \times D_x \mid\ & \\ & (\vec{t_m}, \vec{t_u}, p, d) \in \mathcal{M}, \\ & d \in W(\vec{t_w}),\ \vec{t_m} < \vec{t_w} < \vec{t_u}, \\ & \nexists \vec{t_{m2}},\ \vec{t_{u2}} \in T,\ \forall p_2 \in P_x,\ (p_2, d) \in M(\vec{t_{m2}}, \vec{t_{u2}}),\ \vec{t_m} < \vec{t_{m2}} < \vec{t_w} < \vec{t_{u2}}\Big\}\end{aligned}$$

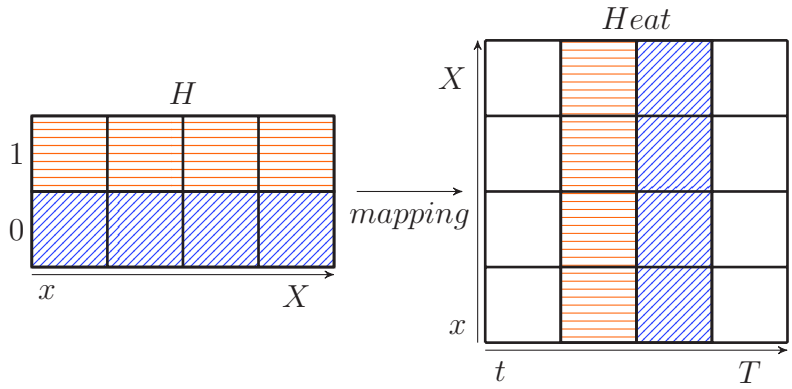
Finally, we define $R_x \subset T \times P_x \times D_x$, the relation that maps a statement time vector to the physical region this action reads That is, the elements $(\vec{t_m}, \vec{t_u}, \vec{t_w}, p, d) \in W_x$ such that $\vec{t_w} < \vec{t_r} < \vec{t_u}$ and $d = d_r$ while there is no other element with a lexicographically greater value of $\vec{t_w}$.

$$\begin{aligned}\mathcal{R}_x = \Big\{(\vec{t_r}, p, d) \in T \times P_x \times D_x \mid\ & \\ & (\vec{t_w}, \vec{t_u}, p, d) \in \mathcal{W}_x, \\ & d \in R(\vec{t_r}),\ \vec{t_w} < \vec{t_r} < \vec{t_u}, \\ & \nexists \vec{t_{w2}}, \vec{t_{u2}} \in T,\ \forall p_2 \in P_x,\ (p_2, d) \in W_x(\vec{t_{w2}}, \vec{t_{u2}}),\ \vec{t_w} < \vec{t_{w2}} < \vec{t_r} < \vec{t_{u2}}\Big\}\end{aligned}$$

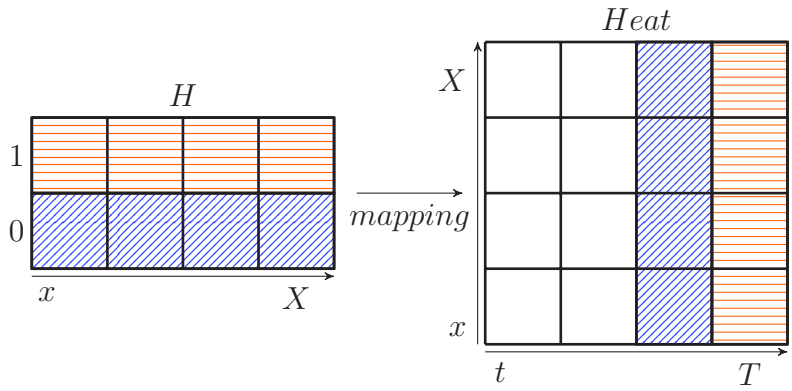
Considering again our example, on Listing 5.12, we can compute W_x



(a) $t = t_0$



(b) $t = t_0 + 1$



(c) $t = t_0 + 2$

Figure 5.2 – Modifications of the memory mapping `H` → `Heat` depending on the time-step. The mapping depends on a modulo operator applied to t (Listing 5.12). The blue, respectively green, region of `Heat` is mapped to the blue, respectively green, region of `H`. At each time-step, the least recently mapped region is unmapped to be mapped to the next region. Three time-steps are represented.

and R_x from $\mathcal{M}$, W and R. As a reminder, we had W and R equal to:

$$R = \Big\{[5,t,3] \rightarrow Heat[i,t-1]\} : 0 \leq i < X\Big\}$$
$$W = \Big\{[4,0,0] \rightarrow Heat[i,0] : 0 \leq i < X$$
$$[5,t,4] \rightarrow Heat[i,t] : 0 \leq i < X\Big\}$$

In our situation, W_x contains three elements. First, for the initialization it reads in the mapping specified by the `in` property. Then, during all time-steps `t`, except the last two, the tuple $(\vec{t_m}, \vec{t_u}, p, d) \in \mathcal{M}$ considered for writing is the one that is unmapped at `t+2`. For the last two time-steps, the tuple used is the one which is never unmapped, that is with $\vec{t_u} = \vec{t_{max}}$. R_x does not contain the unmapping date. Therefore, it contains only one element that depend on the time-step `t`: the physical region written at the previous time-step, `t-1`.

$$R_x = \Big\{[5,t,3] \rightarrow \{H[(t-1)\,mod\,2,i] \rightarrow Heat[i,t-1] : 0 \leq i < X\}\Big\}$$
$$W_x = \Big\{[3,0,0,4,0,0,5,2,1] \rightarrow \{H[0,i] \rightarrow Heat[i,0] : 0 \leq i < X\}$$
$$[5,t,2,5,t,4,5,t+2,1] \rightarrow \{H[t\,mod\,2,i] \rightarrow Heat[i,t] : 0 \leq i < X \wedge t < T-2$$
$$[5,tt,2,5,tt,4,6,0,0] \rightarrow \{H[tt\,mod\,2,i] \rightarrow Heat[i,tt] : 0 \leq i < X \wedge T-2 \leq tt < T$$
$$\Big\}$$

R_x and W_x gives us the information where to read and write during an `update` statement. In particular, it enables the compiler to find which physical buffer should be read or written during each `update` statement, as multiple buffers may be mapped to the same logical area required to be updated. In this situation, the most recently mapped buffer should be written while, in the case of a read, the most recently updated buffer should be read. It is enough to statically verify that the INKS$_{\mathsf{PSO}}$ specification is in accordance with the INKS$_{\mathsf{PIA}}$ algorithm and to generate the code.

We handle the allocation and deallocation in the same way that we have computed $\mathcal{M}$ using the *map* and *unmap* actions. That is, we can check that a physical region is valid to be mapped after it is allocated and before it is freed. Note that the `free` statement involves both a deallocation and

unmap action. Therefore, it is already taken into account to check if the buffer is accessible. That is to say, if a physical region P_x is allocated at the time vector $\vec{ta}$, mapped at the time vector $\vec{t_m}$ and freed at $\vec{tf}$, the window of time at which P_x is accessible is $\vec{ta} < \vec{t_m} < \vec{t_u} < \vec{tf}$

The last step of the compilation stage consists in generating C++ code from the InKS$_{\text{PSO}}$ statements and using the R_x and W_x relations. Each `alloc` statement is translated into a C++ buffer allocation and a set of integers used to pass from a multidimensional coordinate system to a 1D coordinate system, as illustrated on Listing 5.13 (Page 150).

Each `update for` statement is translated into a C++ loop nest. The number of nested for and their bounds correspond to the number of indices and their bounds introduced by the `update for` statement. In addition, each `OMP_FOR` option specified on top of an indice adds an `omp parallel for` directive. Similarly, each `OMP_TASK` option runs the algorithm presented in Appendix C to generate *OpenMP* task code.

Each `update` statement is composed of two phases. The first one relates to the mappings it shall use to read and write. Firstly, it intersects its reading and writing time vectors to R_x and W_x to obtain the mappings it shall use. Then, it generates a C++ class for each logical array it reads or writes. These classes are constituted of three parts. First, all the physical buffers mapped to the logical region read or written. Secondly, the integers related to these physical buffers, to pass from a multidimensional coordinate system to a 1D coordinate system. Thirdly, it contains a parenthesis operator which converts a logical coordinate to a physical coordinate, using the provided mappings, and return the value of a physical cell. This function takes integers in the logical coordinate system (*i.e.* as much integers as the logical array as dimension), uses them to pick the right physical buffer (*i.e.* depending on their value, one buffer or another shall be read/written through an `if` cascade), computes the physical coordinate in a 1D system using the mapping relation and the physical buffer's integers, and finally, return a reference to the correct value. In addition, the compiler optimizes the mapping by removing redundant condition. Typically, if only one physical buffer is read/written, it removes the `if` condition, while if multiple buffers are present, it searches for the minimal number of conditions, to

avoid a cost at runtime. In addition, the classes use constant pointers and references to constant integers, which make possible various optimizations by the C++ compiler (*e.g.* inlining, vectorization). Listing 5.13 illustrates how INKS$_{\text{PSO}}$ mappings to a logical array is turned into C++ code. Appendix B shows the complete INKS$_{\text{PSO}}$ and C++ code.

The second phase of the `update` statement generation relates to the operation ordering. In the case of an `update <operation...>` or an `update auto` statement, the automatic approach presented in Section 5.1 is in charge to generate a valid scheduling. In Section 5.1, this approach has access to all input and output relations and must generate an operation ordering that updates the INKS$_{\text{PIA}}$ output while relying only on the INKS$_{\text{PIA}}$ input. Considering an `update <operation...>` statement, it has access only to the `<operation...>` input and output relations while it must update only the provided logical region. As for an `update auto` statement, it can use all operation input and output relations, but limits the code generation to the update of the provided logical region.

Finally, each `update copy` uses R and R_x to find the logical region mapped to the provided physical region. It then uses W and W_x to find the up-to-date physical region mapped to the same logical region. Finally, it composes these two relations to obtain a relation $C\colon P_x \rightarrow P_x$. This domain of this relation is a polyhedron (*c.f.* Section 3.5) and is constituted of a multitude of integer points. We can pass this polyhedron to *isl* to generate a C++ code that traverses all points, typically in the form of a loop nest. At each point (*i.e.* at the core of the loop nest), the compiler adds an equality between the two physical elements.

Note that the operation ordering of any `update` statement is in accordance with the INKS$_{\text{PIA}}$ algorithm and the INKS$_{\text{PSO}}$ specification, identical for a given code associated to both the INKS compiler version and the *isl* version, but shall not be inferred in another way than by looking into the INKS compiler and the *isl* algorithms. For instance, it may generate loops in any order, typically to update large logical regions.

In addition, when the compiler generates a loop (*i.e.* any `update` statement, including `update for`), it can detect if loop iterations are independent, using the INKS$_{\text{PIA}}$ operation's input and output relation. In the event

that it meets this case, it adds a `#pragma ivdep` directive on top of the loop which advises the C++ compiler to use vectorization. Similarly, on top of any operation call, it adds a `#pragma forceinline recursive` directive. It is mandatory for good performance, as each operation calls a multitude of functions: the parenthesis operators used for the mapping. These calls are costly and may inhibit vectorization. Inlining the operation code as well as the parenthesis operators prevents these issues.

In this section, we have presented the InKS$_{\mathsf{PSO}}$ compiler which translate an InKS$_{\mathsf{PIA}}$ algorithm and the optimization choices written in InKS$_{\mathsf{PSO}}$ to a valid C++ code. In terms of usability, the InKS$_{\mathsf{PSO}}$ compiler is accessible: it relies on *isl* which is based on the `gmp` library, both are simple to install. As for the InKS$_{\mathsf{PSO}}$ compiler, the installation uses a Makefile. The compilation takes up to a few minutes, depending on the size of the InKS$_{\mathsf{PIA}}$ and InKS$_{\mathsf{PSO}}$ code. In the end, most of the cost to try the InKS programming model lies in the languages learning.

```
1  //*** InKSpso ***//
2  inks_function(..., Z, Y, X){
3    //Allocation of a 3D buffers
4    alloc double A(Z, Y, X)
5    alloc double B(Z, Y, X)
6
7    //Map to a 3D logical array
8    map i=[0:X[ j=[0:Y[ k=[0:Z[ A(k, j, i) = logical(0, k, j, i)
9    map i=[0:X[ j=[0:Y[ k=[0:Z[ B(k, j, i) = logical(1, k, j, i)
10
11   //Update statement using the logical array
12   update logical(0:2, 0:Z, 0:Y, 0:X) using op
13 }
14
15 //*** C++ code generated ***/
16
17 //Class definition for the mapping
18 class logical_1_t{
19     const int& Z, & Y, &X;
20     double* const& A;
21     double* const& B;
22     const size_t &A_mult_1;
23     const size_t &A_mult_2;
24     const size_t &B_mult_1;
25     const size_t &B_mult_2;
26
27 logical_1_t(/*params, buffers and multiplicators as parameters*/) :
28   /*params, buffers and multiplicators affectations*/
29 {}
30
31 //In commentary, the original non-optimized conditions
32 double& operator()(const int& t,const int& k,const int& j,const int&
       i)const{
33   //if(t==0 && 0<=k<Z && 0<=j<Y && 0<=i<X)
34   if(t==0)
35     return A[k*A_mult_2 + j*A_mult_1 + i];
36   //if(t==1 && 0<=k<Z && 0<=j<Y && 0<=i<X)
37   if(t==1)
38     return B[k*B_mult_2 + j*B_mult_1 + i];
39 } };
40
41 void inks_function(..., Z, Y, X){
42   //Allocations
43   double* A = new double[Z*Y*X];
44   size_t A_mult_2 = Y*X;
45   size_t A_mult_1 = X;
46
47   double* B = new double[Z*Y*X];
48   size_t B_mult_2 = Y*X;
49   size_t B_mult_1 = X;
50
51   //Mapping initialization
52   logical_1_t logical_1(Z, Y, X, A, B, /*mults...*/);
53
54   //Operation in a loop nest, using the logical structure
55   op(logical_1, t, k, j, i)
56 }
```

Listing 5.13 – Example of the C++ code generated by the InKS compiler for an `alloc` and `map` INKS$_{PSO}$ statements.

5.6 Discussions

In the previous chapter, we have presented the InKS programming model and its InKS$_{\mathsf{PIA}}$ language which aims to express the simulation algorithm, in this chapter we have presented four strategies to express the optimization choices, *i.e.* memory placement and operations scheduling. Firstly, we have presented a compiler-based approach demonstrating that InKS$_{\mathsf{PIA}}$ effectively express the whole algorithm. Then, we have presented InKS$_{\mathsf{Loop}}$ that handles nested loops as well as their optimizations through blocking techniques and InKS$_{\mathsf{XMP}}$ for distributed memory allocations. An evaluation of these two languages leads us to conclude that, although they are efficient for the aspects they handle, they are too specific and cannot be combined. Still, as a first approach, they enabled us to specify a more complete approach. Indeed, we have proposed a fourth approach, InKS$_{\mathsf{PSO}}$, a language with the aim of expressing all optimization concerns manually. What is common to these four approaches is that they ask for minimal pieces of information from optimization specialists and retrieve missing knowledge from the algorithm specification.

Just as InKS$_{\mathsf{PIA}}$, proposing InKS$_{\mathsf{PSO}}$ as a language that expresses optimization choices in the Polyhedral model exhibits several advantages. This strategy enables the InKS$_{\mathsf{PSO}}$ language to express a wide variety of memory layouts (*e.g.* column/row major, blocked mapping) and operations scheduling (*e.g.* nested blocked loops) while making it compatible with static analysis. Hence, we can statically check that a InKS$_{\mathsf{PSO}}$ code is in accordance with its algorithm. Moreover, various optimization transformations can be proposed in the language to ease optimization writing. For instance, in Section 5.2, we have proposed a loop transformation to introduce automatic multi-level loop blocking. Such transformation could be added as options of the InKS$_{\mathsf{PSO}}$ `for` statement easily.

However, compared to general-purpose approaches, such as C++, InKS$_{\mathsf{PSO}}$ limits the supported optimizations to what is expressible as a *Static Control Part*. Therefore, mapping relations shall only be linear functions of parameters and previous dimensions while loop nests shall only be bounded by linear constraints of parameters, surrounding loops indices

and integers. For instance, that prevents the use of optimization based on conditional statements depending on buffers' values or indices' value modification inside the loops. *Static Control Part* are presented with more details in Section 3.5. Still, a wide variety of optimization choices are expressible in the Polyhedral model, including most loop optimizations such as unrolling, skewing, fusion, fission, blocking or interchanging. Furthermore, adding the support of existing optimization tools in InKS$_{\mathsf{PSO}}$ is possible. For approaches originally designed to be used in an existing code, such as OpenMP, it consists in simply wrapping the approaches, as we did with OpenMP for the loop splitting and tasks. For others, InKS$_{\mathsf{PSO}}$ statements must be adapted to the tool. For instance, supporting PGAS model, such as XMP, is possible by providing both an `alloc` statement to use XMP allocation, such as a `XMP alloc` statement, and by adapting the `map` statement to fit the `align` directive of XMP. However, using such tools may avoid the static analysis of code; and therefore, prevent to automatically and statically detect errors. In addition, using the algorithm specification can ease the use of optimization tools.

Chapter 6

Application and evaluation of the InKS programming model

Contents

In Chapter 4, we have presented the InKS programming model which proposes to separate algorithmic and optimization concerns in numeric simulation codes. That separation aims to improve both developers productivity and codes readability, as well as easing application portability. We have also presented the algorithm language of InKS, InKS$_{\mathsf{PIA}}$, to specify all algorithmic concerns while leaving optimization choices unspecified. In Chapter 5, we have presented several approaches to express those optimization choices, for a given InKS$_{\mathsf{PIA}}$ algorithm. One in particular offers a language general enough to express a wide variety of optimization choices: InKS$_{\mathsf{PSO}}$.

In this chapter, we illustrate the usage of the InKS programming model,

composed of the InKS$_{\mathsf{PIA}}$ and InKS$_{\mathsf{PSO}}$ languages, while evaluating the approach. This evaluation discusses the InKS programming model through four angles: its gain in productivity, its generality, its simplicity of use and its performance. First, in Section 6.1 we evaluate the InKS programming model approach using synthetic benchmarks, *i.e.* the heat equation and the NAS parallel benchmarks. Then, in Section 6.2, we present a plasma physics application, the 6D Vlasov-Poisson problem, and implement it using the InKS approach. In Section 6.3, we evaluate the implementation of the 6D Vlasov-Poisson problem using InKS and compare it to a plain Fortran implementation. Finally, Section 6.4 concludes the chapter and presents some identified limitations of the InKS$_{\mathsf{PSO}}$ optimization language.

6.1 Evaluation on synthetic benchmarks

This section evaluates the InKS programming model as well as its InKS$_{\mathsf{PIA}}$ and InKS$_{\mathsf{PSO}}$ languages on the finite difference 1D heat resolution (3-point stencil) and the NAS parallel benchmarks [Bailey et al., 1991]. This evaluation discusses the InKS programming model through its gain in productivity, its generality and its efficiency. We have implemented the algorithm and optimization choices of the following programs using InKS$_{\mathsf{PIA}}$ and InKS$_{\mathsf{PSO}}$:

- finite difference 1D heat resolution (3-point stencil), a handmade C++ version as reference;
- 3 sequential NAS kernels (`IS`, `EP` and `MG`), C++ version [Griebler et al., 2018] as reference.

Listing 6.1 presents the finite difference 1D heat equation solver. It is a simple 3-point stencil code which optimization choice is based on the double-buffer strategy. NAS kernels come from a C++ version of the well-established benchmark suites: the NAS Parallel Benchmarks (NPB). However, `MG` relies on arrays of arrays which are not expressible in InKS$_{\mathsf{PSO}}$. Hence, we have slightly modified the reference to match the type of structures InKS$_{\mathsf{PSO}}$ handles. All codes were compiled with Intel 18 compiler

Algorithm		Execution time (second) InKS$_{PSO}$	Ref.	Diff.
NAS	EP	54.04 (±0.15%)	54.02 (±0.54%)	0.04%
	IS	1.9 (±0.53%)	1.92 (±1.04%)	-1.04%
	MG	4.50 (±0.33%)	4.37 (±0.92%)	2.86%
Heat eq. solver		7.05 (±0.57%)	7.06 (±0.50%)	-0.07%

Table 6.1 – Comparison of InKS$_{PSO}$ and C++ implementations of synthetic benchmarks. Execution time of the InKS$_{PSO}$ implementations of the sequential NAS benchmark (class B) and the 1D heat equation, size (2^{20} in space × 10^4 time-steps). Median in seconds of 10 executions. Maximal relative change between reference and InKS$_{PSO}$ versions.

(`icpc` with `-O3 -xHost -ip -ipo` compilation options) and executed on the *Irene* cluster (TGCC, France), equipped with 192 GB RAM and two *Xeon Platinium 8168* CPUs per node. Table 6.1 summarizes the results of the experiments.

```
1 #define i2D(x, t) ((x) + ((t)%2) * X)
2
3 for ( int t = 1; t < T; ++t ) {
4   for ( int x = 1; x < X-1; ++x ) {
5     heat[i2D(t, x)] = 0.5 * heat[i2D(t-1, x)] +
6       0.25 * (heat[i2D(t-1, x-1)]+heat[i2D(t-1, x+1)]);
7   }
8 }
```

Listing 6.1 – C++ implementation of the 1D heat equation solved using the finite difference method.

InKS separates the specification of algorithm and optimization in distinct files. Multiple optimization strategies can be implemented for a single algorithm. For instance, for this experiments, we used the `IS` and `MG` NAS kernels InKS$_{PIA}$ algorithms we have developed in previous experiments, conducted in Sections 5.2 and 5.3.

The InKS programming model aims to be general enough to express a wide variety of problems. In Sections 4.3 and 5.4, we have already presented limitations of the model. However, these experiments show that regular problems can be implemented using InKS. Still, InKS$_{PIA}$ is not as general as general-purpose language such as C. For instance, the *NAS/CG* kernel relies on runtime dependencies, *i.e.* dependencies depending on arrays'

values, which are not expressible in InKS$_{\mathsf{PIA}}$. It can be done by over-constraining the InKS$_{\mathsf{PIA}}$ code but it would limit the available choices for optimizing. In addition, InKS$_{\mathsf{PSO}}$ could not express the data structure (arrays of arrays) used in the *NAS/MG* kernel.

In terms of efficiency, the use of the InKS programming model with its optimization language does not imply any drawbacks. Table 6.1 shows that we were able to implement the optimization choices used in the 1D heat equation solver and the NAS benchmark, while preserving an identical level of performance. The InKS$_{\mathsf{PSO}}$ versions of the *NAS/IS* and *NAS/EP* benchmarks are within 1% of the C++ reference performance while the *NAS/MG* written is InKS$_{\mathsf{PSO}}$ is only 2.78% slower than its C++ counterpart. Similarly, the 1D heat equation solver written in InKS$_{\mathsf{PSO}}$ and C++ does not show any performance mismatch.

6.2 Motivating application: the 6D Vlasov-Poisson equation

This section presents the implementation of a 6D Vlasov-Poisson solver to illustrate the usage of InKS$_{\mathsf{PIA}}$ and InKS$_{\mathsf{PSO}}$ on a real use-case. The 6D Vlasov-Poisson equation, presented in (6.1), describes the dynamics of particles in a plasma and the resulting electric field. We study its resolution for a single species on a 6D Cartesian mesh with periodic boundary conditions. The main unknown is f (`f6d` in the code), the distribution function of particles in 6D phase space. The vector field E corresponds to the electric field whereas the scalar fields ρ and ϕ respectively represent the charge density and the electric potential.

$$\begin{cases} \dfrac{\partial f(t,x,v)}{\partial t} + v.\nabla_x f(t,x,v) - E(t,x).\nabla_v f(t,x,v) = 0 \\ -\Delta\phi(t,x) = 1 - \rho(t,x) \\ E(t,x) = -\nabla\phi(t,x) \\ \rho(t,x) = \displaystyle\int f(t,x,v)dv \end{cases} \tag{6.1}$$

For the resolution of the Vlasov part we rely on a Strang splitting (order 2 in time) while we solve the Poisson part using a fast Fourier transform (FFT). This leads to six 1D advections: three in space dimensions (x_1, x_2, x_3) and three in velocity dimensions (v_1, v_2, v_3). For each advection in space dimensions we rely on a Lagrange interpolation of degree 4, while for the advections in velocity dimensions we use an interpolation of degree 3. In the space dimensions, we use a semi-Lagrangian approach, where the stencil is not applied around the destination point but at the foot of characteristics, only known at runtime. The methods used for the equation solving are described in more details in [Mehrenberger et al., 2013]. The algorithm and test case come from the SeLaLib [Inria, IPP, IRMA, IRMAR, LJLL, 2018] Fortran implementation.

Due to the Strang splitting, a first half time-step of advections is required after `f6D` initialization, but before the main time-loop. These advections need the electric field E as input. E is obtained through the FFT-based Poisson solver that in turn needs the charge density ρ as input. ρ is computed by integrals over the three velocity dimensions of `f6D`. The main time-loop is composed of 4 steps: advections in space dimensions, computation of the charge density (reduction) and electric field (Poisson solver) and advections in velocity dimensions. However, the two advection steps account for more than 99% of the single-node computation time. The 6D Vlasov-Poisson algorithm is presented in Algorithm 1. Algorithm 2 details the Poisson solver. Figure 6.1 shows a graphical representation of the time-loop of the 6D Vlasov-Poisson SeLaLib implementation.

6.2.1 InKS$_{\mathsf{PIA}}$ version of the 6D Vlasov-Poisson algorithm

To obtain an implementation of the 6D Vlasov-Poisson, the first step is to implement its algorithm using InKS$_{\mathsf{PIA}}$. Listing 6.2 presents the InKS$_{\mathsf{PIA}}$ implementation of Advection v_1, as well as the relevant part of the simulation kernel for this advection. The complete InKS$_{\mathsf{PIA}}$ implementation of the 6D Vlasov-Poisson solving is available on Appendix A.

The first part of the InKS$_{\mathsf{PIA}}$ code describes each fine grain operation.

Algorithm 1 The 6D Vlasov-Poisson algorithm.

Input: $f6d(0, x, v)$
Input: T, the number of time-steps
Output: $f6d(T, x, v)$
  $\rho(0, x) \leftarrow reduction(f6d(0, x, v))$
  $\phi(0, x) \leftarrow Poisson(\rho(0, x))$
  $E(0, x) \leftarrow solve(\phi(0, x))$
  $f6d(0, x, v) \leftarrow advection_{v123}(f6d(0, x, v))$
  for $t = 1$ **to** T **do**
    $f6d(t, x, v) \leftarrow advection_{x123}(f6d(t - 1, x, v))$
    $\rho(t, x) \leftarrow reduction(f6d(t, x, v))$
    $\phi(t, x) \leftarrow Poisson(\rho(t, x))$
    $E(t, x) \leftarrow solve(\phi(t, x))$
    $f6d(t, x, v) \leftarrow advection_{v123}(f6d(t, x, v), E(t, x))$
  end for

Algorithm 2 The 6D Poisson solver algorithm.

Input: $\rho(t, x)$
Output: $\phi(t, x)$
  {Fourier transform in the dimension 1, 2 and 3}
  $\hat{\rho}(t, \xi_x) \leftarrow \mathcal{F}_x(\rho(t, x))$
  $\hat{\rho}(t, \xi_{xy}) \leftarrow \mathcal{F}_y(\hat{\rho}(t, \xi_x))$
  $\hat{\rho}(t, \xi_{xyz}) \leftarrow \mathcal{F}_z(\hat{\rho}(t, \xi_{xy}))$

  {$\hat{\phi}$ computation}
  $\hat{\phi}(t, \xi_{xyz}) = \hat{\rho}(t, \xi_{xyz})/k^2$

  {Inverse Fourier transform in the dim. 3, 2 and 1}
  $\hat{\phi}(t, \xi_{xy}) \leftarrow \mathcal{F}_z^{-1}(\hat{\phi}(t, \xi_{xyz}))$
  $\hat{\phi}(t, \xi_x) \leftarrow \mathcal{F}_y^{-1}(\hat{\phi}(t, \xi_{xy}))$
  $\phi(t, x) \leftarrow \mathcal{F}_z^{-1}(\hat{\phi}(t, \xi_x))$

In the 6D Vlasov-Poisson problem, that corresponds to the 6 advections, the FFTs for the Poisson solver, the reduction to compute ρ, etc. On lines 1 to 12 of Listing 6.2 (Page 161), we define Advection v_1 as an operation of our algorithm. Line 1 defines the operation, named `adv_v1`, and the integers used to access cells in logical arrays as well as defined data dependencies. Integers `i` to `step` are used to access space, velocity and time dimensions while `D4` and `s4` are only used to define data dependencies. The `D4` integer corresponds to the size of the domain in the 4^{th} dimension whereas `s4`

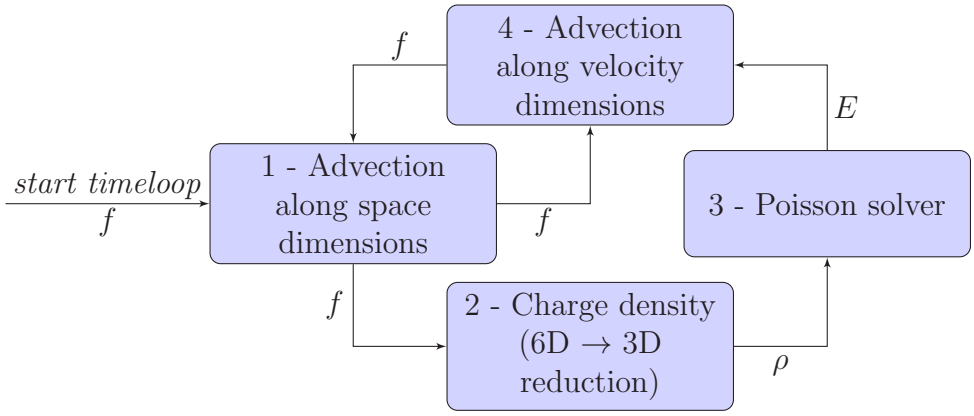


Figure 6.1 – The 6D Vlasov-Poisson solver time-loop.

represents the size of the stencil, used as the number of cells that can be accessed at the boundaries of the 4^{th} dimension of `f6d`, because of the domain periodicity. Lines 2 to 4 define the `f6d` logical array in which the operation read and write cells depending on the integer parameters. Line 5 adds the `p_v1` logical array, containing the advection coefficient. Notice that the memory layout is left unspecified thanks to the dynamic single assignment form. For instance, the time dimension is present in the logical memory layout for both arrays, represented by the `t` variable. Similarly, `f6d` has an 8^{th} dimension, here `step`, that represents the evolution through a time-step. Each *step* at a time-step t, from 0 to 5, holds the values of `f6d` after the $step^{th}$ advection at the t^{th} time-step. Finally, lines 8 to 11 declare the computation using C++ code and parenthesis operators to access logical cells.

The second part of the InKS$_{\text{PIA}}$ code implements the entry-point of the simulation, *i.e.* the simulation kernel which defines the validity domain of each operation, the set of logical arrays that exist and the subset of each array that is available at the beginning and expected by the end of the simulation. This corresponds to the lines 16 to 31 in Listing 6.2. Line 16 defines the `inks_vlasov_poisson6d` as the simulation kernel, parameterized by a set of integers, among which the number of time-steps (`n_iter`) or the number of cells in each dimension (`D1` to `D6`). These integers are set at the execution of the InKS$_{\text{PIA}}$ simulation. Lines 19 to 21 define the set of existing logical arrays, here `f6d` and `p_v1` for Advection v_1 operation. Note

that all values (aside from the halo region) at the first time-step of the `f6d` logical array are marked as available at the beginning of the simulation. And finally, lines 25 to 29 set the parameters of each operation; that is the validity domain and logical arrays. This domain represents the possible values for each operation integer parameter. For instance, the parameter `i` of Advection v_1 can take any value between `s1` included and `D1+s1` excluded.

```
1  op adv_v1(i, j, k, l, m, n, t, step, D4, s4) : (
2    double f6d {in: v1=[0:D4+2*s4[
3            (i, j, k, v1, m, n, t, step-1) |
4          out: (i, j, k, l, m, n, t, step)},
5    double p_v1{in: (i,j,k,t,0:3)}
6  )
7  #CODE (C)
8    f6d(i, j, k, l, m, n, t, step) =
9    p_v1(i,j,k,t,0) * f6d(i, j, k, l-1, m, n, t, step-1)
10 + p_v1(i,j,k,t,1) * f6d(i, j, k, l+0, m, n, t, step-1)
11 + p_v1(i,j,k,t,2) * f6d(i, j, k, l+1, m, n, t, step-1);
12 #END
13
14 /*...other operations definition...*/
15
16 simulation inks_vlasov_poisson6d(D1, D2, D3, D4, D5, D6,
17   nt, s1, s2, s3, s4, s5, s6) : (
18
19   double f6d(8) {in:
20     (s1:D1+s1, s2:D2+s2, s3:D3+s3, s4:D4+s4,
21        s5:D5+s5, s6:D6+s6, 0, 2)
22   },
23   double p_v1(5),
24   /*...other logical arrays...*/
25 )
26 #CODE (inks)
27 advection_v1 x1=[s1:D1+s1[ x2=[s2:D2+s2[ x3=[s3:D3+s3[
28   v1=[s4:D4+s4[ v2=[s5:D5+s5[ v3=[s6:D6+s6[ t=[0:nt[
29   (x1, x2, x3, v1, v2, v3, t, D4, 3, s4) : (f6d, p_v1)
30
31 /*...other operations...*/
32 #END
```

Listing 6.2 – InKS$_{\text{PIA}}$ implementation of Advection v_1.

Once the InKS$_{\text{PIA}}$ algorithm is written, the InKS$_{\text{C}}$ compiler can automatically generate a first instance of the 6D Vlasov-Poisson solver. We used this to check that our algorithm is correct and generates the same results as the SeLaLib implementation. In particular, we have run both implementation using the parameters presented in Table 6.2. The test case used is the *linear Landau damping* shown in Equation (6.2). The 4 dimensional version of this test case is presented in [Filbet et al., 2001]. The linear Landau damping corresponds to a perturbation in space dimensions. According

to the Landau's theory, this `f6d` initial condition leads to an oscillating electric energy over time which amplitude decays exponentially. Figure 6.2 presents the electric energy of both versions as well as their differences. Additionally, it shows the oscillation and exponential decay of the electric energy. Table 6.3 shows L_1, L_2 and L_{inf} norms of the difference between the InKS and the SeLaLib versions of `f6d`. The errors are within the machine error, acknowledging the validity of the algorithm. After this verification, the second step is to use InKS$_{\texttt{PSO}}$ to manually implement efficient sets of optimization choices.

$$f(0, x_1, x_2, x_3, v_1, v_2, v_3) = \frac{1}{(2\pi)^{3/2}} \, e^{-(\sum_{i=1}^{3} v_i^2)/2} \, (1 + \alpha \, \Pi_{i=1}^{3} cos(0.5 x_i)) \tag{6.2}$$

						Stencil orders					
Size	Δt	Iterations	$L_x = L_y = L_z$	v_{min}	v_{max}	Velocity	Space	α	n_{mode}	$v_{thermal}$	B0
32^6	0.1	200	12.5663706144	-6	6	3	4	0.01	1	1	1

Table 6.2 – 6D Vlasov-Poisson physical parameters.

	Absolute			Relative			Local relative		
Iteration	L_1	L_2	L_{inf}	L_1	L_2	L_{inf}	L_1	L_2	L_{inf}
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
200	6.8 e-35	3.3 e-32	5.1 e-29	1.2 e-31	9.1 e-30	7.9 e-28	1.3 e-21	6.4 e-19	1.0 e-15

Table 6.3 – Absolute and relative norms of the difference between the InKS and the SeLaLib versions of `f6d` after 10 and 200 time-steps. The physical parameters used are presented on Table 6.2. The absolute norm corresponds to $\|f6d_{selalib} - f6d_{inks}\|$. The relative norm is $\frac{\|f6d_{selalib} - f6d_{inks}\|}{\|f6d_{selalib}\|}$. The local relative norm equals to $\left\| \frac{f6d_{selalib} - f6d_{inks}}{f6d_{selalib}} \right\|$. Norms are computed from values using a double-precision floating-point format. The zeros at the 10^{th} time-step indicate that values are bit-by-bit identical.

6.2.2 Optimization of the 6D Vlasov-Poisson solver

In addition to the complexity of the plasma physics and the mathematics domain, the 6D Vlasov-Poisson solver is demanding both in terms of mem-

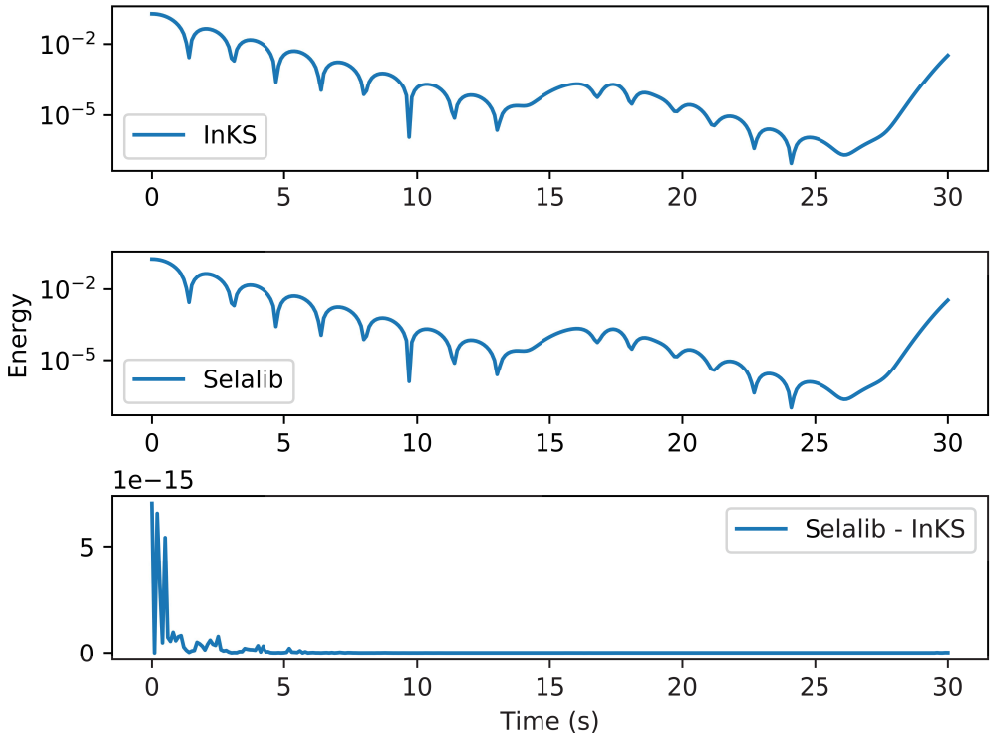


Figure 6.2 – 6D Vlasov-Poisson electric energy in function of time of the InKS and SeLaLib version and their absolute difference. The electric energy $\mathcal{E}$ is defined as $\mathcal{E}(t) = \sqrt{\int_0^{L_z} \int_0^{L_y} \int_0^{L_x} E(x, y, x, t)^2 \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z}$. The physical parameters used are presented on Table 6.2.

ory and computing power. Indeed, this problem is highly memory bound. While advections, which are the computation intensive parts of the application, are very regular computations and do not need load-balancing, they are applied to a 6D domain, which requires both a huge memory footprint and a particular attention to cache use. Therefore, its implementation in SeLaLib is based on a set of complex optimization choices [Kormann et al., 2019], among which multiple MPI strategies, OpenMP implementations and complex loops implementations with blocking and data layout modifications to improve cache use and expose vectorization. In the rest of the section, we present some optimizations implemented in SeLaLib and their expression using the InKS_{PSO} language. We illustrate all these optimizations on Advection v_1. We also present the Poisson solver.

Double-buffer implementation

One of the most naive strategy to apply a stencil of a domain is to use the *double-buffer strategy*. It consists in having an input buffer from which to read the values and a distinct output buffer in which to write the values produced by the computations. Listing 6.3 presents this strategy applied to the `f6d` array for Advection v_1 computation. Line 5 allocates two six dimensional array (`F6D`) that will be used for the double-buffer memory layout. Hence, before each advection a, one of these six dimensional array contains the output values of the previous advection while the other is mapped to the logical output of the advection a. For instance, before Advection v_1, on Line 12, `F6D(0, ...)` contains the values of `f6d(..., 2)`, that is, the output of Advection x_1. On the same Line, `F6D(1, ...)` is mapped to `f6d(..., 3)`, that is, the output of Advection v_1. Unlike following strategies, this one is not currently implemented in SeLaLib.

```
1  dom x1=[0:D1[ x2=[0:D2[ x3=[0:D3[
2    v1=[0:D4[ v2=[0:D5[ v3=[0:D6[
3
4  //Allocate a double-buffer
5  alloc double F6D(2, D6, D5, D4, D3, D2, D1)
6
7  /*...*/
8
9  /* Surrounded by the time loop "t" */
10 {
11   //Define the double-buffer strategy
12   map F6D(1, v3, v2, v1, x3, x2, x1) =
13     f6d(x1, x2, x3, v1, v2, v3, t, 3)
14
15   //specify the region to update
16   update f6d(0:D1, 0:D2, 0:D3, 0:D4, 0:D5, 0:D6, t, 3)
17     //Introduce 6 indices
18     using for(n{0:D6}, m{0:D5}, l{0:D4},
19       k{0:D3}, j{0:D2}, i{0:D1}){
20         //Require the update of a logical cell
21         update f6d(i, j, k, l, m, n, t, 3) using adv_v1
22     }
23 }
```

Listing 6.3 – Double-buffer implementation of Advection v_1 using InKS$_{\mathsf{PSO}}$.

Intermediate buffers

The double-buffer strategy is not practical for real test-cases of the 6D Vlasov-Poisson problem, because of the huge amount of memory two six

dimensional arrays require. Because the considered advections are 1D stencils, a possibility is to use a 1D intermediate buffer. The idea is to copy a line of `f6d` into a buffer and to use it as input, while writing into `f6d` during the advection computation. Going even further, we can use another buffer to write the values and copy them back to `f6d` after the computation. This strategy enables to have `f6d` and two small buffers whose size is equal to the size of a line of `f6d` instead of two `f6d`; hence drastically reducing memory footprint. It also improves cache use and makes possible vectorization. Indeed, during the computation, only the two contiguous buffers are used. On the contrary, `F6D` is accessed contiguously only when an advection reads and writes values to its contiguous dimension; that is to say, during Advection x_1. A last advantage is to avoid the use of modulo operations to handle domain periodicity. This concern can be handled by adding a halo zone at each boundary of the buffer used as input. This would have required a lot of memory for a six dimensional array, but has little impact on a small buffer. The *intermediate buffers* strategy implemented in SeLaLib is presented in Listing 6.4.

```
! buffer allocations
allocate(buf_i(0:D4+2*s4))
allocate(buf_o(0:D4+2*s4))

do n, m, k, j, i
  ! copy f6d to input buffer
  buf_i(s4:D4+s4) = f6d(i,j,k,s4:D4+s4,m,n)

  ! copy left and right boundaries
  buf_i(0:s4) = buf_i(D4+s4:D4+2*s4)
  buf_i(D4+s4:D4+2*s4) = buf_i(s4:2*s4)

  do l=s4, D4+s4
    ! advection v_1
  end do

  !copy output buffer to f6d
  f6d(i,j,k,s4:D4+s4,m,n) = buf_o(s4:D4+s4)
end do
```

Listing 6.4 – Usage of intermediate buffer in Advection v_1 using Fortran.

In comparison, Listing 6.5 presents the same strategy using InKS$_{\text{PSO}}$. First, the input and output buffers, respectively named `BUF_I` and `BUF_O`, are allocated on Lines 6 and 7. Then, on Line 13, the input buffer is mapped to the `f6d` logical array. It is followed by an `update copy` that will copy

the value from F6D, the buffer that holds the f6d logical array values, to BUF_I. Then, left and right boundaries are copied by the copy_left_right_4 operation. Line 19 remaps the processed block of the F6D buffer, already copied to BUF_I, to the same block in f6d at the next *step*. That is, the third step which corresponds to the values after Advection v_1. It is followed by Line 22 that maps the output buffer BUF_O to the f6d area being processed by the calls of Advection v_1. Finally, Line 25, an update copy is performed from BUF_O to F6D.

```
1  //Named bounded sets to ease code writing
2  dom x1d=[s1:D1+s1[ x2d=[s2:D2+s2[ x3d=[s3:D3+s3[
3    v1d=[s4:D4+s4[ v2d=[s5:D5+s5[ v3d=[s6:D6+s6[
4    v1=[0:D4[ fullv1=[0:D4+2*s4[ blockDom=[0:blockSize[
5
6  alloc double BUF_I(D4+2*s4)
7  alloc double BUF_O(D4+2*s4)
8
9  update f6d(x1d, x2d, x3d, v1d, v2d, v3d, t, 3)
10   using for(n {s6:D6+s6}, m {s5:D5+s5},
11       k {s3:D3+s3}, j {s2:D2+s2}, i {s1:D1+s1})
12 {
13   map BUF_I(fullv1) = f6d(i, j, k, fullv1, m, n, t, 2)
14   update BUF_I(v1d) using copy
15
16   update f6d(i, j, k, fullv1, m, n, t, 2)
17     using copy_left_right_4
18
19   map F6D(n-s6, m-s5, v1d-s4, k-s3, j-s2, i-s1) =
20     f6d(i, j, k, v1d, m, n, t, 3)
21
22   map BUF_O(v1d) = f6d(i, j, k, v1d, m, n, t, 3)
23   update f6d(i, j, k, v1d, m, n, t, 3) using adv_v1
24
25   update F6D(n-s6, m-s5, v1, k-s3, j-s2, i-s1) using copy
26 }
27 free BUF_I
28 free BUF_O
```

Listing 6.5 – Intermediate buffers in Advection v_1 using InKS$_{\text{PSO}}$ (Note that the variables written in capital letters refer to physical buffers).

Blocked copies

It is possible to improve the previous optimization. Instead of copying a single line of f6d to the buffer, we can copy multiple contiguous elements of f6d to a 2D buffer. Indeed, as we copy an element of f6d, an entire cache line is moved from the memory to the cache. Since these values are already in cache, we can copy them to our buffer at a little additional cost.

The output buffer, its copy to `f6d` and the advection computation are also done in a blocked fashion. The buffer is now the size of a `f6d` line times the size of block, *i.e.* a cache line. Although it is larger, it remains small enough. On the contrary, copies and computations are now using the cache more efficiently.

The InKS$_{\text{PSO}}$ implementation of that strategy is presented in Listing 6.6. Lines 6 and 7, the buffers are allocated as 2D array. The first dimension is for the input and output, while the second deals with the blocking. The mappings of the buffers are done accordingly lines 13 and 28. Finally, the loops are written in a blocked fashion. On Line 11, `ii` iterates from `s1` to `D1+s1` in steps of the size of the block, `blockSize`, while on Line 25, `i` iterates from `ii` to `min(ii+blockSize, D1+s1)`.

OpenMP implementation

The last optimization consists in using OpenMP to share the copies and advection computations between multiple threads. The parallelism requires the allocation of the input and output buffers for each thread. The rest of the parallelism is straightforward as both the computations and copies do not need load-balancing strategy.

This third optimization is shown in Listing 6.7. It is done through the use of an OpenMP parallel region, Line 6, and an `OMP for` option, Line 12. Since the buffers are allocated inside the parallel region, on Lines 8 and 9, each thread as access to his own buffers. Finally, Line 12, we use the `OMP` option to add a `#pragma omp for collapse` in the generated code; that is, loop spiting parallelism.

Poisson solver

The algorithm of the Poisson solver is presented in Algorithm 2. It is extremely verbose even though it is quite simple and does not take much execution time, in a single node execution, especially compared to the advections. Listing 6.8 presents the SeLaLib sequential implementation of the Poisson solver.

As shown in Listing 6.8, the Poisson solver computes ϕ and relies on 7

```
1  //Named bounded sets to ease code writing
2  dom x1d=[s1:D1+s1[ x2d=[s2:D2+s2[ x3d=[s3:D3+s3[
3    v1d=[s4:D4+s4[ v2d=[s5:D5+s5[ v3d=[s6:D6+s6[
4    v1=[0:D4[ fullv1=[0:D4+2*s4[ blockDom=[0:blockSize[
5
6  alloc double BUF_I(blockSize, D4+2*s4)
7  alloc double BUF_O(blockSize, D4+2*s4)
8
9  update f6d(x1d, x2d, x3d, v1d, v2d, v3d, t, 3)
10   using for({OMP collapse(2)} n {s6:D6+s6}, m {s5:D5+s5},
11       k {s3:D3+s3}, j {s2:D2+s2}, ii {s1:D1+s1:blockSize})
12 {
13   map BUF_I(blockDom, fullv1) =
14     f6d(ii+blockDom, j, k, fullv1, m, n, t, 2)
15   update BUF_I(blockDom, v1d) using copy
16
17   dom iii=[ii:ii+blockSize and :D1+s1[
18   update f6d(iii, j, k, fullv1, m, n, t, 2)
19     using copy_left_right_4
20
21   map F6D(n-s6, m-s5, v1d-s4, k-s3, j-s2, iii-s1) =
22     f6d(iii, j, k, v1d, m, n, t, 3)
23
24   update f6d(iii, j, k, v1d, m, n, t, 3) using
25     for(i {ii:ii+blockSize and :D1+s1})
26 //ii<=i<ii+bSize && i<D1+s1 => ii<=i<min(ii+bSize, D1+s1)
27   {
28     map BUF_O(i-ii, v1d) =
29       f6d(i, j, k, v1d, m, n, t, 3)
30     update f6d(i, j, k, v1d, m, n, t, 3) using adv_v1
31   }
32
33   update F6D(n-s6, m-s5, v1, k-s3, j-s2, iii-s1)
34     using copy
35 }
36 free BUF_I
37 free BUF_O
```

Listing 6.6 – Usage of blocking in Advection v_1 using InKS$_{\mathsf{PSO}}$.

operations: three FFTs, the $\hat{\phi}$ solving and the three inverse FFTs. Since all these operations are defined in the InKS$_{\mathsf{PIA}}$ code, it is possible to ask InKS$_{\mathsf{PSO}}$ to schedule all of them automatically. This strategy may not be the most efficient one; however, it improves readability while not hurting the application performance. Indeed, the Poisson solver counts for less than 1% of the computation time (in a single node version). This strategy is implemented using the `update auto` of InKS$_{\mathsf{PSO}}$ in Listing 6.9.

```
1  //Named bounded sets to ease code writing
2  dom x1d=[s1:D1+s1[ x2d=[s2:D2+s2[ x3d=[s3:D3+s3[
3    v1d=[s4:D4+s4[ v2d=[s5:D5+s5[ v3d=[s6:D6+s6[
4    v1=[0:D4[ fullv1=[0:D4+2*s4[ blockDom=[0:blockSize[
5
6  OMP parallel
7  {
8  alloc double BUF_I(blockSize, D4+2*s4)
9  alloc double BUF_O(blockSize, D4+2*s4)
10
11 update f6d(x1d, x2d, x3d, v1d, v2d, v3d, t, 3)
12   using for({OMP collapse(2)} n {s6:D6+s6}, m {s5:D5+s5},
13       k {s3:D3+s3}, j {s2:D2+s2}, ii {s1:D1+s1:blockSize})
14 {
15   map BUF_I(blockDom, fullv1) =
16     f6d(ii+blockDom, j, k, fullv1, m, n, t, 2)
17   update BUF_I(blockDom, v1d) using copy
18
19   dom iii=[ii:ii+blockSize and :D1+s1[
20   update f6d(iii, j, k, fullv1, m, n, t, 2)
21     using copy_left_right_4
22
23   map F6D(n-s6, m-s5, v1d-s4, k-s3, j-s2, iii-s1) =
24     f6d(iii, j, k, v1d, m, n, t, 3)
25
26   update f6d(iii, j, k, v1d, m, n, t, 3) using
27     for(i {ii:ii+blockSize and :D1+s1})
28   {
29     map BUF_O(i-ii, v1d) =
30       f6d(i, j, k, v1d, m, n, t, 3)
31     update f6d(i, j, k, v1d, m, n, t, 3) using adv_v1
32   }
33
34   update F6D(n-s6, m-s5, v1, k-s3, j-s2, iii-s1)
35     using copy
36 }
37 free BUF_I
38 free BUF_O
39 }
```

Listing 6.7 – InKS$_{\mathsf{PSO}}$ implementation of Advection v_1.

6.3 Evaluation on the 6D Vlasov-Poisson equation

This section evaluates the InKS programming model as well as its InKS$_{\mathsf{PIA}}$ and InKS$_{\mathsf{PSO}}$ languages on the 6D Vlasov-Poisson solver. This evaluation discusses the InKS programming model through four angles: its gain in productivity, its generality, its simplicity of use and its performance. First, we have implemented the algorithm of the complete 6D Vlasov-Poisson equation in InKS$_{\mathsf{PIA}}$, using Fortran/OpenMP SeLaLib as reference.

```
! 1) FFTs in x, y, z directions
  do k=1,D3-1
    do j=1,D2-1
      fftw_in(:) = rho(:, j, k)
      fftw_execute(plan_x)
      hat_rho(:, j, k) = fftw_out(:)
    enddo
  enddo

  ! i, k do loops
      fftw_in(:) = hat_rho(i, :, k)
      fftw_execute(plan_y)
      hat_rho(i, :, k) = fftw_out(:)
  ! end do loops

  ! i, j do loops
      fftw_in(:) = hat_rho(i, j, :)
      fftw_execute(plan_z)
      hat_rho(i, j, :) = fftw_out(:)
  ! end do loops

! 2) hat_phi computation
  ! i, j, k do loops
    hat_phi(i, j, k) = hat_rho(i, j, k) / (0.75 * PI**2)

! 3) Inverse FFTs in z, y, x directions
  ! i, j do loops
      fftw_in(:) = hat_phi(i, j, :)
      fftw_execute(plan_inv_z)
      hat_phi(i, j, :) = fftw_out(:)
  ! end do loops

  ! i, k do loops
      fftw_in(:) = hat_phi(i, :, k)
      fftw_execute(plan_inv_y)
      hat_phi(i, :, k) = fftw_out(:)
  ! end do loops

  ! j, k do loops
      fftw_in(:) = hat_phi(:, j, k)
      fftw_execute(plan_inv_x)
      phi(:, j, k) = fftw_out(:)
  ! end do loops
```

Listing 6.8 – First steps of the Poisson solver.

The equation and general algorithm are presented in Section 6.2.1. Then, we have implemented and compared four InKS$_{\mathsf{PSO}}$ optimizations of the full 6D Vasov-Poisson solver described in Section 6.2.2 with SeLaLib as reference. *C.f.* 6.1 for compiler and architecture details. Table 6.4 presents the results for the experiments.

The InKS programming model separates algorithmic and optimization concerns with two distinct languages. Fixing the simulation algorithm ex-

```
1 /*...rho is available while phi is mapped...*/
2 update phi(x1dom, x2dom, x3dom, t) using auto
```

Listing 6.9 – InKS$_{\mathsf{PSO}}$ implementation of the Poisson solver.

Optimization version	InKS	SeLaLib	Diff.
Double buffering	29.60 (±3.02%)	N/A	N/A
Intermediate buffer	34.07 (±2.80%)	44.26 (±0.32%)	-23.03%
Blocked copies	16.64 (±2.61%)	24.99 (±0.64%)	-33.43%
OpenMP (8 threads)	2.54 (±3.94%)	3.08 (±1.78%)	-17.53%

Table 6.4 – Comparison of InKS (InKS$_{\mathsf{PIA}}$ + InKS$_{\mathsf{PSO}}$) and SeLaLib (Fortran) implementations of the 6D Vlasov-Poisson solver. Time/iteration in seconds of the InKS and the SeLaLib implementations of the 6D Vlasov-Poisson, size (32^6). Median in seconds of 10 time-steps. Maximal relative change between references and InKS versions. The maximal relative change r to the median m of a set of n values V is defined as $r = \frac{m}{max_{i=0..n}(|m-V_i|)}$.

Optimization version	InKS	Modified SeLaLib		Diff.
Double buffering	29.60 (±3.02%)	N/A		N/A
Intermediate buffer	34.07 (±2.80%)	44.82	(±2.09%)	-23.99%
Blocked copies	16.64 (±2.61%)	20.09	(±0.64%)	-17.19%
OpenMP (8 threads)	2.54 (±3.94%)	2.51	(±2.53%)	1.52 %

Table 6.5 – Comparison of InKS (InKS$_{\mathsf{PIA}}$ + InKS$_{\mathsf{PSO}}$) and modified SeLaLib (Fortran) implementations of the 6D Vlasov-Poisson solver. Time/iteration in seconds of the InKS and the modified SeLaLib implementations of the 6D Vlasov-Poisson, size (32^6). Median in seconds of 10 time-steps. Maximal relative change between references and InKS versions.

hibits several advantages compared to traditional approaches. Firstly, it eases the collaboration between specialists of the simulated domain and specialists of computer optimizations. Secondly, and more importantly, it limits the parts of code that must be rewritten to implement new optimization strategies. As shown in Table 6.4, using a single InKS$_{\mathsf{PIA}}$ code describing the 6D Vlasov-Poisson solver, we have derived 4 different versions of optimization choices. To target the bleeding edge of supercomputers architectures, scientists in the field of numeric simulation have to tune their code while having limited information about which optimization strategy is the best for a given architecture. Narrowing the part of code to rewrite

	Number of lines	
Code	InKS$_{\mathsf{PSO}}$	SeLaLib
Poisson operator	1	70
Vlasov operator	6×18	6×22

Table 6.6 – Comparison of the number of lines of InKS (InKS$_{\mathsf{PSO}}$) and SeLaLib (Fortran) implementations of the 6D Vlasov-Poisson solver. Comparison of the InKS$_{\mathsf{PSO}}$ OpenMP version and the SeLaLib Fortran version. Only the optimization choices were considered: in both versions, the algorithm (*i.e.* advection computations or the Fast Fourier Transform calls) was not counted.

diminishes implementation costs; and therefore, makes possible a more extensive exploration of the best optimization strategy for each architecture, especially the new ones.

An advantage of the Polyhedral model used in the InKS approach relates to the code efficiency. The algorithm gives a perfect knowledge of the users objective to the InKS compiler, which can then use it to automatically optimize parts of the code, whereas general-purpose approaches fail in understanding its users actual goals. For instance, our compiler can check whether there are dependencies between iterations of a loop. It can then either suggest to the user to add a parallel construct around this loop or inform the C++ compiler that this loop contains no dependence through the use of directives. At the moment, we chose to implement the second possibility in our compiler. The information coming from the algorithm and its use by the compiler helped us obtain performance improvements over the SeLaLib reference versions, automatically. As shown in Table 6.4, the InKS$_{\mathsf{PSO}}$ *intermediate buffer* version surpasses the SeLaLib reference by a factor 1.30. Similarly, our *blocked copies* version is 1.5 faster than the reference whereas our *OpenMP* parallel version improves the reference, achieving a speedup up of 1.21. We were able to reduce the performance gap by adding vectorization directives manually to the SeLaLib code. That corresponds to the modified SeLaLib version whose results are presented in Table 6.5. Hence, comparing the InKS$_{\mathsf{PSO}}$ and the modified SeLaLib versions, the speedup of the *blocked copies* version is reduced from 1.5 to 1.21. As for the *OpenMP* version, there is no performance gap; with a difference

inside the confidence interval.

Although our implementation of the InKS programming model is not as general as general-purpose language such as C, Listing 6.7 shows it is possible to implement the optimization strategies done in *SeLaLib*, including the addition of intermediate copies, highlights of vectorization possibilities to the compiler and uses of OpenMP. InKS$_{\mathsf{PIA}}$ and InKS$_{\mathsf{PSO}}$ are thus general enough to express a wide variety of programs and optimization strategies. More specifically, InKS$_{\mathsf{PIA}}$ can express algorithm of programs analogous to Parameterized Task Graph ([Cosnard and Jeannot, 1999]): directed graph of tasks dependent on invariant integer parameters set at execution. InKS$_{\mathsf{PSO}}$ is capable of specifying optimizations that are compatible with the Polyhedral model. That includes row and column-major memory layouts, as well as more convoluted ones (*e.g.* blocked memory layouts), and operation ordering based on loop nests, optimized with traditional techniques (*e.g.* unrolling, skewing, fusion, fission, blocking, interchanging and more). Moreover, while PTGs cover many classes of programs, the InKS programming model is usable in conjunction with traditional programming models, such as C or Fortran.

In terms of productivity, as illustrated in Listings 6.2, expressing algorithms in InKS$_{\mathsf{PIA}}$ is similar to writing a naive C implementation, free of any platform-specific optimization, and where loops are replaced by InKS validity domains. However, contrary to C, InKS$_{\mathsf{PIA}}$ is dedicated to express the algorithmic concerns and is more suited to be used as a base for optimization.

InKS$_{\mathsf{PSO}}$ heavily relies on the algorithm description written using InKS$_{\mathsf{PIA}}$, as illustrated in Listings 6.7. Basically, it expresses optimization choices using three steps. The first one consists in allocating a buffer using the `alloc` instruction. This is close to a Fortran allocation. Then, using the `map` instruction, users describe the memory layouts. That is, a function that maps a logical coordinate to a physical one. We believe that this is no more complex than an index computation in C, especially since the InKS compiler can check whether the mapping function is consistent with the rest of the program. Finally, the `for` and `update` instructions enable users to schedule finely the computation, potentially in parallel. Although InKS

`for` loops are similar to the C/Fortran ones, the `update` keyword is a bit different. In C/Fortran, a computation has two members on both sides of an assignment operator. The values read are on the right while the value written is on the left. Using InKS$_{\mathsf{PSO}}$ `update`, one expresses a set of values to write, and everything is inferred from the `map` instruction used earlier. Therefore, as it is done in C/Fortran, to achieve good performance using InKS$_{\mathsf{PSO}}$, one must mind memory layouts, `map` in InKS$_{\mathsf{PSO}}$, and scheduling, `for` in InKS$_{\mathsf{PSO}}$, to enable vectorization and a good cache use. For the Vlasov part, when performance is required, Table 6.6 shows that the number of lines needed to specify an InKS$_{\mathsf{PSO}}$ version of optimization choices is not greater than its Fortran counterpart.

Furthermore, InKS$_{\mathsf{PSO}}$ proposes the `update auto` to automatically schedule larger sets of computations, especially for "non-critical" parts of the code. This eases the writing of optimization choices, as shown on the Table 6.6, while making it possible to tune the computation scheduling when necessary. As shown in Listing 6.9, in our 6D Vlasov-Poisson experiments, we have let the compiler schedule the Poisson part, which accounts for less than 1% of the execution time. In comparison to the reference, presented in Listing 6.8, our approach greatly improves the readability of the optimization choices, without introducing performance penalties. At the same time, we have described precisely the schedule of the Vlasov part to match the efficiency of the reference version, as shown in Table 6.4. Moreover, the use of the Polyhedral model and this permanent link between algorithm optimization choices make bug detection possible. Indeed, our compiler can detect errors at compile-time, such as the use of uninitialized values, and generates code to detect others at runtime, such as an out-of-bounds access. Note that these could be detect at compile-time, thanks to the Polyhedral model. In terms of software engineering, the InKS model, through the separation of concerns, also encourages users to adopt the best practice approach of optimizing the code incrementally. We applied this approach to develop Listing 6.7: each version extends the set of optimization choices developed in the previous versions with a new one. Moreover, the use of the automatic approach of the InKS$_{\mathsf{C}}$ compiler ensures we find more easily any bug we could introduce with a new set of optimizations.

In terms of efficiency, the use of the InKS programming model with its optimization language does not imply any drawbacks. Tables 6.4 and 6.5 show that all InKS$_{\mathsf{PSO}}$ versions match or surpass the reference versions. This includes simple case such as the 1D heat equation, in which there is no notable difference between InKS$_{\mathsf{PSO}}$ and plain C++ versions, but also real world application like the 6D Vlasov-Poisson solver. This application exhibits a clear advantage for the InKS$_{\mathsf{PSO}}$ version. Thanks to the knowledge of the algorithm by the compiler, the generated C++ code is enhanced with various directives that latter help the C++ compiler automatically generates a more efficient executable. This proves, to some extent, that it is possible to use the InKS programming model and achieve great performance and even go beyond efficiency offered by traditional approaches.

6.4 Conclusion

In the previous chapter, we have presented four approaches to take into account optimization choices in the InKS programming model: the automatic compiler, InKS$_{\mathsf{Loop}}$, InKS$_{\mathsf{XMP}}$ and InKS$_{\mathsf{PSO}}$. One in particular enables its users to express many kinds of optimization choices: InKS$_{\mathsf{PSO}}$. In this chapter, we have evaluated this approach in terms of efficiency, generality, productivity gain and usability on three codes: the 1D heat equation solved by the finite difference method, the NAS parallel benchmarks and the 6D Vlasov-Poisson system, in comparison to Fortran. This evaluation showed that the InKS$_{\mathsf{PSO}}$ approach:

1. matches or improves performance in comparison to traditional approaches;
2. is general enough to express complex optimization choices on real applications;
3. improves productivity on non-critical parts of the code by relying on an automatic compiler;
4. comes with well-defined concepts that are not more complex than the ones existing in C or Fortran.

Still, InKS$_{\mathsf{PSO}}$ suffers from several limitations. At this time, it does not implement a dedicated interface for MPI. Although it could be possible to extend InKS$_{\mathsf{PSO}}$, by relying on PGAS concepts, as mentioned in Section 5.6. However, MPI is still usable in two ways with InKS$_{\mathsf{PSO}}$. Firstly, the InKS model can be applied on a subset of an application, relying on other, unmodified parts of that application to handle the distributed computing algorithmics. Secondly, at the moment, we consider the algorithm written in InKS$_{\mathsf{PIA}}$ as applying at the node level. Thus, one can write InKS$_{\mathsf{PIA}}$ operations that call MPI routines and update parts of logical arrays. Indeed, as mentioned in Section 4.4, even though it is not advised to use complex InKS$_{\mathsf{PIA}}$ operations – since it may limit the range of possible optimizations – it is supported to manage specific patterns (*e.g.* converging loop, runtime dependence) or to use libraries, such as MPI. The same analysis holds for any libraries and tools as well, including IOs. For instance, in the Poisson part of the 6D Vlasov-Poisson solver, we used the *fftw* library inside the InKS$_{\mathsf{PIA}}$ operations to handle the Fourier transforms.

Also, note that more complex InKS$_{\mathsf{PIA}}$ operations may be irregular. Processing irregular operations in parallel is challenging: the common loop-splitting parallelism strategies may not offer great performance as each iteration do not need the same amount of computations. In this situation, the OpenMP task paradigm would be useful in order to balance these operations among the available resources or to overlap computations with communications. In Appendix C, we presented a preliminary work to support this paradigm in InKS$_{\mathsf{PSO}}$.

Another limitation lies in data structures. Currently, we transparently support array of plain old data, *i.e.* scalar types or structures with, in particular, no user-defined constructor. For instance, it is not possible to express arrays of arrays, such as done in the `MG` NAS kernel.

Chapter 7

Conclusion

7.1 Contribution

In numerical simulation codes, performance, productivity and readability appear to be antagonistic goals. Although, focusing on plain performance impedes readability, efficiency is at the core of most simulation codes as it enables fast computations on volumes of data that no person could complete in a lifetime. As a result, scientists and engineers can solve scientific problems and better understand the laws of physics and their interactions. Still, readability is essential to stimulate the cooperation between domain scientists and optimization specialists and to favor the maintainability of large codes. Hence, application developers must carefully separate the code in a multitude of functions, each dedicated to a specific role, related either to complex optimization implementations or to plain mathematical solving. Currently, both readability and efficiency may be reached, but productivity will suffer. Indeed, architectures widely differ from one another, it is necessary to implement as many versions as there are architectures. This leads to the repetition of pieces of information that remain the same between each version, such as domains and halos sizes or loop bounds. In the end, it negatively impacts productivity and increases the risk of errors. This situation is even more concerning as newest architectures become more and more complex and varied.

Scientists around the world have proposed various approaches to alleviate this problem and ease efficient code writing while minimizing readability

issues. These solutions propose to hide the complexity of all or part of optimization choices by relying on APIs, language extensions, runtimes or dedicated languages and compilers. Although these approaches are fundamental, as they ease complex optimization expression and improve readability, they may not be enough to separate rigorously domain science and optimizations while avoiding information repetition. This is especially true when porting a code to bleeding edge architectures for which the best optimization patterns are in the process of being identified and not yet encoded in such approaches.

In this thesis, we proposed a novel approach to provide efficiency, productivity and readability in numerical simulation codes: the InKS programming model. The model can express all or parts of simulations which are static control parts; that is to say, programs which the structure (*e.g.* loop bounds, array sizes) is parameterized by integer values known at execution. It proposes two distinct languages to separate the expression of the algorithm and optimization choices. Relying on two languages enables to separate the concerns both physically, in distinct set of files, and semantically, with notions and concepts adapted to the aspects the users need to express, in the end improving readability and maintainability. The first one is the InKS_{PIA} language which enables domain scientists to express the simulation algorithm independently of any concern for optimization choices. The second language is InKS_{PSO} to derive, from a specific InKS_{PIA} code, optimization choices only. As the algorithm stays the same, information does not need to be repeated between each optimization versions, improving productivity.

We have defined and implemented the InKS_{PIA} language. The simulation algorithm expression must be complete while not limiting the possibilities of future optimizations. Taking into account these prerequisites, we proposed the InKS_{PIA} language as a declarative language which consists of logical arrays and fine grain operations with logical data dependencies. This strategy leads to a separation of logical and physical memory spaces while limiting over-constraining scheduling possibilities, enabling domain scientists to express algorithm with no concerns of performance. In a second step, optimization specialists can choose the best mapping between

these two memory spaces and the choice of a scheduling, in accordance with the targeted architecture. As InKS$_{\mathsf{PIA}}$ is at the foundation of the InKS approach, it is essential for it to be self-sufficient. Therefore, we also demonstrated that it contains all information required for code generation.

An algorithm language, alone, has only a limited use if there are no ways to set the optimization choices, *i.e.* memory placement and operations scheduling. Therefore, we have also proposed four approaches to take into account such choices. A first one, completely automated, relies on a compiler. Although it does not deliver the best possible performance, it can be used to test the validity of the algorithm description. The existence of this compiler supports the fact that InKS$_{\mathsf{PIA}}$ contains all necessary information. We have then proposed two experimental languages to express a specific set of optimization choices: nested loops expression and logical domain decomposition. Finally, we have proposed the InKS$_{\mathsf{PSO}}$ language. InKS$_{\mathsf{PSO}}$ enables application developers to accurately express all aspects of optimization choices while relying on information contained in the InKS$_{\mathsf{PIA}}$ code it refers. As expected by the InKS programming model, this strategy enables us to provide a language which limits information repetition while offering good performance and capable of expressing parallel concerns, using *OpenMP* parallel constructs. In addition, we have implemented and described our InKS$_{\mathsf{PIA}}$/InKS$_{\mathsf{PSO}}$ compiler.

Finally, we have evaluated the InKS approach on a real-world application: the 6D Vlasov-Poisson system. This system simulates the dynamics and the interactions of particles in a plasma. Therefore, it relies on complex mathematics and physics, but also on elaborate optimization choices. This part especially includes parallel non-trivial loop ordering and memory layouts to expose vectorization and improve cache uses. To evaluate our approach, we have implemented the 6D Vlasov-Poisson algorithm using InKS$_{\mathsf{PIA}}$ and its complex optimization choices using InKS$_{\mathsf{PSO}}$ and compared both to a reference version, written in Fortran in the SeLaLib library. This evaluation demonstrates that the InKS programming model fulfills all three objectives. Firstly, the use of a InKS$_{\mathsf{PIA}}$ code as a shared foundation for all InKS$_{\mathsf{PSO}}$ optimization choices versions avoids information repetition. Secondly, it matches or even improves efficiency compared

to traditional approaches, thanks to the knowledge the algorithm provides to our compiler. And finally, distinct dedicated languages improves readability, enabling domain scientists and optimization specialists to focus on their specialty.

7.2 Perspective and future works

In this thesis, we have proposed, implemented and evaluated the InKS programming model. Even though we have demonstrated the ability of the model to fully separate algorithmic and optimization concerns while maintaining the performance requirements, a novel programming model needs several years, if not decades, to reach the desired maturity level for widespread use. InKS and our implementation of the model do not escape the rule and therefore, we propose several possible enhancements.

Firstly, improvements to the algorithm language can be made. A first improvement is mentioned in Section 4.4: InKS$_{\mathsf{PIA}}$ may propose predefined basic operations, such as the addition of two logical array cells or the copy from one to another. This would benefit the language in two ways. First, it eases the algorithm expression and its readability, as the data dependencies are implied by the operation. Then, it gives more meaning to the compiler which can later optimize better the code. For instance, depending on the memory layout, copy operations of boundaries from a time-step to another may not be necessary and could be eliminated, if only the compiler could know these operations were copies.

Another improvement may come in enabling nested InKS$_{\mathsf{PIA}}$ code. Real-world applications often rely on multiple interdependent parts and it could be more convenient to have an InKS$_{\mathsf{PIA}}$ code for each of them and a main InKS$_{\mathsf{PIA}}$ code that invokes them all. Moreover, it would ease code reusability.

Secondly, we have only explored the surface of the InKS$_{\mathsf{PSO}}$ capabilities. While in traditional approach users actual objectives are mixed with optimization choices, we separate them and are able to understand their nature. With InKS$_{\mathsf{PIA}}$, giving all dependencies and data information, as well as the actual operation information, with the improvements we men-

tioned in the previous paragraph, the InKS$_{\mathsf{PSO}}$ compiler is capable of doing much more than what it is doing today. For instance, our InKS implementation currently supports the automatic addition of vectorization directives as well as the scheduling of non-critical applications parts. We would like to go further in this approach to also add alignment information to the C++ compiler or generate automatic memory layouts for non-critical arrays. In addition, auto-tuning could be achieved with the InKS approach. For the user, it consists in submitting numerous memory layouts and scheduling. Then the compiler, thanks to the Polyhedral model, could detect the pairs that are consistent from the one that are not, run them all and detect which strategy fit the best to a given architecture.

Then, InKS$_{\mathsf{PSO}}$ can be upgraded to support more optimization choices. One of the most trivial would be to add the call to external Polyhedral tool, such as Pluto, which automatically reorders nested loops to expose both data locality and parallelism. Other approaches, similar to the one we have tried with OpenMP task, could be to handle distributed memory environment, through the use of PGAS model for example, or kernel offloading to GPU, using tools such as Kokkos. Moreover, the InKS languages lack of conditional statements. This could be supported in the form of conditions on the values of the invariant program parameters, already supported by the Polyhedral model. For instance, this could be useful to choose between an optimization or another depending on the size of the grid in a given dimension. Finally, adding more InKS$_{\mathsf{PSO}}$ `for` options is worth considering. The goal would be to provide options for classical loops optimization, such as blocking, as done in XFor.

We hope for a further development and use of the InKS programming model. However, proposing not one, but two novel languages will presumably limit its popularity, in comparison to C or Fortran. The burden placed on developers' shoulder can be alleviated by proposing a source-to-source compiler capable of generating an InKS$_{\mathsf{PIA}}$ algorithm from a C or Fortran code; in the event that code is compatible with the Polyhedral model. Similarly, it could generate the core of an InKS$_{\mathsf{PSO}}$ code from the InKS$_{\mathsf{PIA}}$ specification.

In addition, we could greatly improve the InKS compiler by logging

choices that were made, giving more information about the issues it faces during the compilation phases or by simply reducing the complexity of its algorithms, especially by minimizing the size of the time vector. Indeed, the compilation time increases rapidly way with the size of this vector.

A last line of improvements we have identified lies in the foundation of the programming model. We intensively used the Polyhedral model, as it enables us various and simple analysis. However, it comes with some limitations, as mentioned in Section 3.5. It could be interesting to test another approach or limit its usage. For instance, currently, memory layouts in InKS$_{\mathsf{PSO}}$ must be described as affine relations, directly translated into the Polyhedral model. We could imagine using more expressive C function to map logical to physical memory cells. However, its validity would be at the users' expense.

Appendix A

Complete InKS$_{\mathsf{PIA}}$ implementation of the 6D Vlasov-Poisson solver.

```
#HEADER (C)

#ifdef __cplusplus
  #include <cmath>
  #include <cstring>
  #include <cstdlib>
#else
  #include <math.h>
  #include <string.h>
  #include <stdlib.h>
#endif
#include <fftw3.h>
typedef fftw_complex* fftw_complex_ptr;

const double inv_6 = 1.0/6.0;
#define MY_MAX(a, b) ((a) > (b) ? (a) : (b))

#END

op compute_Energy(t, MAX_DIM1, MAX_DIM2, MAX_DIM3) : (
  double ex {in: i=[0:MAX_DIM1[ j=[0:MAX_DIM2[ k=[0:MAX_DIM3[ (i, j, k, t)},
  double ey {in: i=[0:MAX_DIM1[ j=[0:MAX_DIM2[ k=[0:MAX_DIM3[ (i, j, k, t)},
  double ez {in: i=[0:MAX_DIM1[ j=[0:MAX_DIM2[ k=[0:MAX_DIM3[ (i, j, k, t)},
  double delta_eta {in: (0); (1); (2)},
  double Es {out: (t)}
)
#CODE (C)
```

```
28 double sum = 0;
29 for(int k=0; k<MAX_DIM3; k++)
30   for(int j=0; j<MAX_DIM2; j++)
31     for(int i=0; i<MAX_DIM1; i++)
32       sum += ex(i, j, k, t)*ex(i, j, k, t) + ey(i, j, k, t)*ey(i, j, k, t) +
             ez(i, j, k, t)*ez(i, j, k, t);
33 Es(t) = sqrt(sum*delta_eta(0)*delta_eta(1)*delta_eta(2));
34 #END
35
36 op compute_mass(t, MAX_DIM1, MAX_DIM2, MAX_DIM3) : (
37   double rho {in: i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ k=[0: MAX_DIM3[ (i, j, k,
         t)},
38   double mass {out: (t)},
39   double volume_eta123 {in}
40 )
41 #CODE (C)
42 double sum = 0;
43 for(int i=0; i<MAX_DIM1; i++)
44   for(int j=0; j<MAX_DIM2; j++)
45     for(int k=0; k<MAX_DIM3; k++)
46       sum += rho(i, j, k, t);
47 mass(t) = sum * volume_eta123;
48 #END
49
50 op compute_charge_density(t, MAX_DIM1, MAX_DIM2, MAX_DIM3, MAX_DIM4,
       MAX_DIM5, MAX_DIM6, step, shift1, shift2, shift3, shift4, shift5,
       shift6) : (
51   double rho {out: (0:MAX_DIM1, 0:MAX_DIM2, 0:MAX_DIM3, t)},
52   double f6d {in: (shift1:MAX_DIM1+shift1, shift2:MAX_DIM2+shift2,
         shift3:MAX_DIM3+shift3,
53             shift4:MAX_DIM4+shift4, shift5:MAX_DIM5+shift5,
                 shift6:MAX_DIM6+shift6, t, step)},
54   double volume_eta456 {in}
55 )
56 #CODE (C)
57   #ifndef USE_OMP
58 double* sum = (double*)malloc(MAX_DIM1*sizeof(double));
59 for(int k = 0; k<MAX_DIM3; ++k){
60   for(int j = 0; j<MAX_DIM2; ++j){
61     memset(sum, 0, sizeof(double) * MAX_DIM1);
62     for(int n = 0; n<MAX_DIM6; n++){
63       for(int m = 0; m<MAX_DIM5; m++){
64         for(int l = 0; l<MAX_DIM4; l++){
65           for(int i = 0; i<MAX_DIM1; ++i){
66             sum[i] += f6d(i+shift1, j+shift2, k+shift3, l+shift4, m+shift5,
                   n+shift6, t, step);
67           }
68         }
69       }
```

```
70      }
71      #pragma ivdep
72      for(int i = 0; i<MAX_DIM1; ++i){
73        rho(i, j, k, t) = (sum[i] * volume_eta456) - 1.0;
74      }
75    }
76  }
77  free(sum);
78  #else
79  double* sum;
80         #pragma omp parallel private(sum)
81         {
82       sum = (double*)malloc(MAX_DIM1*sizeof(double));
83       #pragma omp for collapse(2) schedule(static)
84       for(int k = 0; k<MAX_DIM3; ++k){
85         for(int j = 0; j<MAX_DIM2; ++j){
86           memset(sum, 0, sizeof(double) * MAX_DIM1);
87           for(int n = 0; n<MAX_DIM6; n++){
88             for(int m = 0; m<MAX_DIM5; m++){
89               for(int l = 0; l<MAX_DIM4; l++){
90                 for(int i = 0; i<MAX_DIM1; ++i){
91                   sum[i] += f6d(i+shift1, j+shift2, k+shift3, l+shift4,
                          m+shift5, n+shift6, t, step);
92                 }
93               }
94             }
95           }
96           #pragma ivdep
97           for(int i = 0; i<MAX_DIM1; ++i){
98             rho(i, j, k, t) = (sum[i] * volume_eta456) - 1.0;
99           }
100        }
101      }
102      free(sum);
103    }
104 #endif
105 #END
106
107 /*_____________POISSON*/
108 op init_fftw_buffer(MAX_DIM1, MAX_DIM2, MAX_DIM3) : (
109   fftw_complex_ptr fftw_in {out},
110   fftw_complex_ptr fftw_out {out}
111 )
112 #CODE (C)
113   size_t N =  MY_MAX(MY_MAX(MAX_DIM1, MAX_DIM2), MAX_DIM3);
114   fftw_in =  fftw_alloc_complex(N);
115   fftw_out =  fftw_alloc_complex(N);
116 #END
117
```

```
118 op init_fftw_plan(MAX_DIMn) : (
119   fftw_complex_ptr fftw_in {in},
120   fftw_complex_ptr fftw_out {in},
121   fftw_plan p  {out},
122   fftw_plan p_inv {out}
123 )
124 #CODE (C)
125   p =  fftw_plan_dft_1d(MAX_DIMn, fftw_in, fftw_out, FFTW_FORWARD,
          FFTW_ESTIMATE);
126   p_inv = fftw_plan_dft_1d(MAX_DIMn, fftw_in, fftw_out, FFTW_BACKWARD,
          FFTW_ESTIMATE);
127 #END
128
129 op fftw_x(j, k, t, MAX_DIM1) : (
130   fftw_complex_ptr fftw_in {in},
131   fftw_complex_ptr fftw_out {in},
132   fftw_plan px    {in},
133   double rho     {in: i=[0:MAX_DIM1[ (i, j, k, t)},
134   fftw_complex hat_rho  {out: i=[0:MAX_DIM1[ (i, j, k, t, 0)}
135 )
136 #CODE (C)
137   for(int i=0; i<MAX_DIM1; i++){
138     fftw_in[i][0] = rho(i, j, k, t);
139     fftw_in[i][1] = 0.0;
140   }
141   fftw_execute(px);
142   for(int i=0; i<MAX_DIM1; i++){
143     (hat_rho(i, j, k, t, 0))[0] = fftw_out[i][0];
144     (hat_rho(i, j, k, t, 0))[1] = fftw_out[i][1];
145   }
146 #END
147
148 op fftw_y(i, k, t, MAX_DIM2) : (
149   fftw_complex_ptr fftw_in {in},
150   fftw_complex_ptr fftw_out {in},
151   fftw_plan py    {in},
152   fftw_complex hat_rho  {in: j=[0:MAX_DIM2[ (i, j, k, t, 0) | out:
          j=[0:MAX_DIM2[ (i, j, k, t, 1)}
153 )
154 #CODE (C)
155   for(int j=0; j<MAX_DIM2; j++){
156     fftw_in[j][0] = (hat_rho(i, j, k, t, 0))[0];
157     fftw_in[j][1] = (hat_rho(i, j, k, t, 0))[1];
158   }
159   fftw_execute(py);
160   for(int j=0; j<MAX_DIM2; j++){
161     (hat_rho(i, j, k, t, 1))[0] = fftw_out[j][0];
162     (hat_rho(i, j, k, t, 1))[1] = fftw_out[j][1];
163   }
```

```
#END

op fftw_z(i, j, t, MAX_DIM1, MAX_DIM2, MAX_DIM3) : (
  fftw_complex_ptr fftw_in {in},
  fftw_complex_ptr fftw_out {in},
  fftw_plan pz    {in},
  fftw_complex hat_rho  {in: k=[0:MAX_DIM3[ (i, j, k, t, 1) | out:
      k=[0:MAX_DIM3[ (i, j, k, t, 2)}
)
#CODE (C)
  double normalisation = 1.0 / ((double)(MAX_DIM1 * MAX_DIM2 * MAX_DIM3));
  for(int k=0; k<MAX_DIM3; k++){
    fftw_in[k][0] = (hat_rho(i, j, k, t, 1))[0];
    fftw_in[k][1] = (hat_rho(i, j, k, t, 1))[1];
  }
  fftw_execute(pz);
  for(int k=0; k<MAX_DIM3; k++){
    (hat_rho(i, j, k, t, 2))[0] = fftw_out[k][0] * normalisation;
    (hat_rho(i, j, k, t, 2))[1] = fftw_out[k][1] * normalisation;
  }
#END

op compute_hat_phi(i, j, k, t, MAX_DIM1, MAX_DIM2, MAX_DIM3) : (
  double length    {in: (0:3)},
  fftw_complex hat_rho {in: (i, j, k, t, 2)},
  fftw_complex hat_phi {out: (i, j, k, t, 0)}
)
#CODE (C)
  int ind_x, ind_y, ind_z;
  double kx, ky, kz;
  const double kx0 = 2*M_PI/length(0);
  const double ky0 = 2*M_PI/length(1);
  const double kz0 = 2*M_PI/length(2);
  if(i==0 && j==0 && k==0){
    (hat_phi(0, 0, 0, t, 0))[0] = 0.0;
    (hat_phi(0, 0, 0, t, 0))[1] = 0.0;
  }else{
    ind_x = i < MAX_DIM1/2.0 ? i : MAX_DIM1 - i;
    ind_y = j < MAX_DIM2/2.0 ? j : MAX_DIM2 - j;
    ind_z = k < MAX_DIM3/2.0 ? k : MAX_DIM3 - k;

    kx = kx0 * (double)ind_x;
    ky = ky0 * (double)ind_y;
    kz = kz0 * (double)ind_z;

    (hat_phi(i, j, k, t, 0))[0] = (hat_rho(i, j, k, t, 2))[0] / (kx*kx +
        ky*ky + kz*kz);
    (hat_phi(i, j, k, t, 0))[1] = (hat_rho(i, j, k, t, 2))[1] / (kx*kx +
        ky*ky + kz*kz);
```

```
210   }
211 #END
212
213 op fftw_inv_z(i, j, t, MAX_DIM3) : (
214   fftw_complex_ptr fftw_in {in},
215   fftw_complex_ptr fftw_out {in},
216   fftw_plan pz_inv   {in},
217   fftw_complex hat_phi  {in: k=[0:MAX_DIM3[ (i, j, k, t, 0) | out:
          k=[0:MAX_DIM3[ (i, j, k, t, 1)}
218 )
219 #CODE (C)
220   for(int k=0; k<MAX_DIM3; k++){
221     fftw_in[k][0] = (hat_phi(i, j, k, t, 0))[0];
222     fftw_in[k][1] = (hat_phi(i, j, k, t, 0))[1];
223   }
224   fftw_execute(pz_inv);
225   for(int k=0; k<MAX_DIM3; k++){
226     (hat_phi(i, j, k, t, 1))[0] = fftw_out[k][0];
227     (hat_phi(i, j, k, t, 1))[1] = fftw_out[k][1];
228   }
229 #END
230
231 op fftw_inv_y(i, k, t, MAX_DIM2) : (
232   fftw_complex_ptr fftw_in {in},
233   fftw_complex_ptr fftw_out {in},
234   fftw_plan py_inv   {in},
235   fftw_complex hat_phi  {in: j=[0:MAX_DIM2[ (i, j, k, t, 1) | out:
          j=[0:MAX_DIM2[ (i, j, k, t, 2)}
236 )
237 #CODE (C)
238   for(int j=0; j<MAX_DIM2; j++){
239     fftw_in[j][0] = (hat_phi(i, j, k, t, 1))[0];
240     fftw_in[j][1] = (hat_phi(i, j, k, t, 1))[1];
241   }
242   fftw_execute(py_inv);
243   for(int j=0; j<MAX_DIM2; j++){
244     (hat_phi(i, j, k, t, 2))[0] = fftw_out[j][0];
245     (hat_phi(i, j, k, t, 2))[1] = fftw_out[j][1];
246   }
247 #END
248
249 op fftw_inv_x(j, k, t, MAX_DIM1) : (
250   fftw_complex_ptr fftw_in {in},
251   fftw_complex_ptr fftw_out {in},
252   fftw_plan px_inv   {in},
253   fftw_complex hat_phi  {in: i=[0:MAX_DIM1[ (i, j, k, t, 2)},
254   fftw_complex phi   {out: i=[0:MAX_DIM1[ (i, j, k, t)}
255 )
256 #CODE (C)
```

```
257   for(int i=0; i<MAX_DIM1; i++){
258     fftw_in[i][0] = (hat_phi(i, j, k, t, 2))[0];
259     fftw_in[i][1] = (hat_phi(i, j, k, t, 2))[1];
260   }
261   fftw_execute(px_inv);
262   for(int i=0; i<MAX_DIM1; i++){
263     (phi(i, j, k, t))[0] = fftw_out[i][0];
264     (phi(i, j, k, t))[1] = 0.0;
265   }
266 #END
267
268 op compute_ex_from_phi(j, k, t, MAX_DIM1) : (
269   double length    {in: (0)},
270   fftw_complex_ptr fftw_in {in},
271   fftw_complex_ptr fftw_out {in},
272   fftw_plan px    {in},
273   fftw_plan px_inv   {in},
274   fftw_complex phi   {in: i=[0:MAX_DIM1[ (i, j, k, t)},
275   double ex     {out: i=[0:MAX_DIM1[ (i, j, k, t)}
276 )
277 #CODE (C)
278   double norm_fac = 1.0/MAX_DIM1;
279   double kx0 = 2*M_PI/length(0);
280
281   for(int i=0; i<MAX_DIM1; i++){
282     fftw_in[i][0] = (phi(i, j, k, t))[0];
283     fftw_in[i][1] = (phi(i, j, k, t))[1];
284   }
285   fftw_execute(px);
286   for(int i=0; i<MAX_DIM1/2; i++){
287     fftw_in[i][0] = - fftw_out[i][1] * -kx0*i * norm_fac;
288     fftw_in[i][1] = fftw_out[i][0] * -kx0*i * norm_fac;
289   }
290   for(int i=MAX_DIM1/2; i<MAX_DIM1; i++){
291     fftw_in[i][0] = - fftw_out[i][1] * kx0*(MAX_DIM1-i) * norm_fac;
292     fftw_in[i][1] = fftw_out[i][0] * kx0*(MAX_DIM1-i) * norm_fac;
293   }
294   fftw_execute(px_inv);
295   for(int i=0; i<MAX_DIM1; i++)
296     ex(i, j, k, t) = fftw_out[i][0];
297 #END
298
299 op compute_ey_from_phi(i, k, t, MAX_DIM2) : (
300   double length    {in: (1)},
301   fftw_complex_ptr fftw_in {in},
302   fftw_complex_ptr fftw_out {in},
303   fftw_plan py    {in},
304   fftw_plan py_inv   {in},
305   fftw_complex phi   {in: j=[0:MAX_DIM2[ (i, j, k, t)},
```

```
306   double ey     {out: j=[0:MAX_DIM2[ (i, j, k, t)}
307 )
308 #CODE (C)
309   double norm_fac = 1.0/MAX_DIM2;
310   double ky0 = 2*M_PI/length(1);
311
312   for(int j=0; j<MAX_DIM2; j++){
313     fftw_in[j][0] = (phi(i, j, k, t))[0];
314     fftw_in[j][1] = (phi(i, j, k, t))[1];
315   }
316   fftw_execute(py);
317   for(int j=0; j<MAX_DIM2/2; j++){
318     fftw_in[j][0] = - fftw_out[j][1] * -ky0*j * norm_fac;
319     fftw_in[j][1] = fftw_out[j][0] * -ky0*j * norm_fac;
320   }
321   for(int j=MAX_DIM2/2; j<MAX_DIM2; j++){
322     fftw_in[j][0] = - fftw_out[j][1] * ky0*(MAX_DIM2-j) * norm_fac;
323     fftw_in[j][1] = fftw_out[j][0] * ky0*(MAX_DIM2-j) * norm_fac;
324   }
325   fftw_execute(py_inv);
326   for(int j=0; j<MAX_DIM2; j++)
327     ey(i, j, k, t) = fftw_out[j][0];
328 #END
329
330 op compute_ez_from_phi(i, j, t, MAX_DIM3) : (
331   double length    {in: (2)},
332   fftw_complex_ptr fftw_in {in},
333   fftw_complex_ptr fftw_out {in},
334   fftw_plan pz     {in},
335   fftw_plan pz_inv   {in},
336   fftw_complex phi   {in: k=[0:MAX_DIM3[ (i, j, k, t)},
337   double ez     {out: k=[0:MAX_DIM3[ (i, j, k, t)}
338 )
339 #CODE (C)
340   double norm_fac = 1.0/MAX_DIM3;
341   double kz0 = 2*M_PI/length(2);
342
343   for(int k=0; k<MAX_DIM3; k++){
344     fftw_in[k][0] = (phi(i, j, k, t))[0];
345     fftw_in[k][1] = (phi(i, j, k, t))[1];
346   }
347   fftw_execute(pz);
348   for(int k=0; k<MAX_DIM3/2; k++){
349     fftw_in[k][0] = - fftw_out[k][1] * -kz0*k * norm_fac;
350     fftw_in[k][1] = fftw_out[k][0] * -kz0*k * norm_fac;
351   }
352   for(int k=MAX_DIM3/2; k<MAX_DIM3; k++){
353     fftw_in[k][0] = - fftw_out[k][1] * kz0*(MAX_DIM3-k) * norm_fac;
354     fftw_in[k][1] = fftw_out[k][0] * kz0*(MAX_DIM3-k) * norm_fac;
```

```
  }
  fftw_execute(pz_inv);
  for(int k=0; k<MAX_DIM3; k++)
    ez(i, j, k, t) = fftw_out[k][0];
#END

/*_____________END POISSON*/

op compute_pV(i, j, k, t, dim, shift1, shift2, shift3) : (
  double e   {in: (i, j, k, t)},
  double p_v   {out: coef=[0:3[ (i+shift1, j+shift2, k+shift3, t, coef)},
  double delta_eta {in: (dim)},
  double delta_t  {in}
)
#CODE (C)
double coef = -e(i, j, k, t)*delta_t/delta_eta(dim);
p_v(i+shift1, j+shift2, k+shift3, t, 0) = coef*(coef-1.0)*0.5;
p_v(i+shift1, j+shift2, k+shift3, t, 1) = (1.0 - coef*coef);
p_v(i+shift1, j+shift2, k+shift3, t, 2) = coef*(coef+1.0)*0.5;
#END


op compute_pi_pq(i, shiftV) : (
  double disp_eta {in: (i)},
  int pi {out: (i+shiftV)},
  double pq {out: k=[0:4[ (i+shiftV, k)}
)
#CODE (C)
  double p = disp_eta(i);
  pi(i+shiftV) = (int)floor(p);
  double coef = p - (double)pi(i+shiftV);
  pq(i+shiftV, 0) = (-coef*(coef-1.0)*(coef-2.0)*inv_6);
  pq(i+shiftV, 1) = ((coef*coef-1.0)*(coef-2.0)*0.5);
  pq(i+shiftV, 2) = (-coef*(coef+1.0)*(coef-2.0)*0.5);
  pq(i+shiftV, 3) = (coef*(coef*coef-1.0)*inv_6);
#END

/*Centered lagrange*/
op advection_eta1(i, j, k, l, m, n, t, MAX_DIM1, step, shift1) : (
  double f6d {in: x=[0:MAX_DIM1+2*shift1[ (x, j, k, l, m, n, t-1, 5) |
        out: (i, j, k, l, m, n, t, step)},
  int pi_v1 {in: (l)},
  double pq_v1 {in: x=[0:4[ (l, x)}
)
#CODE (C)
f6d(i, j, k, l, m, n, t, step) =
    pq_v1(l, 0) * f6d(i-1+pi_v1(l), j, k, l, m, n, t-1, 5)
  + pq_v1(l, 1) * f6d(i+0+pi_v1(l), j, k, l, m, n, t-1, 5)
  + pq_v1(l, 2) * f6d(i+1+pi_v1(l), j, k, l, m, n, t-1, 5)
```

```
404   + pq_v1(l, 3) * f6d(i+2+pi_v1(l), j, k, l, m, n, t-1, 5);
405 #END
406
407 op advection_eta2(i, j, k, l, m, n, t, MAX_DIM2, step, shift2) : (
408   double f6d {in: y=[0:MAX_DIM2+2*shift2[ (i, y, k, l, m, n, t, step-1) |
409         out: (i, j, k, l, m, n, t, step)},
410   int pi_v2 {in: (m)},
411   double pq_v2 {in: x=[0:4[ (m, x)}
412 )
413 #CODE (C)
414 f6d(i, j, k, l, m, n, t, step) =
415     pq_v2(m, 0) * f6d(i, j-1+pi_v2(m), k, l, m, n, t, step-1)
416   + pq_v2(m, 1) * f6d(i, j+0+pi_v2(m), k, l, m, n, t, step-1)
417   + pq_v2(m, 2) * f6d(i, j+1+pi_v2(m), k, l, m, n, t, step-1)
418   + pq_v2(m, 3) * f6d(i, j+2+pi_v2(m), k, l, m, n, t, step-1);
419 #END
420
421 op advection_eta3(i, j, k, l, m, n, t, MAX_DIM3, step, shift3) : (
422   double f6d {in: z=[0:MAX_DIM3+2*shift3[ (i, j, z, l, m, n, t, step-1) |
423         out: (i, j, k, l, m, n, t, step)},
424   int pi_v3 {in: (n)},
425   double pq_v3 {in: x=[0:4[ (n, x)}
426 )
427 #CODE (C)
428 f6d(i, j, k, l, m, n, t, step) =
429     pq_v3(n, 0) * f6d(i, j, k-1+pi_v3(n), l, m, n, t, step-1)
430   + pq_v3(n, 1) * f6d(i, j, k+0+pi_v3(n), l, m, n, t, step-1)
431   + pq_v3(n, 2) * f6d(i, j, k+1+pi_v3(n), l, m, n, t, step-1)
432   + pq_v3(n, 3) * f6d(i, j, k+2+pi_v3(n), l, m, n, t, step-1);
433 #END
434
435 op advection_eta4(i, j, k, l, m, n, t, MAX_DIM4, step, shift1, shift2,
      shift3, shift4, shift5, shift6) : (
436   double f6d {in: v1=[0:MAX_DIM4+2*shift4[ (i, j, k, v1, m, n, t, step-1) |
437         out: (i, j, k, l, m, n, t, step)},
438   double p_v1{in: coef=[0:3[ (i, j, k, t, coef)}
439 )
440 #CODE (C)
441 f6d(i, j, k, l, m, n, t, step) =
442     p_v1(i, j, k, t, 0) * f6d(i, j, k, l-1, m, n, t, step-1)
443   + p_v1(i, j, k, t, 1) * f6d(i, j, k, l+0, m, n, t, step-1)
444   + p_v1(i, j, k, t, 2) * f6d(i, j, k, l+1, m, n, t, step-1);
445 #END
446 op advection_eta5(i, j, k, l, m, n, t, MAX_DIM5, step, shift1, shift2,
      shift3, shift4, shift5, shift6) : (
447   double f6d {in: v2=[0:MAX_DIM5+2*shift5[ (i, j, k, l, v2, n, t, step-1) |
448         out: (i, j, k, l, m, n, t, step)},
449   double p_v2 {in: coef=[0:3[ (i, j, k, t, coef)}
450 )
```

```
#CODE (C)
f6d(i, j, k, l, m, n, t, step) =
    p_v2(i, j, k, t, 0) * f6d(i, j, k, l, m-1, n, t, step-1)
  + p_v2(i, j, k, t, 1) * f6d(i, j, k, l, m+0, n, t, step-1)
  + p_v2(i, j, k, t, 2) * f6d(i, j, k, l, m+1, n, t, step-1);
#END

op advection_eta6(i, j, k, l, m, n, t, MAX_DIM6, step, shift1, shift2,
    shift3, shift4, shift5, shift6) : (
  double f6d {in: v3=[0:MAX_DIM6+2*shift6[ (i, j, k, l, m, v3, t, step-1) |
       out: (i, j, k, l, m, n, t, step)},
  double p_v3 {in: coef=[0:3[ (i, j, k, t, coef)}
)
#CODE (C)
f6d(i, j, k, l, m, n, t, step) =
    p_v3(i, j, k, t, 0) * f6d(i, j, k, l, m, n-1, t, step-1)
  + p_v3(i, j, k, t, 1) * f6d(i, j, k, l, m, n+0, t, step-1)
  + p_v3(i, j, k, t, 2) * f6d(i, j, k, l, m, n+1, t, step-1);
#END

/*
  Copy1 -> ADV1 -> Copy2 -> ADV2 -> Copy3 -> ADV3 -> Copy4 -> ADV4 -> Copy5
      -> ADV6 -> Copy6 -> ADV6
*/

op copy_left_right_1(j, k, l, m, n, t, MAX_DIM1, step, shift1) : (
  double f6d {in: (shift1:MAX_DIM1+shift1, j, k, l, m, n, t, step) |
       out: (0:MAX_DIM1+2*shift1, j, k, l, m, n, t, step)}
)
#CODE (C)
  #pragma ivdep
  for(int i=0; i<shift1; i++)
    f6d(i, j, k, l, m, n, t, step) = f6d(shift1+MAX_DIM1-shift1+i, j, k, l,
        m, n, t, step);
  #pragma ivdep
  for(int i=0; i<shift1; i++)
    f6d(MAX_DIM1+shift1+i, j, k, l, m, n, t, step) = f6d(shift1+i, j, k, l,
        m, n, t, step);
#END

op copy_left_right_2(i, k, l, m, n, t, MAX_DIM2, step, shift2) : (
  double f6d {in: (i, shift2:MAX_DIM2+shift2, k, l, m, n, t, step) |
       out: (i, 0:MAX_DIM2+2*shift2, k, l, m, n, t, step)}
)
#CODE (C)
  #pragma ivdep
  for(int j=0; j<shift2; j++)
    f6d(i, j, k, l, m, n, t, step) = f6d(i, shift2+MAX_DIM2-shift2+j, k, l,
        m, n, t, step);
```

```
495   #pragma ivdep
496   for(int j=0; j<shift2; j++)
497     f6d(i, MAX_DIM2+shift2+j, k, l, m, n, t, step) = f6d(i, shift2+j, k, l,
          m, n, t, step);
498 #END
499
500 op copy_left_right_3(i, j, l, m, n, t, MAX_DIM3, step, shift3) : (
501   double f6d {in: (i, j, shift3:MAX_DIM3+shift3, l, m, n, t, step) |
502         out: (i, j, 0:MAX_DIM3+2*shift3, l, m, n, t, step)}
503 )
504 #CODE (C)
505   #pragma ivdep
506   for(int k=0; k<shift3; k++)
507     f6d(i, j, k, l, m, n, t, step) = f6d(i, j, shift3+MAX_DIM3-shift3+k, l,
          m, n, t, step);
508   #pragma ivdep
509   for(int k=0; k<shift3; k++)
510     f6d(i, j, MAX_DIM3+shift3+k, l, m, n, t, step) = f6d(i, j, shift3+k, l,
          m, n, t, step);
511 #END
512
513 op copy_left_right_4(i, j, k, m, n, t, MAX_DIM4, step, shift4) : (
514   double f6d {in: (i, j, k, shift4:MAX_DIM4+shift4, m, n, t, step) |
515         out: (i, j, k, 0:MAX_DIM4+2*shift4, m, n, t, step)}
516 )
517 #CODE (C)
518   #pragma ivdep
519   for(int l=0; l<shift4; l++)
520     f6d(i, j, k, l, m, n, t, step) = f6d(i, j, k, shift4+MAX_DIM4-shift4+l,
          m, n, t, step);
521   #pragma ivdep
522   for(int l=0; l<shift4; l++)
523     f6d(i, j, k, MAX_DIM4+shift4+l, m, n, t, step) = f6d(i, j, k, shift4+l,
          m, n, t, step);
524 #END
525
526 op copy_left_right_5(i, j, k, l, n, t, MAX_DIM5, step, shift5) : (
527   double f6d {in: (i, j, k, l, shift5:MAX_DIM5+shift5, n, t, step) |
528         out: (i, j, k, l, 0:MAX_DIM5+2*shift5, n, t, step)}
529 )
530 #CODE (C)
531   #pragma ivdep
532   for(int m=0; m<shift5; m++)
533     f6d(i, j, k, l, m, n, t, step) = f6d(i, j, k, l,
          shift5+MAX_DIM5-shift5+m, n, t, step);
534   #pragma ivdep
535   for(int m=0; m<shift5; m++)
536     f6d(i, j, k, l, MAX_DIM5+shift5+m, n, t, step) = f6d(i, j, k, l,
          shift5+m, n, t, step);
```

```
#END

op copy_left_right_6(i, j, k, l, m, t, MAX_DIM6, step, shift6) : (
  double f6d {in: (i, j, k, l, m, shift6:MAX_DIM6+shift6, t, step) |
       out: (i, j, k, l, m, 0:MAX_DIM6+2*shift6, t, step)}
)
#CODE (C)
  #pragma ivdep
  for(int n=0; n<shift6; n++)
    f6d(i, j, k, l, m, n, t, step) = f6d(i, j, k, l, m,
        shift6+MAX_DIM6-shift6+n, t, step);
  #pragma ivdep
  for(int n=0; n<shift6; n++)
    f6d(i, j, k, l, m, MAX_DIM6+shift6+n, t, step) = f6d(i, j, k, l, m,
        shift6+n, t, step);
#END


op init_f6d(i, j, k, l, m, n, MX_D1, MX_D2, MX_D3, MX_D4, MX_D5, MX_D6,
    shift1, shift2, shift3, shift4, shift5, shift6) : (
  double f6d   {out: (i, j, k, l, m, n, 0, 2) },
  double factor  {in},
  double alpha   {in},
  double kx    {in : size=[0:3[ (size)},
  double tensor_1  {in : size=[0:MX_D1[ (size)},
  double tensor_2  {in : size=[0:MX_D2[ (size)},
  double tensor_3  {in : size=[0:MX_D3[ (size)},
  double tensor_4  {in : size=[0:MX_D4[ (size)},
  double tensor_5  {in : size=[0:MX_D5[ (size)},
  double tensor_6  {in : size=[0:MX_D6[ (size)}
)
#CODE (C)
  /* Initialization sll_s_initialize_landau_prod_6d */
  f6d(i, j, k, l, m, n, 0, 2) = factor*(1.0 + alpha *
      cos(kx(0)*tensor_1(i-shift1))*
      cos(kx(1)*tensor_2(j-shift1))*
      cos(kx(2)*tensor_3(k-shift1))
    ) *
    exp(-0.5*((tensor_4(l-shift4) * tensor_4(l-shift4)) +
         (tensor_5(m-shift4) * tensor_5(m-shift4)) +
         (tensor_6(n-shift4) * tensor_6(n-shift4)))
    );
#END

op fill_tensor(MX_D, n) : (
  double eta_min  {in: (n)},
  double delta_eta {in: (n)},
  double tensor_n  {out: size=[0:MX_D[ (size)}
)
```

```
#CODE (C)
  for(int i=0; i<MX_D; i++)
    tensor_n(i) = eta_min(n) + delta_eta(n) * i;
#END

op compute_init_factor() : (
  double factor {out}
)
#CODE (C)
  factor = 1.0/(pow(sqrt(2.0*M_PI), 3)); //*v_thermal = 1.0
#END

op compute_half_delta_t() : (
  double half_delta_t  {out},
  double delta_t   {in}
)
#CODE (C)
  half_delta_t = 0.5*delta_t;
#END

simulation inks_vlasov_poisson6d(MAX_DIM1, MAX_DIM2, MAX_DIM3, MAX_DIM4,
    MAX_DIM5, MAX_DIM6, n_iterations,
                shift1, shift2, shift3, shift4, shift5, shift6) : (
  double rho(4),
  double f6d(8),
  double ex(4),
  double ey(4),
  double ez(4),
  double delta_eta(1) {in: step=[0:6[ (step)},
  double disp_eta1(1) {in: size=[0:MAX_DIM4[ (size)},
  double disp_eta2(1) {in: size=[0:MAX_DIM5[ (size)},
  double disp_eta3(1) {in: size=[0:MAX_DIM6[ (size)},
  double length(1) {in: size=[0:6[ (size)},
  double kx(1)  {in: size=[0:3[ (size)},
  double Es(1)   {out: t=[0:n_iterations[ (t)},
  double eta_min(1) {in: size=[0:6[ (size)},
  double mass(1){out: t=[0:n_iterations[ (t)},
  double volume_eta123 {in},
  double volume_eta456 {in},
  double delta_t {in},
  double alpha {in},
  int pi_v1(1),
  int pi_v2(1),
  int pi_v3(1),
  double pq_v1(2),
  double pq_v2(2),
  double pq_v3(2),
  double p_v1(5),
  double p_v2(5),
```

```
631   double p_v3(5),
632   double init_factor,
633   double tensor_1(1),
634   double tensor_2(1),
635   double tensor_3(1),
636   double tensor_4(1),
637   double tensor_5(1),
638   double tensor_6(1),
639   double half_delta_t,
640
641   /*For Poisson*/
642   fftw_complex_ptr fftw_in,
643   fftw_complex_ptr fftw_out,
644   fftw_plan px,
645   fftw_plan py,
646   fftw_plan pz,
647   fftw_plan px_inv,
648   fftw_plan py_inv,
649   fftw_plan pz_inv,
650   fftw_complex phi(4),
651   fftw_complex hat_phi(5),
652   fftw_complex hat_rho(5)
653 )
654
655 #RESTRICT
656   shift1 > 0 and shift2 > 0 and shift3 > 0 and shift4 > 0 and shift5 > 0
          and shift6 > 0;
657
658 #CODE (inks)
659   compute_init_factor () : (init_factor),
660   compute_half_delta_t() : (half_delta_t, delta_t),
661
662   fill_tensor(MAX_DIM1, 0) : (eta_min, delta_eta, tensor_1),
663   fill_tensor(MAX_DIM2, 1) : (eta_min, delta_eta, tensor_2),
664   fill_tensor(MAX_DIM3, 2) : (eta_min, delta_eta, tensor_3),
665   fill_tensor(MAX_DIM4, 3) : (eta_min, delta_eta, tensor_4),
666   fill_tensor(MAX_DIM5, 4) : (eta_min, delta_eta, tensor_5),
667   fill_tensor(MAX_DIM6, 5) : (eta_min, delta_eta, tensor_6),
668
669   init_f6d x1=[shift1:MAX_DIM1+shift1[ x2=[shift2:MAX_DIM2+shift2[
          x3=[shift3:MAX_DIM3+shift3[
670         v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
              v3=[shift6:MAX_DIM6+shift6[
671         (x1, x2, x3, v1, v2, v3, MAX_DIM1, MAX_DIM2, MAX_DIM3, MAX_DIM4,
              MAX_DIM5, MAX_DIM6, shift1, shift2, shift3, shift4, shift5,
              shift6)
672         : (f6d, init_factor, alpha, kx, tensor_1, tensor_2, tensor_3,
              tensor_4, tensor_5, tensor_6),
673
```

```
674  advection_eta4 x1=[shift1:MAX_DIM1+shift1[ x2=[shift2:MAX_DIM2+shift2[
         x3=[shift3:MAX_DIM3+shift3[
675         v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
676         t=[0: n_iterations[
677         (x1, x2, x3, v1, v2, v3, t, MAX_DIM4, 3, shift1, shift2, shift3,
                shift4, shift5, shift6) :
678         (f6d, p_v1),
679  advection_eta5 x1=[shift1:MAX_DIM1+shift1[ x2=[shift2:MAX_DIM2+shift2[
         x3=[shift3:MAX_DIM3+shift3[
680         v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
681         t=[0: n_iterations[
682         (x1, x2, x3, v1, v2, v3, t, MAX_DIM5, 4, shift1, shift2, shift3,
                shift4, shift5, shift6) :
683         (f6d, p_v2),
684  advection_eta6 x1=[shift1:MAX_DIM1+shift1[ x2=[shift2:MAX_DIM2+shift2[
         x3=[shift3:MAX_DIM3+shift3[
685         v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
686         t=[0: n_iterations[
687         (x1, x2, x3, v1, v2, v3, t, MAX_DIM6, 5, shift1, shift2, shift3,
                shift4, shift5, shift6) :
688         (f6d, p_v3),
689  advection_eta1 x1=[shift1:MAX_DIM1+shift1[ x2=[shift2:MAX_DIM2+shift2[
         x3=[shift3:MAX_DIM3+shift3[
690         v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
691         t=[1: n_iterations[
692         (x1, x2, x3, v1, v2, v3, t, MAX_DIM1, 0, shift1) :
693         (f6d, pi_v1, pq_v1),
694  advection_eta2 x1=[shift1:MAX_DIM1+shift1[ x2=[shift2:MAX_DIM2+shift2[
         x3=[shift3:MAX_DIM3+shift3[
695         v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
696         t=[1: n_iterations[
697         (x1, x2, x3, v1, v2, v3, t, MAX_DIM2, 1, shift2) :
698         (f6d, pi_v2, pq_v2),
699  advection_eta3 x1=[shift1:MAX_DIM1+shift1[ x2=[shift2:MAX_DIM2+shift2[
         x3=[shift3:MAX_DIM3+shift3[
700         v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
701         t=[1: n_iterations[
702         (x1, x2, x3, v1, v2, v3, t, MAX_DIM3, 2, shift3) :
703         (f6d, pi_v3, pq_v3),
704
705  copy_left_right_1  x1=[shift1:MAX_DIM1+shift1[
         x2=[shift2:MAX_DIM2+shift2[ x3=[shift3:MAX_DIM3+shift3[
706           v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
```

```
                v3=[shift6:MAX_DIM6+shift6[
707           t=[0:n_iterations[
708           (x2, x3, v1, v2, v3, t, MAX_DIM1, 5, shift1) :
709           (f6d),
710   copy_left_right_2  x1=[shift1:MAX_DIM1+shift1[
          x2=[shift2:MAX_DIM2+shift2[ x3=[shift3:MAX_DIM3+shift3[
711           v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
712           t=[0:n_iterations[
713           (x1, x3, v1, v2, v3, t, MAX_DIM2, 0, shift2) :
714           (f6d),
715   copy_left_right_3  x1=[shift1:MAX_DIM1+shift1[
          x2=[shift2:MAX_DIM2+shift2[ x3=[shift3:MAX_DIM3+shift3[
716           v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
717           t=[0:n_iterations[
718           (x1, x2, v1, v2, v3, t, MAX_DIM3, 1, shift3) :
719           (f6d),
720   copy_left_right_4  x1=[shift1:MAX_DIM1+shift1[
          x2=[shift2:MAX_DIM2+shift2[ x3=[shift3:MAX_DIM3+shift3[
721           v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
722           t=[0:n_iterations[
723           (x1, x2, x3, v2, v3, t, MAX_DIM4, 2, shift4) :
724           (f6d),
725   copy_left_right_5  x1=[shift1:MAX_DIM1+shift1[
          x2=[shift2:MAX_DIM2+shift2[ x3=[shift3:MAX_DIM3+shift3[
726           v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
727           t=[0:n_iterations[
728           (x1, x2, x3, v1, v3, t, MAX_DIM5, 3, shift5) :
729           (f6d),
730   copy_left_right_6  x1=[shift1:MAX_DIM1+shift1[
          x2=[shift2:MAX_DIM2+shift2[ x3=[shift3:MAX_DIM3+shift3[
731           v1=[shift4:MAX_DIM4+shift4[ v2=[shift5:MAX_DIM5+shift5[
                v3=[shift6:MAX_DIM6+shift6[
732           t=[0:n_iterations[
733           (x1, x2, x3, v1, v2, t, MAX_DIM6, 4, shift6) :
734           (f6d),
735
736   compute_pi_pq l=[0:MAX_DIM4[ (l, shift4) :
737          (disp_eta1, pi_v1, pq_v1),
738   compute_pi_pq m=[0:MAX_DIM5[ (m, shift4) :
739          (disp_eta2, pi_v2, pq_v2),
740   compute_pi_pq n=[0:MAX_DIM6[ (n, shift4) :
741          (disp_eta3, pi_v3, pq_v3),
742 /*Delta t*/
743   compute_pV i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ k=[0: MAX_DIM3[
744           t=[1: n_iterations[ (i, j, k, t, 3, shift1, shift2, shift3) :
```

```
745         (ex, p_v1, delta_eta, delta_t),
746   compute_pV i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ k=[0: MAX_DIM3[
747         t=[1: n_iterations[ (i, j, k, t, 4, shift1, shift2, shift3) :
748         (ey, p_v2, delta_eta, delta_t),
749   compute_pV i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ k=[0: MAX_DIM3[
750         t=[1: n_iterations[ (i, j, k, t, 5, shift1, shift2, shift3) :
751         (ez, p_v3, delta_eta, delta_t),
752
753 /*Half Delta t*/
754   compute_pV i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ k=[0: MAX_DIM3[
755         (i, j, k, 0, 3, shift1, shift2, shift3) :
756         (ex, p_v1, delta_eta, half_delta_t),
757   compute_pV i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ k=[0: MAX_DIM3[
758         (i, j, k, 0, 4, shift1, shift2, shift3) :
759         (ey, p_v2, delta_eta, half_delta_t),
760   compute_pV i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ k=[0: MAX_DIM3[
761         (i, j, k, 0, 5, shift1, shift2, shift3) :
762         (ez, p_v3, delta_eta, half_delta_t),
763
764   compute_charge_density t=[0: n_iterations[
765     (t, MAX_DIM1, MAX_DIM2, MAX_DIM3, MAX_DIM4, MAX_DIM5, MAX_DIM6, 2,
          shift1, shift2, shift3, shift4, shift5, shift6) :
766     (rho, f6d, volume_eta456),
767   /*Poisson*/
768   init_fftw_buffer(MAX_DIM1, MAX_DIM2, MAX_DIM3) :
769     (fftw_in, fftw_out),
770   init_fftw_plan(MAX_DIM1) :
771     (fftw_in, fftw_out, px, px_inv),
772   init_fftw_plan(MAX_DIM2) :
773     (fftw_in, fftw_out, py, py_inv),
774   init_fftw_plan(MAX_DIM3) :
775     (fftw_in, fftw_out, pz, pz_inv),
776
777   fftw_x j=[0:MAX_DIM2[ k=[0:MAX_DIM3[ t=[0: n_iterations[
778     (j, k, t, MAX_DIM1) :
779     (fftw_in, fftw_out, px, rho, hat_rho),
780   fftw_y i=[0: MAX_DIM1[ k=[0:MAX_DIM3[ t=[0: n_iterations[
781     (i, k, t, MAX_DIM2) :
782     (fftw_in, fftw_out, py, hat_rho),
783   fftw_z i=[0: MAX_DIM1[ j=[0:MAX_DIM2[ t=[0: n_iterations[
784     (i, j, t, MAX_DIM1, MAX_DIM2, MAX_DIM3) :
785     (fftw_in, fftw_out, pz, hat_rho),
786   compute_hat_phi i=[0: MAX_DIM1[ j=[0:MAX_DIM2[ k=[0: MAX_DIM3[ t=[0:
        n_iterations[
787     (i, j, k, t, MAX_DIM1, MAX_DIM2, MAX_DIM3) :
788     (length, hat_rho, hat_phi),
789   fftw_inv_z i=[0: MAX_DIM1[ j=[0: MAX_DIM2[ t=[0: n_iterations[
790     (i, j, t, MAX_DIM3) :
791     (fftw_in, fftw_out, pz_inv, hat_phi),
```

```
792   fftw_inv_y i=[0: MAX_DIM1[ k=[0: MAX_DIM3[ t=[0: n_iterations[
793     (i, k, t, MAX_DIM2) :
794     (fftw_in, fftw_out, py_inv, hat_phi),
795   fftw_inv_x j=[0: MAX_DIM2[ k=[0: MAX_DIM3[ t=[0: n_iterations[
796     (j, k, t, MAX_DIM1) :
797     (fftw_in, fftw_out, px_inv, hat_phi, phi),
798   compute_ex_from_phi j=[0: MAX_DIM2[ k=[0: MAX_DIM3[ t=[0: n_iterations[
799     (j, k, t, MAX_DIM1) :
800     (length, fftw_in, fftw_out, px, px_inv, phi, ex),
801   compute_ey_from_phi i=[0: MAX_DIM1[ k=[0: MAX_DIM3[ t=[0: n_iterations[
802     (i, k, t, MAX_DIM2) :
803     (length, fftw_in, fftw_out, py, py_inv, phi, ey),
804   compute_ez_from_phi j=[0: MAX_DIM2[ i=[0: MAX_DIM1[ t=[0: n_iterations[
805     (i, j, t, MAX_DIM3) :
806     (length, fftw_in, fftw_out, pz, pz_inv, phi, ez),
807
808   compute_mass t=[0: n_iterations[ (t, MAX_DIM1, MAX_DIM2, MAX_DIM3) :
809     (rho, mass, volume_eta123),
810   compute_Energy t=[0: n_iterations[
811     (t, MAX_DIM1, MAX_DIM2, MAX_DIM3) :
812     (ex, ey, ez, delta_eta, Es)
813 #END
```

Appendix B

Complete example of the C++ code generated by the InKS compiler for an `alloc` and `map` InKS$_{PSO}$ statements.

```
1  //*** InKSpso ***//
2  inks_function(..., Z, Y, X){
3    //Allocation of a 3D buffers
4    alloc double A(Z, Y, X)
5    alloc double B(Z, Y, X)
6
7    //Map to a 3D logical array
8    map i=[0:X[ j=[0:Y[ k=[0:Z[ A(k, j, i) = logical(0, k, j, i)
9    map i=[0:X[ j=[0:Y[ k=[0:Z[ B(k, j, i) = logical(1, k, j, i)
10
11   //Update statement using the logical array
12   update logical(0:2, 0:Z, 0:Y, 0:X) using op
13 }
14
15 //*** C++ code generated ***/
16
17 //Class definition for the mapping
18 class logical_1_t{
19     const int& Z, & Y, &X;
20     double* const& A;
21     double* const& B;
22     const size_t &A_mult_1;
23     const size_t &A_mult_2;
```

```
24      const size_t &B_mult_1;
25      const size_t &B_mult_2;
26
27  logical_1_t(const int& Z_param, const int& Y_param, const int& X_param,
        double* const& A_param, double* const& B_param, const size_t& A_mult_1,
        const size_t& A_mult_2, const size_t& B_mult_1, const size_t& B_mult_2)
        :
28    Z(Z_param),
29    Y(Y_param),
30    X(X_param),
31    A(A_param),
32    B(B_param),
33    A_mult_1(A_mult_1_param),
34    A_mult_2(A_mult_2_param),
35    B_mult_1(B_mult_1_param),
36    B_mult_2(B_mult_2_param),
37  {}
38
39  //In commentary, the original non-optimized conditions
40  double& operator()(const int& t,const int& k,const int& j,const int&
        i)const{
41    //if(t==0 && 0<=k<Z && 0<=j<Y && 0<=i<X)
42    if(t==0)
43      return A[k*A_mult_2 + j*A_mult_1 + i];
44    //if(t==1 && 0<=k<Z && 0<=j<Y && 0<=i<X)
45    if(t==1)
46      return B[k*B_mult_2 + j*B_mult_1 + i];
47  } };
48
49  void inks_function(..., Z, Y, X){
50    //Allocations
51    double* A = new double[Z*Y*X];
52    size_t A_mult_2 = Y*X;
53    size_t A_mult_1 = X;
54
55    double* B = new double[Z*Y*X];
56    size_t B_mult_2 = Y*X;
57    size_t B_mult_1 = X;
58
59    //Mapping initialization
60    logical_1_t logical_1(Z, Y, X, A, B, A_mult_1, A_mult_2, B_mult_1,
          B_mult_2);
61
62    //Operation in a loop nest, using the logical structure
63    for(int k=0; k<Z; k++)
64      for(int j=0; j<Z; j++)
65        for(int i=0; i<X; i++)
66          op(logical_1, t, k, j, i)
67  }
```

Appendix C

InKS$_{\mathsf{PSO}}$ and the OpenMP Task paradigm

As mentioned in the Section 5.4, the current implementation of InKS$_{\mathsf{PSO}}$ supports loop parallelism through a keyword located on the nested loops that should be executed in parallel. This results in the generation of `parallel for` OpenMP directives before the targeted loops. However, this approach introduces many synchronizations at runtime that can hinder application performance, especially problematic at large scale. This is especially true with the recent shift of HPC platforms from multi-core to many-core architectures. Task-based approaches are promising to address this problem [Song et al., 2009, Broquedis et al., 2012]. The overall idea is to split computation parts into small dependent units of work scheduled on available resources such as computing cores.

Since OpenMP 3 [OpenMP Architecture Review Board, 2008], the API offers to express tasks, blocks of codes, that can be executed in parallel. It consists in encapsulating a block of code in an `omp task`. This construct was enhanced with the 4^{th} version of the OpenMP standard and now enables users to specify that a task reads data from previously scheduled tasks and writes data that may be read by future tasks. Such specification is complex, verbose and error-prone. Moreover, there are costs for handling the tasks dependencies [Richard et al., 2019a]. Indeed, the OpenMP runtime, that executes the tasks in accordance with the dependencies, con-

sumes more and more memory and computing power as the number of data dependencies submitted by the program increases. Therefore, to take advantage of the task paradigm of OpenMP, we have to minimize the number of data dependencies used during the program execution. With OpenMP 4.5 [OpenMP Architecture Review Board, 2015], a new way to target the task paradigm by involving a *for loop*, marked as an `omp taskloop`. This directive expresses that each iteration can be divided into tasks. There are no dependency relation between such tasks.

This section presents the fruit of a collaborative effort involving Jérôme Richard, an expert in the task paradigm of the OpenMP API [Aumage et al., 2017, Bigot et al., 2018, Richard et al., 2019b, Richard, 2017], with the aim of supporting this paradigm in the INKS$_{\mathsf{PSO}}$ approach. The scope of the work is reduced to the following points:

1. No support of task-recursion: the code within tasks does not submit other tasks;
2. Tasks are applied to loops only and more especially only one loop of nested ones, but multiple independent loops can run in parallel;
3. Each task must have a statically bounded number of dependencies (limitation inherited from OpenMP);
4. Setting the task granularity is left to the user (it may be done using additional enclosing loops).

In comparison to the OpenMP 4.5 *taskloop*, we propose a modified *taskloop* construct that comes with data dependencies. This would enable users to specify that each iteration of a loop can be divided into tasks while specifying dependencies between these tasks. Our approach consists in adding two keywords: one to define where tasks must be used and one to put synchronization points. The task-based keyword called `OMP_TASKS` is added to nested loops in a similar way to the existing keyword based on `parallel for`. Hence, it is up to the developer of the INKS$_{\mathsf{PSO}}$ code to set which loops should contain tasks. The annotation will submit a task for each iteration of the loop through the directive `omp task`. Using the

InKS approach, and more specifically the InKS$_{\text{PIA}}$ algorithm, the main goal is to automatically retrieve the minimal number of data dependencies according to the inputs and outputs of operations inside the tasks. If the number of dependencies is not statically bounded, the compilation fails and the problem is reported to the user. The synchronization keyword called `OMP_TASKWAIT` acts as a barrier: it specifies a point where all tasks previously submitted at runtime should be awaited. This is useful to drastically reduce the amount of dependency per tasks and for the interoperability between task-based parts of the code and those based on parallel loops.

C.0.1 Algorithm and Implementation

The algorithm is divided into four sequential dependent stages. Each one consists in iterating over the whole Abstract Syntax Tree (AST, generated by the Bison-based parser) of the InKS$_{\text{PSO}}$ code, using a top-down recursion (root to the leaves) followed by a bottom-up one (leaves to the root). When two AST elements (*e.g.* loops) lies in the same AST element (*e.g.* parent loop), stages are applied in-order (in the same order elements are written in the code). Those stages work on *isl* data structures and rely mainly on fields created during previous compilation phases with the assistance of the InKS$_{\text{PIA}}$ code: the relations associating a loop iteration to the data instance it reads or writes, respectively named R_I and R_O. The stages are the following:

1. Projection: transforms local in-loop relations into global ones;
2. Computation of dependencies: generate dependency relations so that previously executed iterations that write into data read by another iteration are dependent;
3. Section: perform the inverse operation of the Projection step on dependency sets so that global sets are transformed into local ones needed to generate OpenMP directives;
4. Code generation: generate OpenMP task-based directives for each loop.

INKS$_{\mathsf{PSO}}$ offers a *for loop* construct capable of specifying nested loops, using multiple indexes. Hence, the R_I and R_O relations associate an iteration, *i.e.* a specific value for all loops' index in the nested loop, to data instance. Since only one of those loops is marked as task loop, Stage 1 projects the inner indexes to obtain relations that associate a task, *i.e.* an iteration of the task loop, to the data it reads and writes. Moreover, the relation are parameterized by the surrounding loops. In order to compare the data accessed by all iterations, *i.e.* future tasks, we move these parameters to the relations' domain.

Stage 2 tracks data that will be read and written at runtime to statically define task dependencies. However, due to loops, the control flow is not linear, making this analysis complex. Indeed, one should handle the case where a synchronization point is put in the middle of a loop or when task dependencies cross loop iterations. In this case, a task of one iteration (*e.g.* time-step) can be dependent of tasks of the previous iteration (*e.g.* previous time-step), even though no synchronization point is executed in-between.

To solve this problem, Stage 2 constructs $PrevR_O$, the union of all previous output relation R_O (including the current loop). Then, it computes the dependencies through an application of the relation between each R_I and the reverse relation $PrevR_O$. This gives a new relation called *Deps*, associating a task that reads a piece of data to the task that produces it. It is important to note that *Deps* cannot be used directly to generate the OpenMP task-based directives since this operation needs to be able to reason about a given task (to compute and check some properties on each) and loop iterators are not yet parameters but variables of the *isl* relation. This is the goal of Stage 3.

Stage 3 first applies the reverse operation of Stage 1 on *Deps*: it sections (can be seen as an "un-projection") the *Deps* relation so that iterators are re-injected as parameters. This new parameterized relation, *LocalDeps*, is used to check if the number of dependencies is statically bounded per task and extract each of them. To check the number of dependencies, the *LocalDeps* is split into contiguous sub-relations. Then we compute the bounding box of the relation. If, and only if, the side of each dimension of the bounding box is a constant, we can be sure that there is a statically

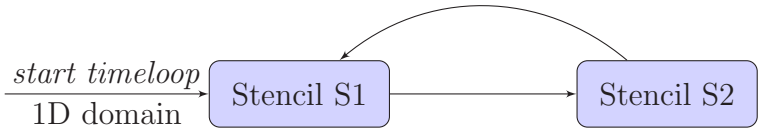


Figure C.1 – The double stencil application.

bounded number of dependencies. If the number of dependencies is not statically bounded the compilation stops and an error is reported. Otherwise, the lexicographically minimum value of *LocalDeps* is extracted from the relation iteratively until it becomes empty. The algorithm is proven to terminate because of the constant boundary.

Stage 4 generates the final OpenMP code. First of all, it computes the size of the dependency space of each loop to reserve a memory area for dependencies. Indeed, it is important to note that dependencies are memory storage location in OpenMP and memory area referenced by pointers should not need to be allocated according to the OpenMP documentation. Then, Stage 4 generates pointers for each dependency of each task before generating finally the `omp task` directive of each task.

C.0.2 Preliminary evaluation

To evaluate our current implementation of the Taskloop construct in INKS$_{\text{PSO}}$, we have developed a double stencil application. This application applies to a one-dimensional domain two stencils, one after the other, multiple times. Figure C.1 presents a graphical representation of this application's algorithm. After implementing an INKS$_{\text{PIA}}$ version of this algorithm, we have developed two INKS$_{\text{PSO}}$ optimization versions of this application. Both rely on a time loop containing two independent taskloops, each performing a stencil operation. The first version uses tasks that operate on a single coordinate while the second use coarser tasks, using a blocking strategy. All codes were compiled with Intel 18 compiler (`icpc` with `-O3 -xHost -ip -ipo` compilation options), the OpenMP Intel implementation and executed on a single node of the *Irene* cluster (TGCC, France), equipped with 192 GB RAM and two *Xeon Platinium 8168* CPUs per node. Table C.1 presents the result of this experiment.

Version	# cores	Execution time (second)
Sequential	1	1.53e−4 (±8.20%)
Fine task	1	1.23e−2 (±2.07%)
	2	6.58e−1 (±0.96%)
	4	6.63e−1 (±1.62%)
	8	7.09e−1 (±1.15%)
Coarse task	1	3.80e−3 (±8.44%)
	2	1.24e−1 (±1.06%)
	4	1.26e−1 (±2.04%)
	8	1.36e−1 (±4.22%)

Table C.1 – Strong scaling of the INKS$_{\mathsf{PSO}}$ taskloop construct on a double stencil application. Execution time of the INKS$_{\mathsf{PSO}}$ taskloop implementations of a double stencil application. Fine grain tasks compute a single element, coarser ones manage eight elements. Size=10000 with 10 time-steps. Median and maximal relative change of 10 executions.

Although the current implementation is experimental, the algorithm has been tested successfully on these two versions. However, the task-based parallelism strategy is clearly inefficient in our case, as shown on Table C.1. The task approach is interesting in the case of irregular or recursive problems. These often need load-balancing solution to run efficiently in parallel. In our case, the problem is regular, limiting the interest of the task approach. On the contrary, the task approach needs to compute the dependencies at runtime, inducing an overhead. This is illustrated on Table C.1. Indeed, the *fine-grain tasks* version is less efficient than the sequential version, even though it runs in parallel. Similarly, the *coarse-grain tasks* version is slower than the sequential version, probably because of the more complex dependence that must be computed at runtime. Still, this experiment demonstrates that the INKS$_{\mathsf{PSO}}$ compiler, assisted by the INKS$_{\mathsf{PIA}}$ algorithm, can generate automatically OpenMP task dependencies. Also, note that this paradigm may be interesting to execute in parallel INKS$_{\mathsf{PIA}}$ operations that need some balancing.

Currently, we have identified two limitations to our algorithm. Firstly, each task reads one or multiple storage locations (artificial data dependencies) and writes into a one unique storage location associated with a given

task. Since a lot of tasks can be generated at runtime, the same number of storage locations is generated too. However, some runtimes like the Clang/Intel implementation of OpenMP or KOMP accumulate those storage locations within their own internal data structures. As a result, the memory overhead can be significant compared to hand-written OpenMP code. Moreover, these data structures can grow so much that they cannot fit in the caches anymore, introducing additional runtime overheads. This is the reason behind such a small experiment: 10000 points is quite small, but increasing the number of points to 100000 causes the OpenMP versions to take minutes or even hours to run. This effect can be negated by reusing storage locations as much as possible. Hence, a task t can reuse a storage location d safely if, and only if, all tasks that read or write on d are executed before. Introducing synchronization points after having submitting a user-defined number of tasks is a possibility. This would let the runtime complete these tasks and make possible memory reuse, increasing dramatically the performance, especially in the case of larger runs.

Secondly, while tasks dependencies are automatically generated for each task, the number of dependencies is not currently minimized. Using too many dependencies can introduce additional runtime overheads. Hence, reducing the number of dependencies by analyzing the overall graph could improve performance at runtime (at the expense of a longer C++ compilation time).

Bibliography

[Anderson et al., 1990] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C., and Sorensen, D. (1990). LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press.

[Augonnet et al., 2011] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198.

[Aumage et al., 2017] Aumage, O., Bigot, J., Coullon, H., Pérez, C., and Richard, J. (2017). Combining Both a Component Model and a Task-Based Model for HPC Applications: A Feasibility Study on GYSELA. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 635–644.

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA.

[Backus, 1954] Backus, J. (1954). Preliminary report: specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN. Technical report.

[Bagnara et al., 2008] Bagnara, R., Hill, P. M., and Zaffanella, E. (2008). The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1):3 – 21. Special Issue on Second issue of experimental software and toolkits (EST).

[Bailey et al., 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

[Bastoul et al., 2012] Bastoul, C., Cohen, A., Girbal, S., Sharma, S., and Temam, O. (2012). Clan a polyhedral representation extractor for high level programs edition 1.0, for clan 0.7.0.

[Basu et al., 2017] Basu, P., Williams, S., Van Straalen, B., Oliker, L., Colella, P., and Hall, M. (2017). Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers. *Parallel Computing*, 64:50–64.

[Bauer et al., 2012] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Bigot et al., 2018] Bigot, J., Grandgirard, V., Latu, G., Méhaut, J.-F., Felipe Millani, L., Passeron, C., Quinito Masnada, S., Richard, J., and Videau, B. (2018). Building and Auto-Tuning Computing Kernels: Experimenting with Boast and Starpu in the Gysela Code. *ESAIM: Proceedings and Surveys*, 63:152–178.

[Bondhugula et al., 2008] Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. (2008). Automatic transformations for communication-minimized parallelization and locality optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

[Bosilca et al., 2013] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., and Dongarra, J. J. (2013). PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering*, 15(6):36–45.

[Broquedis et al., 2012] Broquedis, F., Gautier, T., and Danjean, V. (2012). LIBKOMP, an Efficient openMP Runtime System for Both Fork-join and Data Flow Paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 102–115, Berlin, Heidelberg. Springer-Verlag.

[Cardoso et al., 2012] Cardoso, J. a., Carvalho, T., Coutinho, J., Luk, W., Nobre, R., Diniz, P., and Petrov, Z. (2012). LARA: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 179–190.

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

[Christen et al., 2011] Christen, M., Schenk, O., and Burkhart, H. (2011). PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS) 2011*. IEEE.

[Cosnard and Jeannot, 1999] Cosnard, M. and Jeannot, E. (1999). Compact DAG Representation and Its Dynamic Scheduling. *Journal of Parallel and Distributed Computing*, 58(3):487 – 514.

[Cosnard and Loi, 1995] Cosnard, M. and Loi, M. (1995). Automatic task graph generation techniques. In *Proceedings of the twenty-eighth annual Hawaii international conference on system sciences*, volume 2, pages 113–122. IEEE.

[Danalis et al., 2014] Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2014). PTG: An Abstraction for Unhindered Parallelism. In *2014 Fourth International Workshop on Domain-Specific Lan-*

guages and High-Level Frameworks for High Performance Computing, pages 21–30.

[Darte et al., 2016] Darte, A., Isoard, A., and Yuki, T. (2016). Extended lattice-based memory allocation. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 218–228, New York, NY, USA. ACM.

[Darte et al., 2005] Darte, A., Schreiber, R., and Villard, G. (2005). Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257.

[Duran et al., 2011] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193.

[Edwards et al., 2014] Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[Ejjaaouani et al., 2017] Ejjaaouani, K., Aumage, O., Bigot, J., and Mehrenberger, M. (2017). InKS, a programming model to decouple performance from semantics in simulation codes. *hal*.

[Ejjaaouani et al., 2019a] Ejjaaouani, K., Aumage, O., Bigot, J., and Mehrenberger, M. (2019a). InKSpso, a language to express optimizations independently from the algorithm in InKS. Written, submitted to P3HPC (Supercomputing workshop).

[Ejjaaouani et al., 2018] Ejjaaouani, K., Aumage, O., Bigot, J., Mehrenberger, M., Murai, H., Nakao, M., and Sato, M. (2018). InKS, a programming model to decouple performance from algorithm in HPC codes. In *Repara 2018-4th International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms*, pages 1–12.

[Ejjaaouani et al., 2019b] Ejjaaouani, K., Aumage, O., Bigot, J., Mehrenberger, M., Murai, H., Nakao, M., and Sato, M. (2019b). InKS, a programming model to decouple performance from algorithm in HPC codes. *The Journal of Supercomputing*.

[Ernstsson et al., 2018] Ernstsson, A., Li, L., and Kessler, C. (2018). SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*, 46(1):62–80.

[Fassi and Clauss, 2015] Fassi, I. and Clauss, P. (2015). XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 100–109.

[Feautrier, 1991] Feautrier, P. (1991). Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53.

[Feautrier, 1996] Feautrier, P. (1996). Automatic Parallelization in the Polytope Model. In *The Data Parallel Programming Model*.

[Feautrier and Lengauer, 2011] Feautrier, P. and Lengauer, C. (2011). *Polyhedron Model*, pages 1581–1592. Springer US, Boston, MA.

[Fernández et al., 2014] Fernández, A., Beltran, V., Mateo, S., Patejko, T., and Ayguadé, E. (2014). A Data Flow Language to Develop High Performance Computing DSLs. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 11–20.

[Filbet et al., 2001] Filbet, F., Sonnendrücker, E., and Bertrand, P. (2001). Conservative Numerical Schemes for the Vlasov Equation. *Journal of Computational Physics*, 172(1):166 – 187.

[Griebler et al., 2018] Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018). Efficient NAS benchmark kernels with C++ parallel programming. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*.

[Harbulot and R. Gurd, 2006] Harbulot, B. and R. Gurd, J. (2006). A join point for loops in AspectJ. volume 2006, pages 63–74.

[Inria, IPP, IRMA, IRMAR, LJLL, 2018] Inria, IPP, IRMA, IRMAR, LJLL (2018). Semi Lagrangian Library: Modular library for the kinetic and gyrokinetic simulation of tokamak plasmas by the semi-lagrangian or particle-in-cell methods. `http://selalib.gforge.inria.fr/`. Accessed: 2018-04-10.

[Isoard, 2016] Isoard, A. (2016). *Extending Polyhedral Techniques towards Parallel Specifications and Approximations*. PhD thesis, École normale supérieure de Lyon. Thèse de doctorat dirigée par Darte, Alain Informatique Lyon 2016.

[Kaiser et al., 2019] Kaiser, H., aka wash, B. A. L., Heller, T., Bergé, A., Simberg, M., Biddiscombe, J., Bikineev, A., Mercer, G., Schäfer, A., Serio, A., Kwon, T., Huck, K., Habraken, J., Anderson, M., Copik, M., Brandt, S. R., Stumpf, M., Bourgeois, D., Blank, D., Jakobovits, S., Amatya, V., Viklund, L., Khatami, Z., Bacharwar, D., Yang, S., Schnetter, E., Diehl, P., Gupta, N., Wagle, B., and Christopher (2019). STEllAR-GROUP/hpx: HPX V1.2.1: The C++ Standards Library for Parallelism and Concurrency.

[Kamil, 2012] Kamil, S. (2012). StencilProbe: A microbenchmark for stencil applications. `http://people.csail.mit.edu/skamil/projects/stencilprobe/`. Accessed: 25-04-2019.

[Kelly et al., 1995] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D. (1995). The Omega library interface guide. Technical report, College Park, MD, USA.

[Kong et al., 2016] Kong, M., Pouchet, L.-N., Sadayappan, P., and Sarkar, V. (2016). PIPES: A Language and Compiler for Task-based Programming on Distributed-memory Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 39:1–39:12, Piscataway, NJ, USA. IEEE Press.

[Kormann et al., 2019] Kormann, K., Reuter, K., and Rampp, M. (2019). A massively parallel semi-lagrangian solver for the six-dimensional Vlasov-Poisson equation. *The International Journal of High Performance Computing Applications*.

[Lee and Sato, 2010] Lee, J. and Sato, M. (2010). Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *2010 39th International Conference on Parallel Processing Workshops*, pages 413–420.

[Lengauer, 1993] Lengauer, C. (1993). Loop Parallelization in the Polytope Model. In *CONCUR*.

[Loechner et al., 1999] Loechner, V., S. Brant, B., and Illkirch, F. (1999). PolyLib: A library for manipulating parameterized polyhedra.

[Mehrenberger et al., 2013] Mehrenberger, M., Steiner, C., Marradi, L., Crouseilles, N., Sonnendrucker, E., and Afeyan, B. (2013). Vlasov on GPU (VOG project). *ESAIM: Proc.*, 43:37–58.

[OpenMP Architecture Review Board, 2008] OpenMP Architecture Review Board (2008). OpenMP application program interface - version 3.0. `https://www.openmp.org/wp-content/uploads/spec30.pdf`.

[OpenMP Architecture Review Board, 2015] OpenMP Architecture Review Board (2015). OpenMP application programming interface - version 4.5. `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

[Pijanowski et al., 2014] Pijanowski, B. C., Tayyebi, A., Doucette, J., Pekin, B. K., Braun, D., and Plourde, J. (2014). A big data urban growth simulation at a national scale: Configuring the GIS and neural network based Land Transformation Model to run in a High Performance Computing (HPC) environment. *Environmental Modelling & Software*, 51:250 – 268.

[President's Information Technology Advisory Committee, 2005] President's Information Technology Advisory Committee (2005).

Computational science: Ensuring america's competitiveness. Report to the President. https://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf.

[Ragan-Kelley et al., 2017] Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. (2017). Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115.

[Richard, 2017] Richard, J. (2017). *Conception d'un modèle de composants logiciels avec ordonnancement de tâches pour les architectures parallèles multi-coeurs, application au code Gysela.* PhD thesis, École normale supérieure de Lyon.

[Richard et al., 2019a] Richard, J., Latu, G., and Bigot, J. (2019a). Using OpenMP tasks for a 2D Vlasov-Poisson Application. Research report, CEA Cadarache.

[Richard et al., 2019b] Richard, J., Latu, G., and Bigot, J. (2019b). Using OpenMP tasks for a 2D Vlasov-Poisson Application. Research report, CEA Cadarache.

[Ritchie, 1993] Ritchie, D. M. (1993). The Development of the C Language. *SIGPLAN Not.*, 28(3):201–208.

[Robbiano, 1985] Robbiano, L. (1985). Term orderings on the polynominal ring. volume 204, pages 513–517.

[Robison, 2013] Robison, A. D. (2013). Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 15(2):66.

[Sanderson and Curtin, 2016] Sanderson, C. and Curtin, R. (2016). Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1(2):26–32.

[Schaller, 1997] Schaller, R. R. (1997). Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59.

[Song et al., 2009] Song, F., YarKhan, A., and Dongarra, J. (2009). Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-memory Multicore Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA. ACM.

[Stansifer, 1984] Stansifer, R. (1984). Presburger's article on integer arithmetic: Remarks and translation. Technical report, Ithaca, NY, USA.

[Steuwer et al., 2017] Steuwer, M., Remmelg, T., and Dubach, C. (2017). LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85.

[Stroustrup, 1997] Stroustrup, B. (1997). *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition.

[Tang et al., 2011] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. (2011). The Pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA. ACM.

[Tanno and Iwasaki, 2009] Tanno, H. and Iwasaki, H. (2009). Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09. Springer.

[TGCC, 2019] TGCC (2019). The Irene cluster. `http://www-hpc.cea.fr/fr/complexe/tgcc-Irene.htm`.

[Thoman et al., 2015] Thoman, P., Gschwandtner, P., and Fahringer, T. (2015). On the quality of implementation of the C++ 11 thread support library. In *2015 23rd euromicro international conference on parallel, distributed, and network-based processing*, pages 94–98. IEEE.

[Vanbroekhoven, 2002] Vanbroekhoven, P. (2002). Dynamic Single Assignment. `http://www.elis.ugent.be/aces/edegem2002/vanbroekhoven.pdf`.

[Verdoolaege, 2007] Verdoolaege, S. (2007). barvinok: User guide. *Version 0.23), Electronically available at http://www. kotnet. org/skimo/barvinok*.

[Verdoolaege, 2010] Verdoolaege, S. (2010). isl: An integer set library for the Polyhedral Model. In *International Congress on Mathematical Software*, pages 299–302. Springer.

[Verdoolaege, 2014] Verdoolaege, S. (2014). ISCC calculator. `http://compsys-tools.ens-lyon.fr/iscc/index.php`.

[Verdoolaege and Grosser, 2012] Verdoolaege, S. and Grosser, T. (2012). Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, pages 1–16.

[Verdoolaege et al., 2013] Verdoolaege, S., Juega, J. C., Cohen, A., Gómez, J. I., Tenllado, C., and Catthoor, F. (2013). Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4):54:1–54:23.

[Verdoolaege et al., 2007] Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., and Bruynooghe, M. (2007). Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66.

[Xu et al., 2014] Xu, M., Liang, X.-Z., Samel, A., and Gao, W. (2014). MODIS consistent vegetation parameter specifications and their impacts on regional climate simulations. *Journal of Climate*, 27(22):8578–8596.

Université de Strasbourg

Ksander EJJAAOUANI
Le modèle de programmation InKS

msii ÉCOLE DOCTORALE

Résumé

Le modèle de programmation InKS vise à améliorer la lisibilité, la portabilité et la maintenabilité des codes de simulation tout en accroissant la productivité des développeurs de telles applications. Pour atteindre ces objectifs, InKS propose deux langages, chacun dédié à une partie de l'application. Le premier, $InKS_{PIA}$, permet d'exprimer les aspects algorithmiques d'un code de simulation scientifique tout en laissant les choix d'optimisation de côté. Il s'agit de décrire les fondations de la simulation : son algorithme. Le second langage, $InKS_{PSO}$, permet aux spécialistes de l'optimisation de réutiliser les informations contenues dans l'algorithme pour exprimer une large variété de choix d'optimisation. Le modèle permet d'écrire de nombreuses versions des optimisations, typiquement une par architecture, à partir d'un unique algorithme. En basant les différentes versions d'un programme sur sa partie invariante, l'algorithme, le modèle InKS limite la réécriture du code, boostant la productivité des développeurs.

Suite à la proposition du modèle InKS et de son implémentation, nous avons évalué le modèle au travers de l'implémentation de simulations de plasma: le système Vlasov-Poisson 6D. Cette évaluation a permis de mettre en évidence certaines bonnes propriétés du modèle, notamment en matière de séparation des aspects, mais aussi à montrer sa généralité et ses performances. Pour ce faire, nous avons comparé l'efficacité de deux implémentations du système Vlasov-Poisson 6D, écrite sur InKS ou Fortran. Pour les optimisations les plus importantes, le nombre de lignes dans les deux cas est similaire. Cependant, la présence de l'algorithme permet d'ordonnancer automatiquement, en une simple ligne, un large ensemble de calcul, verbeux et potentiellement sujet aux erreurs. Ainsi, dans les parties non critiques du code, $InKS_{PSO}$ peut résumer une partie du programme de manière concise, permettant de se concentrer sur les parties de calcul intensif. Par ailleurs, en limitant la réécriture du code aux seules optimisations, InKS permet de tester plusieurs stratégies d'optimisations afin d'identifier la plus adaptée à une architecture donnée. En comparaison, dans ces situations, les applications traditionnelles requièrent la réécriture d'une plus grande partie du code, limitant le temps consacré à la recherche des meilleures stratégies d'optimisation.

Mots-clés : Modèle de programmation, Séparation des préoccupations, Calcul haute performance

Résumé en anglais

The InKS programming model aims to improve readability portability and maintainability of simulation codes as well as boosting developer productivity. To fulfill these objectives, InKS proposes two languages, each dedicated to a specific concern. First, $InKS_{PIA}$ provides concepts to express simulation algorithms with no concerns for optimization. Once this foundation is set, $InKS_{PSO}$ enables optimization specialists to reuse the algorithm in order to specify the optimization part. The model offers to write numerous versions of the optimizations, typically one per architecture, from a single algorithm. This strategy limits the rewriting of code for each new optimization specification, boosting developer productivity.

We have evaluated the InKS programming model by using it to implement the 6D Vlasov-Poisson solver and compared our version with a Fortran one. This evaluation highlighted that, in addition to the separation of concerns, the InKS approach is not more complex that traditional ones while offering the same performance. Moreover, using the algorithm, it is able to generate valid code for non-critical parts of code, leaving optimization specialists more time to focus on optimizing the computation intensive parts.

Keywords: Programming model, Separation of concerns, High performance computing