Automatic code generation and optimization of multi-dimensional stencil computations on distributed-memory architectures

Mariem Saied

UNIVERSITÉ DE STRASBOURG

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur (MSII)

Laboratoire des Sciences de l'Ingénieur, de l'Informatique et de l'Imagerie (ICube)

THÈSE présentée par :

Mariem SAIED

soutenue le : **25 Septembre 2018**

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

Discipline/ Spécialité : Infomatique

Automatic Code Generation and Optimization of Multi-dimensional Stencil Computations on Distributed-Memory Architectures

THÈSE dirigée par :

M. GUSTEDT Jens	Directeur de Recherche à Inria, Strasbourg, France

THÈSE co-dirigée par :

M. MULLER Gilles	Directeur de Recherche à Inria, Paris 6, France

RAPPORTEURS :

M. CONTASSOT-VIVIER Sylvain	Professeur, Université de Lorraine
M. RÉVEILLÈRE Laurent	Professeur, Université de Bordeaux

EXAMINATEUR:

M. BASTOUL Cédric	Professeur, Université de Strasbourg

To my Beloved Family and Friends...

" إِذَا مَا طَمَحْتُ إِلَى غَايَةٍ رَكِبْتُ ٱلْمُنَى وَ نَسِيتُ ٱلْحَذَرْ ... "

إِرَادَةُ ٱلْحَيَاةْ ـ أَبُو ٱلْقَاسِمْ ٱلشَّابِّي

"When i aspire to a goal, i ride ambition and i forget caution. . . "

Abu-AlQasim AlShabi, Will to Live

Contents

List of Figures

List of Tables

CHAPTER 0

Résumé en Français

0.1 Introduction

La mémoire partagée et la mémoire distribuée sont les deux abstractions de bas niveau prédominantes de la programmation parallèle. OpenMP et POSIX sont les interfaces de programmation les plus couramment utilisées pour la programmation en mémoire partagée. Cependant, la puissance de traitement des multiprocesseurs à mémoire partagée reste limitée par l'impossibilité de mettre à l'échelle la mémoire partagée à un grand nombre de processeurs. Pour de nombreux domaines scientifiques et d'ingénierie, où des simulations à grande échelle et intenses en calcul sont nécessaires, écrire un code parallèle pour les architectures à mémoire distribuée devient inévitable.

La programmation sur les architectures à mémoire distribuée a ses propres défis inhérents. Un défi majeur consiste à définir comment les processeurs peuvent se coordonner pour résoudre ensemble un problème commun. Une telle coordination implique la définition d'un schéma de partitionnement des données en plus de la synchronisation explicite et de la communication pour gérer le mouvement des données entre les différents processeurs. MPI est actuellement le paradigme dominant et le plus largement utilisé pour la programmation parallèle à mémoire distribuée. Aucun des paradigmes listés précédemment n'est suffisant pour résoudre les défis actuels des plateformes à grande échelle qui combinent les mémoire partagées et distribuées dans des architectures hiérarchiques.

ORWL [CG10b] propose un modèle d'accès uniforme pour les ressources (données, processeurs, mémoire, etc.) utilisables en mémoire partagée, en mémoire distribuée ou en contextes mixtes. Ce paradigme de synchronisation inter-tâches cible des algorithmes parallèles itératifs orientés ressources. Il favorise le contrôle algorithmique et la cohérence des données en introduisant un mécanisme de verrouillage basé sur des FIFOs qui gère les dépendances de données entre les tâches et les ressources. L'une des originalités de ORWL est l'annonce proactive des ressources qu'une tâche nécessite pour un calcul futur. Une implémentation de référence [CG10a, GVM14] a prouvé la validité et la performance du modèle ORWL.

De même que d'autres paradigmes de programmation parallèle à usage général pour les architectures à mémoire distribuée, le code ORWL peut être complexe et fastidieux

à écrire. Actuellement, le principal obstacle à l'utilisation d'ORWL réside dans sa phase d'initialisation exigeante où le programmeur doit spécifier le schéma d'accès entre les tâches et les ressources et les positions initiales des tâches dans les FIFOs. Ces derniers sont décisifs pour la vivacité de l'application, et doivent donc être attribués avec beaucoup de soin. Une fois que cette étape fastidieuse mais nécessaire est effectuée correctement, toutes les itérations suivantes sont garanties d'être équitables et sans interblocage. Afin de réduire ce fardeau et améliorer la productivité des programmeurs, nous visons à générer automatiquement le code ORWL, y compris l'attribution de positions initiales dans les FIFOs, sans sacrifier la performance. Une approche commune est d'abandonner la généralité et d'adapter une solution pour un domaine particulier, ici les stencils. Le modèle de calcul des stencils consiste en des balayages itératifs sur une grille de données, où chaque élément de grille est mis à jour en fonction de ses éléments voisins. Comme les noyaux des calculs stencils sont bien structurés, ils devraient considérablement bénéficier d'une modélisation utilisant ORWL. Bien que le pouvoir de modélisation de ORWL dépasse largement les stencils, nous nous concentrons dans ce travail sur ceux-ci.

Les calculs stencils constituent la partie dominante de nombreuses applications scientifiques et d'ingénierie. Par exemple, de nombreux phénomènes physiques tels que la dynamique moléculaire ou les ondes sismiques peuvent être décrits par des équations aux dérivées partielles elliptiques dépendantes du temps (EDP), et s'appuyer sur des méthodes de discrétisation numérique basées sur des stencils pour les résoudre. Les calculs stencils constituent la partie dominante de nombreuses applications scientifiques et d'ingénierie. D'autres calculs de stencils sont au coeur de nombreux algorithmes de traitement d'images et applications telles que l'imagerie CT et IRM [CZ08], ainsi que la détection d'objets et le tracking. Une parallélisation et une optimisation efficaces des noyaux de stencil peuvent grandement améliorer les performances globales de ces applications.

Les calculs de stencils sont souvent parallélisés par une décomposition de la grille de données en plusieurs *blocs*, de telle sorte que le calcul d'un bloc spécifique nécessite des données mises à jour des blocs voisins. Conceptuellement, ceci impose l'utilisation de régions dites régions d'ombre (*shadow regions*) qui entourent chaque bloc avec ses données de voisinage mises à jour. Celles-ci sont très difficiles à gérer car elles nécessitent souvent une analyse de cas complexe pour la gestion des communication ainsi que de complexes calculs d'index. Habituellement, le fardeau de la création et de la gestion des régions d'ombre repose sur le programmeur d'application.

De plus, le programmeur est responsable de la gestion des calculs et des communications qui s'alternent sur les ressources des blocs, d'où la nécessité du support d'un puissant mécanisme de synchronisation inter-tâches tel que ORWL, mais pas sans effort comme mentionné précédemment. Pour alléger ce fardeau, notre approche consiste à ajouter une couche, basée sur un langage dédié (DSL) implicitement parallèle [FRR+07] au-dessus de ORWL.

Les DSLs offrent d'une part des notations, des constructions et des abstractions spécifiques au domaine, ce qui améliore l'expressivité et ainsi, la productivité dans le domaine particulier pour lequel ils sont conçus. D'un autre côté, les frameworks spécifiques au

domaine sont efficaces grâce aux connaissances spécifiques au domaine qui y sont incorporées. Ils fournissent généralement des performances acceptables et peuvent parfois atteindre des niveaux de performance qui surpassent les implémentations codées à la main. Par conséquent, ils permettent d'équilibrer la programmabilité et la performance.

Avec ce travail, nous visons à combler le fossé entre la productivité et la haute performance en permettant aux experts des domaines scientifiques ou aux programmeurs moyens de développer des codes stencils ORWL qui répondent aux besoins spécifiques de leurs applications sans devenir experts en programmation parallèle en général, ni en ORWL en particulier. Nous proposons une solution basée sur un DSL pour rendre les implémentations de stencils dans le cadre ORWL plus simples et plus facile à utiliser.

0.2 ORWL

ORWL (Ordered Read-Write Locks) est un paradigme de synchronisation inter-tâches pour les algorithmes parallèles et distribués orientés ressources. Il fournit des méthodes de synchronisation basées sur le verrouillage qui permettent un contrôle implicite, équitable et sans interblocage des ressources protégées. Ici, une ressource peut être une abstraction des données, du matériel ou du software sur lesquels des tâches interagissent. ORWL fournit un moyen de contrôler de manière algorithmique l'ordre d'exécution des tâches en fonction de leurs dépendances de données en introduisant un mécanisme de verrouillage basé sur des FIFOs qui gère les dépendances de données entre les threads. Il favorise le contrôle algorithmique et la cohérence des données, tout en garantissant un degré élevé de parallélisme et d'asynchronicité. Le modèle ORWL est basé sur les propriétés suivantes:

1. Association d'un verrou à chaque ressource ORWL. Les verrous et les ressources sont étroitement liés.

2. Une file d'accès (FIFO) pour chaque verrou.

3. Une distinction entre verrouillage pour écriture exclusive et verrouillage pour accès inclusif en lecture.

4. Une distinction entre les verrous et les *handles* de verrous (LH) qui sont des interfaces agissant sur les verrous. L'acquisition est accordée pour les *handles* de verrous (LH) plutôt que les threads.

5. Une opération de verrouillage consiste en une séquence formée de trois étapes: *request* (insertion à la file), *acquire* (bloquant jusqu'à atteindre la première position dans la file d'attente), puis *release* (libérer la ressource).

Toutes ces propriétés combinées assurent la vitalité, l'équité et l'efficacité de l'application.

Annonce Proactive de l'Utilisation des Ressources

La logique derrière ORWL est qu'une tâche anticipe et annonce de manière proactive les ressources nécessaires pour les calculs futurs. Par l'intermédiaire d'un *handle* de verrou, la tâche demande une place dans la file d'attente FIFO de la ressource (voir Listing 1). Ce mécanisme est non bloquant, c'est-à-dire qu'après avoir effectué la demande, la tâche est libre de poursuivre son exécution. Ce n'est que lorsque l'accès à la ressource devient indispensable à la poursuite de la tâche que celle-ci tente de l'acquérir. La tâche est ensuite bloquée jusqu'à ce que la demande atteint la première position dans la file d'attente et que l'accès à la ressource associée soit autorisé. Une tâche n'est dite active et ne peut être exécutée que lorsqu'elle acquiert tous les verrous demandés. En d'autres termes, une tâche est retardée jusqu'à ce qu'elle atteigne le premier rang dans les files d'attente de toutes les ressources nécessaires. Ce n'est qu'alors qu'elle est autorisée à être exécutée et à effectuer toute action pour laquelle elle est conçue.

Listing 1 — Une simple section critique opérant sur une ressource via un `orwl_handle`.

```
/*                      announce the access                      */
orwl_write_request(&loc, &handle);
/*              some operation without the resource              */
...
/*               then, block until access granted                */
orwl_acquire(&handle);
/*          some critical operation with locked resource         */
...
/*                    then, free the resource                    */
orwl_release(&handle);
```

Overlay de Synchronisation

Pour un système de tâches interdépendantes donné, le modèle ORWL associe un *overlay* de synchronisation en tant qu'abstraction du modèle d'accès aux données. Son rôle principal est de spécifier l'ordre d'exécution exact des différentes tâches. Il permet également de détecter d'éventuels blocages. Un *overlay* de synchronisation sur un système de tâches interdépendantes est défini comme suit:

1. L'espace de données est partitionné de façon maximale en fragments primitifs appelés les *lieux* ORWL en fonction des différentes dépendances de données.

2. Un verrou ORWL est associé à chaque lieu ORWL.

3. Un handle de verrou est associé à chaque demande (*request*) de verrouillage.

4. Chaque lieu ORWL peut empiler un nombre illimité de demandes de verrouillage en lecture et en écriture.

5. Les demandes de verrouillage en attente d'un lieu dans une file d'attente sont numérotées de bas en haut. Cette numérotation correspond à la position de la demande de verrouillage dans la file d'attente des ressources.

6. Une demande ne peut être servie que si elle occupe le rang le plus bas dans la file d'attente de la ressource en question.

7. Une tâche est active et peut donc être exécutée lorsque toutes ses demandes de verrouillage sont acquises.

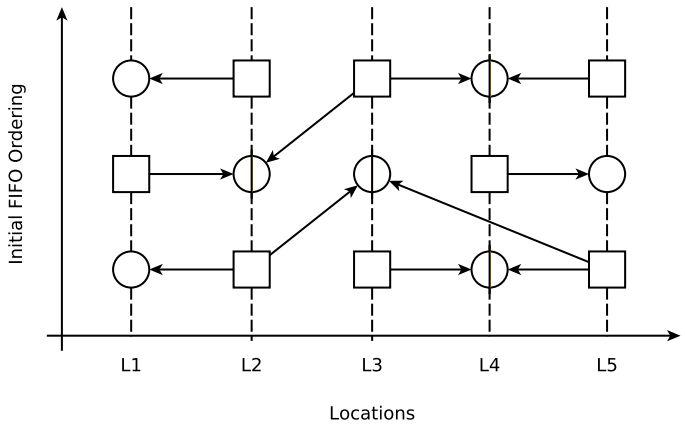


Figure 1 — Un overlay de synchronization

L'overlay de synchronisation est visualisé via un graphique de dépendance des données, comme illustré à la figure 1. Les lieux sont représentés sur l'axe des x. Les files d'attente sont présentées par des lignes pointillées verticales. Les demandes d'écriture exclusives (Xreqs) et les demandes de lecture inclusives (Ireqs) sont symbolisées par □ et ◯, respectivement. Les flèches connectent les Xreqs aux Ireqs, symbolisant ainsi le besoin d'une demande d'écriture donnée d'accéder en lecture à d'autres lieux requis avant son exécution.

0.3 Modèle Polyédrique

Le modèle polyédrique, également appelé modèle *polytope*, est une abstraction mathématique qui offre une représentation capturant l'exécution de programmes sous une forme mathématique adaptée à l'analyse et à la transformation à l'aide d'outils de l'algèbre et de la programmation linéaires. Il offre une plateforme puissante pour

l'optimisation des programmes à la compilation. L'utilisation la plus courante du modèle polyédrique est la transformation des boucles imbriquées. Le modèle polyédrique représente l'espace d'itération sous la forme d'un ensemble de points entiers à l'intérieur d'un polyèdre paramétrique sur lequel des opérations correspondant à des transformations de boucles sont effectuées. Les polytopes transformés sont ensuite traduits en nids de boucles équivalents, mais optimisés. Le modèle polyédrique est applicable à toute séquence de boucles imbriquées de manière arbitraire avec des bornes et des accès affines, également appelées nids de boucles affines. Ceux-ci constituent le noyau des calculs stencils.

L'utilisation du modèle polyédrique en tant que représentation de programmes permet l'utilisation d'une large gamme d'outils puissants et de libraries qui implémentent des opérations liées telles que Clan [Bas08], ISL [Ver10], Pluto [Bon09] et Cloog-ISL [Bas13] pour l'extraction polyédrique, test de dépendance, transformation automatique et génération de code, respectivement. Dans ce travail, nous utilisons Pluto, un optimiseur de boucles source à source basé sur le modèle polyédrique. Il est largement utilisé pour sa capacité à fournir simultanément un parallélisme à gros grain sans synchronisation et à améliorer la localité des données pour des boucles affines statiques imbriquées. Pluto est principalement connu pour ses capacités de parallélisation et de *tiling* efficace, mais il prend également en charge un large éventail de transformations de boucles, telles que la fission, fusion, skewing, etc. Pluto prend en entrée des nids de boucles affines séquentiels écrits en C et génère du code C OpenMP parallèle.

0.4 Calculs Stencils

Les noyaux de stencils apparaissent dans un large éventail d'applications scientifiques et techniques allant des solveurs numériques et d'équations différentielles partielles au calculs de physique [TU90, BRHS92, NKV94], ainsi qu'au traitement d'images [CHZ11, CZ08]. Malgré leur apparente simplicité et similitude, les calculs stencils sont en effet divers et peuvent être distingués par différents aspects et caractéristiques. Ceux-ci ont un impact direct sur leur parallélisation efficace.

Les calculs stencils constituent un pattern de calcul largement utilisé qui effectue des balayages sur une grille régulière multidimensionnelle, réalisant des calculs de plus proches voisins. La grille de calcul est un sous-ensemble contigu de $\mathbb{Z}^n$ dans un système de coordonnées cartésien. À chaque itération, chaque élément de la grille est mis à jour en fonction d'un sous-ensemble des éléments voisins des itérations actuelles ou précédentes. Dans ce travail, nous considérons des calculs stencils effectuant des mises à jour itératives point par point sur une grille à n dimensions, selon une fonction similaire au calcul suivant:

$$
\begin{aligned}
&R^{\varphi}[i_1][i_2]...[i_n] = \\
&\sum_{\nu=0}^{\mu}\left(\sum_{k} C_k[i_1][i_2]...[i_n] \times A^{\varphi-\nu}[i_1 \pm h^{\nu}_{k,1}][i_2 \pm h^{\nu}_{k,2}]...[i_n \pm h^{\nu}_{k,n}]\right. \\
&\left. + \sum_{m} \alpha_m \times A^{\varphi-\nu}[i_1 \pm \ell^{\nu}_{m,1}][i_2 \pm \ell^{\nu}_{m,2}]...[i_n \pm \ell^{\nu}_{m,n}]\right)
\end{aligned}
$$

Ici, φ est l'itération actuelle et μ le nombre d'itérations précédentes impliquées dans le calcul. L'élément central et ses éléments voisins des précédentes itérations $A^{\varphi-\nu}$, $\nu = 0,\ldots,\mu$, sont pondérés par des grilles de coefficients C_k et des constantes scalaires α_m.

Le calcul peut impliquer autant de grilles de coefficients C_k et de constantes scalaires α_m que nécessaire. La notation $\pm$ est un raccourci que nous utilisons ici pour économiser de l'espace. Le nouvel élément A^{φ} est déduit de la valeur intermédiaire R^{φ}. Nous appelons A la donnée principale (*main data*), c'est-à-dire la grille qui subit le calcul. Nous présupposons qu'il n'existe qu'une seule grille de données principale, mais qu'il peut y avoir plusieurs grilles de coefficients que nous appelons des données auxiliaires. Les données auxiliaires sont restreints à avoir la même topologie et la même taille que les données principales. Ils doivent également être accessibles au même endroit $[i_1][i_2]...[i_n]$ que l'élément central. Contrairement aux grilles de coefficients, il est possible d'accéder à la grille de données principale à n'importe quelle position. $h^{\nu}_{k,d}$ et $\ell^{\nu}_{m,d}$ présentent des offsets séparant les éléments voisins accédés de l'élément central. Nous appelons le maximum de ces offsets le *halo* du calcul:

$$halo = \max_{\nu,k,d,m} \{h^{\nu}_{k,d}, \ell^{\nu}_{m,d}\}.$$

0.5 Dido

Dido est un langage dédié implicitement parallèle pour les applications stencils multi-dimensionnelles générales qui utilise ORWL en tant que backend de communication et d'exécution. Dido offre une interface conviviale qui capture les spécifications de stencils et génère automatiquement du code haute performance parallèle pour les architectures à mémoire distribuée.

0.5.1 Architecture

Comme illustré dans la figure 2, Dido est en interne composé de quatre composants principaux: le lexer, l'analyseur syntaxique (*parser*), l'arbre de syntaxe abstraite (AST) et le générateur de code. L'utilisateur fournit des spécifications écrites dans le langage introduit. Celles-ci sont analysées afin d'extraire les caractéristiques du stencils qui sont transformées, dans un deuxième temps, en un arbre de syntaxe abstraite (AST). Etant donné la représentation interne du stencil sous la forme d'un AST, Dido génère du code

ORWL. Dido est implémenté en utilisant les générateurs de lexer et d'analyseur syntaxique OCaml (ocamllex, ocamlyacc).

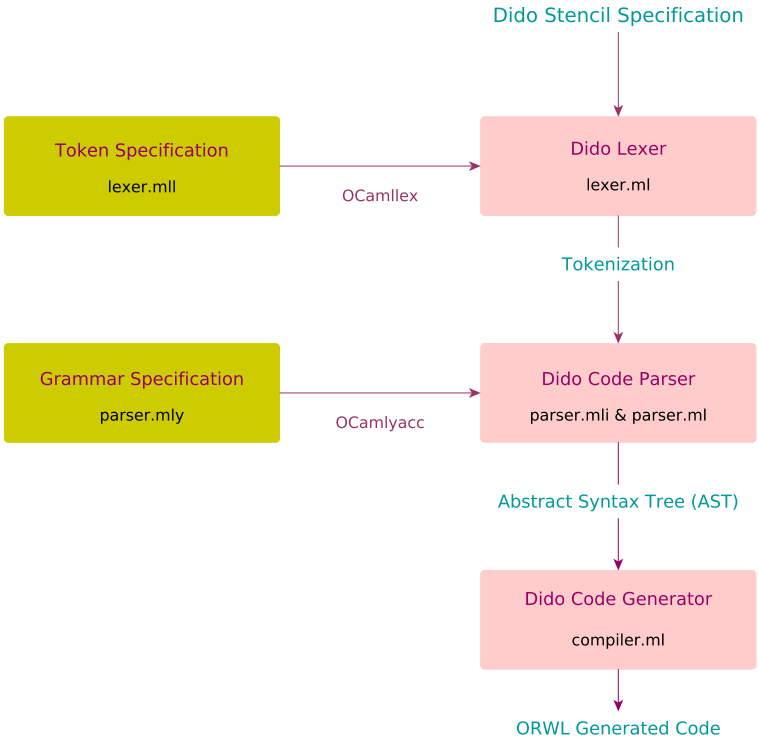


Figure 2 — Architecture de Dido.

0.5.2 Grammaire

Dido prend en entrée un fichier contenant un ensemble de paramètres spécifiant la topologie et la taille du calcul stencil dans une syntaxe concise et triviale. Le Listing 2 décrit la grammaire complète de la syntaxe de spécification de stencil au sein de Dido.

Afin d'économiser du temps et des efforts de compilation, le DSL est divisé en deux parties:

- La première partie englobe les paramètres structurels de l'application. Ces paramètres définissent la topologie du problème et permettent la génération des parties les plus complexes du code ORWL. Cela inclut les lieux de données, les opérations, les initialisations des *handles* et des positions initiales dans les FIFOs, ainsi que les sections critiques. Le temps de compilation du code ORWL, généré par cette partie du DSL, peut être relativement long.

- La deuxième partie comprend les paramètres d'exécution pour une instance particulière de problème. Ils peuvent être fournis dans le DSL, s'ils sont fixes, ou sous forme de paramètres runtime. Le code résultant de la partie structurelle du DSL peut être utilisé pour différentes instances de problèmes. L'utilisateur a la possibilité de générer tout le code à partir de zéro ou de ne générer que la fonction main en fonction des paramètres d'exécution spécifiés dans la partie instanciation.

0.5.3 Pattern de Génération de Code

Le but de ce travail est d'automatiser la génération de code stencil multidimensionnel avec ORWL comme interface de communication et d'exécution. Pour ce faire, nous devions définir un pattern d'utilisation généralisé qui réponde aux propriétés de vivacité des stencils ORWL. À cette fin, une connaissance approfondie de la sémantique du domaine était nécessaire. Nous devions d'abord analyser les implémentations ORWL des stencils 2D et 3D et extraire les contraintes qu'ils devaient respecter. Ensuite, le pattern extrait était généralisé pour des stencils multidimensionnels. Le pattern suggéré consiste de la combinaison d'un schéma non trivial de partitionnement des données, avec une forme itérative d'opérations que nous appelons *CompUp*.

CompUp Form

Pour améliorer l'expressivité de notre outil et faciliter la génération de code, nous avons convenu d'exprimer les programmes itératifs ORWL sous une forme que nous avons nommée *CompUp*. Comme le montre la Figure 3, nous convertissons un programme ORWL sous la forme de trois types d'opérations itératives: *Compute*, *Local Update* et *Global Update*.

1. L'opération *Compute* effectue le calcul comme spécifié par l'application. Elle lit les données accessibles localement qui sont importées par les opérations *Global Update* et sauvegardées dans des buffers locaux. Ensuite, elle exécute le noyau de calcul et écrit les résultats à la *main location*.

2. L'opération *Local Update* assure le transfert de données entre différentes ressources d'une même tâche. Elle lit les données mises à jour à partir de la *main location* et les stocke dans des buffers locaux pour les rendre disponibles pour les tâches voisines.

3. L'opération *Global Update* établit la communication entre la tâche principale et les tâches voisines. Elle lit les données des tâches voisines distantes et les écrit sur des buffers locaux, en les rendant disponibles pour la prochaine opération de calcul.

Afin de respecter les contraintes de forme canonique, des efforts ont été faits pour associer une seule opération à chaque lieu et, par conséquent, un accès en écriture exclusive à chaque lieu. Nous ajoutons des contraintes supplémentaires sur les positions

Listing 2 — Grammaire du Langage Dido.

```
program             = app-name, structural-params, execution-params;
app-name            = "ORWL_Application = ", name, ";";
name                = string;

(* Structural Parameters *)
structural-params = main-data, auxiliary-data, application;
main-data           = "Main_Data = {",
                      md-name, "(", md-dim, "D) in (", dimlist, ");",
                      "}";
md-name             = string;
md-dim              = natural number;
dimlist             = dim, {",",dim};
dim                 = string;

auxiliary-data      = "Auxiliary_Data = {", auxlist, "}";
auxlist             = [aux, {",",aux}, ";"];
aux                 = string;

application         = "Application = {", kernel, types, halo,
                      iter-halo, boundary-conditions,
                      neighbourhood, grid-traversal, data-element-type,
                      "}";
kernel              = "kernel = ", kernel-fun, "in", kernel-file-name;
kernel-fun          = string;
kernel-file-name    = string, ";";
types               = "types = ", type-file-name, ";";
type-file-name      = string;
halo                = "halo = ", halo-value, ";";
halo-value          = natural number;
iter-halo           = "iteration_halo = ", iter-halo-value, ";";
iter-halo-value     = natural number;
boundary-conditions = "boundary_conditions = ",
                      ("periodic" | "non-periodic"), ";";
neighbourhood       = "neighb = ",
                      ("Moore" | "von-Neumann" | shapelist), ";";
shapelist           = shape, {",",shape};
shape               = "(", coord, {",",coord}, ")";
coord               = natural number;
grid-traversal      = "grid_traversal = ", ("Jacobi" | "Seidel"), ";";
data-element-type = "data_element_type = ", string, ";";

(* Execution Parameters *)
execution-params  =  "Execution_Parameters = {",
                     ["iterations_number = ", iter-nmb, ";",
                     md-name, sizelist, "into", blocksizelist, ";",
                     "number_nodes = ", node-nmb, ";",
                     "number_tasks_per_node = ", tpn-nmb, ";"],
                     "init_file =", init-file, ";"
                     "}";
sizelist            = "[", size, "]", {"[", size, "]"};
blocksizelist       = "[", blocksize, "]", {"[", blocksize, "]"};
iter-nmb            = natural number;    init-file    = string;
node-nmb            = natural number;    tpn-nmb      = natural number;
size                = natural number;    blocksize    = natural number;
```

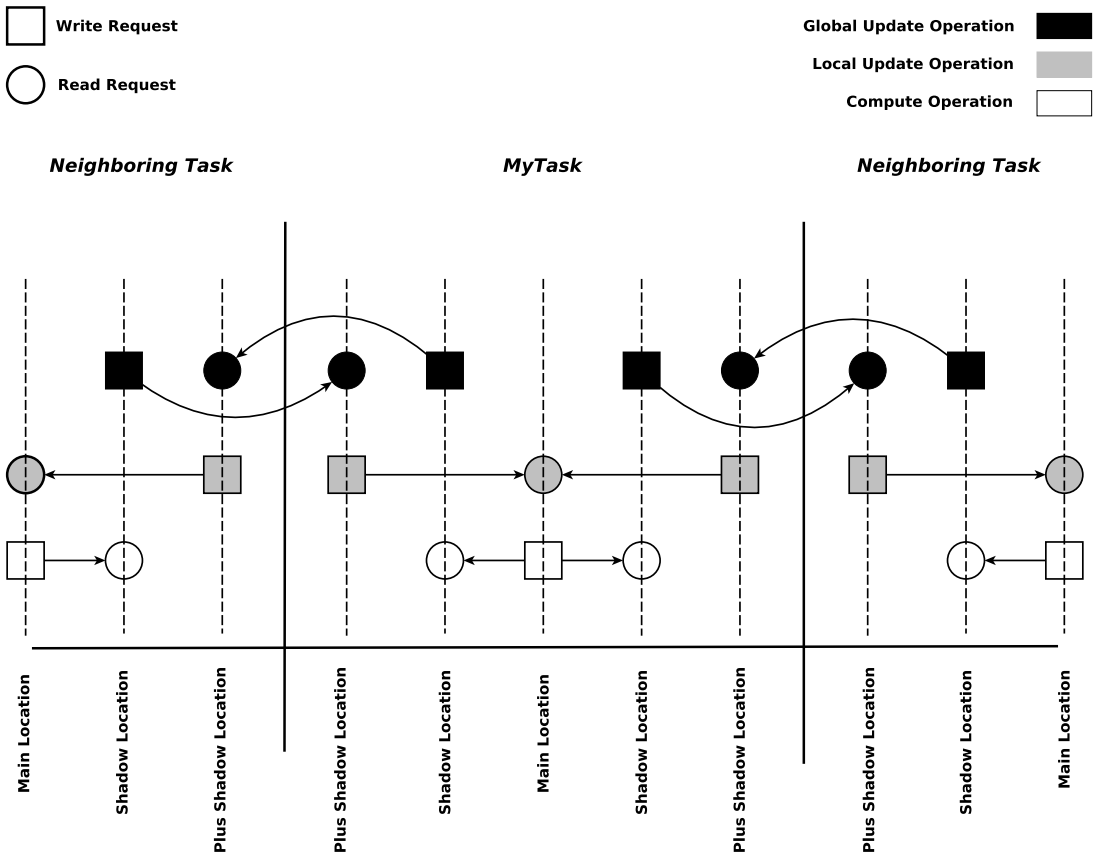
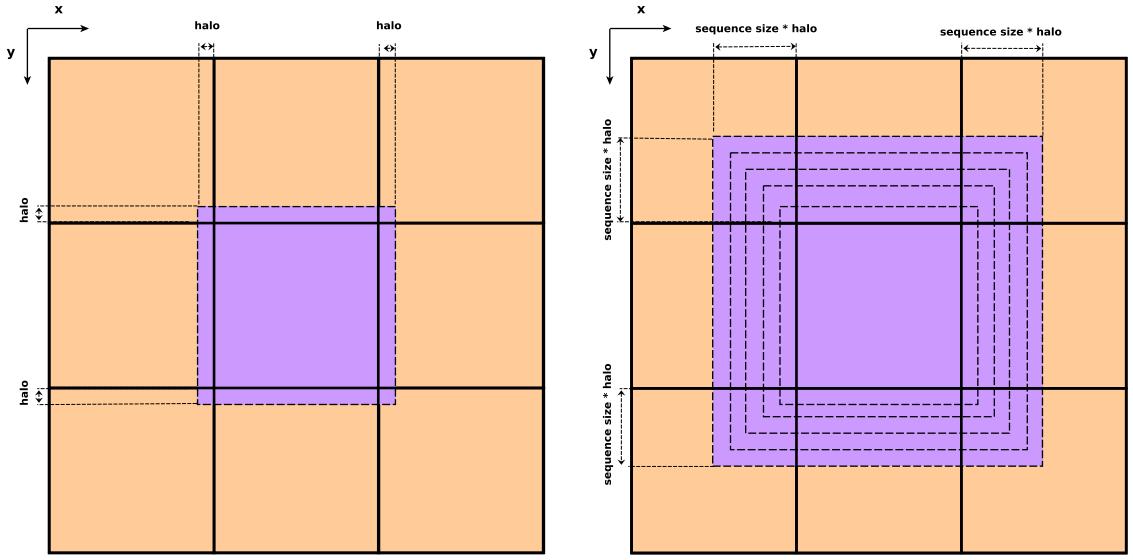


Figure 3 — Overlay de la Forme CompUp: Ordre d'insertion des requests des opérations *Compute*, *Local Update* et *Global Update* dans les FIFOs.

initiales prises par les opérations dans les FIFOs des lieux de données. D'abord, chaque opération doit avoir la même position initiale sur toutes les ressources dont elle a besoin. De plus, nous imposons que ces positions de priorités initiales suivent l'ordre ci-dessus. À savoir, l'opération *Compute* a la priorité sur les autres. Ensuite, l'opération *Local Update* arrive en deuxième position pour enregistrer les résultats calculés sur les buffers locaux. La dernière priorité est attribuée à l'opération de mise à jour globale. En résumé, les ordres de requête dans les FIFOs sont d'abord organisés avec les positions indiquées par l'énumération ci-dessus, puis régénérés cycliquement au fur et à mesure de l'avancement des itérations. Nous prouvons qu'en ajoutant ces contraintes a la forme *CompUp*, la suite du calcul est garantie sans interbloccage.

Partitionnement Chevauché des Données

Le partitionnement chevauché des données consiste à étendre la *main location* pour inclure les éléments de la région *halo*. Chaque bloc est ensuite agrandi par deux fois le *halo*



(a) Overlapped Data Partitioning (b) Temporal Blocking Data Partitioning

Figure 4 — Principaux schémas de partitionnement des données appliqués par Dido pour un stencil 2D.

sur chaque dimension, comme indiqué à la figure 4(a). Le partitionnement chevauché des données simplifie grandement le code et plus précisément le calcul des éléments sur les frontières de chaque bloc. Il constitue le schéma par défaut de partitionnement des données appliqué par le générateur de code de Dido.

Localité Temporelle des Données

Ayant une faible intensité arithmétique, réduire la quantité et la fréquence des échanges de données frontières entre différents nœuds de calcul peuvent avoir un impact notable dans la parallélisation des calculs stencils. Le blocage temporel est l'une des optimisations les plus largement utilisées pour réduire le temps de communication [DMV+08, CSN+10, MS11, WHZ+09]. Il consiste à partitionner l'espace d'itération en séquences de taille fixe. Une telle séquence est composée d'itérations locales qui sont calculées alors que les données sont conservées dans la mémoire locale. Les communications ne sont effectuées qu'à la fin de chaque séquence. À cette fin, dans chaque dimension, chaque bloc de données est agrandi de deux fois le halo multiplié par la taille de la séquence, comme illustré à la figure 4(b). Pour calculer la première itération locale d'une séquence, tous les éléments du bloc étendu sont requis. Les résultats sont stockés localement. Pour la prochaine itération, aucune autre communication n'est requise car tous les éléments sont déjà présents. Après chaque itération locale, le domaine de la boucle est réduit de deux fois le halo.

Dido applique l'optimisation du blocage temporel lors de la génération du code pour les stencils 2D et 3D uniquement si spécifié par l'utilisateur. Sinon, le schéma par défaut

de partitionnement de données est appliqué.

Code Généré

Après avoir exécuté le générateur de code Dido, Dido génère cinq fichiers différents, y compris le fichier *Makefile*. Les quatre autres fichiers contiennent le code ORWL applicatif suivant:

- `{ApplicationName}-def.h` et `{ApplicationName}-def.c` composent la déclaration et définition des tâches ORWL, des lieux et des unités d'exécution.
- Le fichier `{ApplicationName}-main.c` contient la fonction *main* du programme ORWL.
- Le fichier `{ApplicationName}-tasks.c` comprend le code de communication ainsi que le différentes fonctions et opérations *CompUp*.

Le code généré est constitué de code C avec les bibliothèques P99 et ORWL.

Pluto pour l'Optimisation du Code Généré

Les calculs de stencil peuvent être écrits sous forme de simples boucles imbriquées avec des bornes et des accès affines linéaires. Cependant, les implémentations intuitives de boucles souffrent souvent d'une faible localité de cache. Pour améliorer la réutilisation des données et la localité intra-nœuds, nous avons combiné notre technique de génération de code avec l'optimiseur de boucle polyédrique *Pluto*. Nous avons choisi *Pluto* car il s'agit d'un transformateur de code source à source et peut donc être appliqué directement. Cependant, nous devions décider de l'entrelacement du code généré par *Pluto* avec ORWL et avons proposé deux solutions différentes: La première consiste à utiliser *Pluto* pour optimiser la localité des données sur chaque cœur, c.-à-d. au niveau de la tâche. Ici, chaque nœud de calcul est associé à un processus ORWL composé de plusieurs tâches. Ces tâches fonctionnent sur différents blocs de données en parallèle. Nous utilisons *Pluto* pour optimiser le noyau servi par ces tâches. Donc, ici, *Pluto* n'est pas utilisé comme un paralléliseur, mais plutôt comme un optimiseur de localité de données: le parallélisme est uniquement basé sur ORWL et le code généré est exclusivement du code ORWL. En particulier, cette solution n'utilise pas OpenMP. La deuxième solution consiste à utiliser du code parallélisé par *Pluto* au niveau du nœud. Ici, nous plaçons un processus ORWL consistant en une tâche/bloc par nœud. Pour cette solution, nous nous appuyons sur *Pluto* pour la localité des données, mais également pour le parallélisme à gros grain. Ceci fournit une solution hybride où ORWL et OpenMP sont utilisés pour garantir le parallélisme à différents niveaux.

Propriétés de Dido

Dido est un langage dédié implicitement parallèle qui présente une grande variabilité et prend en charge une large gamme de calculs stencils multidimensionnels. Il répond aux

Table 1 — Nombre de lignes de code manuscrites vs. nombre de lignes de code générées

métrique	**Livermore**	**3D Wave**	**3D 27-pt Jacobi**
lignes manuscrites	68	63	77
lignes générées	711	988	2081

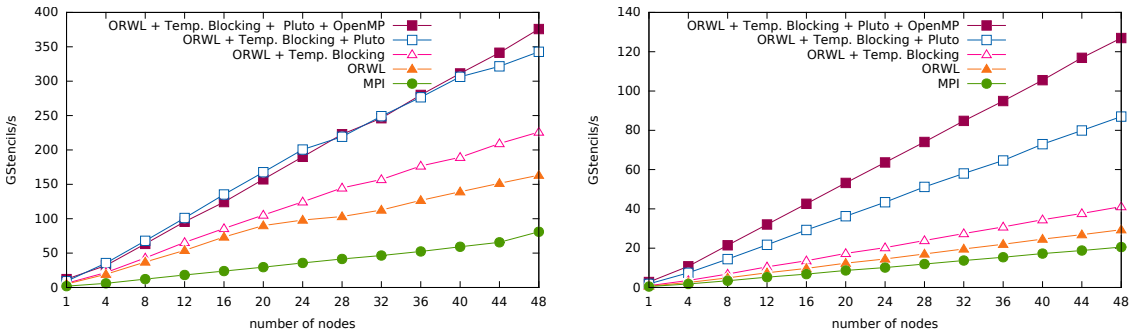
besoins spécifiques de l'application à un niveau d'abstraction élevé et offre à la fois des avantages en termes de productivité et de performances. Dido a été conçu pour améliorer la productivité des programmeurs en leur évitant d'écrire les parties complexes du code parallèle ORWL. Avec une spécification de stencils qui ne dépasse pas quelques lignes de code simple, des centaines de lignes de code sont générées. Le code généré est garanti d'être correct et sans erreur. Le tableau 1 présente le nombre de lignes manuscrites par rapport au nombre de lignes de code générées pour les benchmarks considérés. Le nombre de lignes à écrire par l'utilisateur, dans le cas d'un Jacobi 3D à 27 points par exemple, est réduit de 96%. Nous estimons qu'il s'agit d'une amélioration considérable en termes de productivité des programmeurs.

En outre, Dido offre des caractéristiques intéressantes pour les applications basées sur des stencils multi-dimensionnelles. D'abord, il permet de profiter de toutes les propriétés de ORWL. A cela s'ajoute de nouvelles propriétés telles que la garantie de l'exactitude, de la lisibilité et de la découvrabilité du code généré. La combinaison de ces propriétés fait de Dido un outil puissant de génération de code pour les calculs stencil en général, et en particulier pour les architectures à mémoire distribuée.

0.6 Évaluation des Performances

Nous avons effectué une campagne de tests pour prouver l'efficacité et la scalabilité du code généré par Dido sur des configurations composées de 768 coeurs physiques. Nous avons évalué les performances de différents benchmarks avec de différentes configurations:

1. Code ORWL généré par Dido avec uniquement les optimisations standards du compilateur.
2. Code ORWL généré par Dido avec blocage temporel.
3. Code ORWL généré par Dido avec blocage temporel et les optimisations de localité de données de Pluto. Cette solution n'utilise pas OpenMP.
4. Code ORWL généré par Dido avec blocage temporel ainsi que les optimisations de localité de données et la parallélisation de Pluto. ORWL et OpenMP sont ici tous deux utilisés pour assurer le parallélisme à de différents niveaux.
5. Code stencil MPI.



(a) Scalabilité de Heat 2D

(b) Scalabilité de Wave 3D

Figure 5 — Scalabilité du Code Généré par Dido pour les Benchmarks Heat 2D et Wave 3D

Nous avons aussi étudié la scalabilité du code généré par Dido pour une application réelle, *i.e.* la dynamique moléculaire.

Les expériences ont montré que le code généré par Dido est scalable et exploite les propriétés d'efficacité et de scalabilité du modèle ORWL (*cf.* figure 5). Ceci reste valable pour de différentes caractéristiques de stencils ainsi que pour des applications réelles (*cf.* figure 6). Le code généré par Dido atteint également des performances plus élevées que les implémentations manuscrites ORWL et MPI. Ceci est dû aux connaissances du domaine qui ont été incluses dans le pattern de génération de code ansi qu'à la qualité du code généré. La forme *CompUp*, en particulier, assure aux tâches une certaine indépendance par rapport à leurs voisins. En fait, dès que les régions d'ombre sont mis à jour dans la *main location*, les buffers correspondant aux lieux d'ombres sont libérés et prêts à recevoir les prochaines mises à jour des données des voisins sans interférer avec le calcul. Dès que l'opération *Compute* nécessite des valeurs mises à jour, celles-ci sont déjà disponibles et prêtes à être lues. En conséquence, les opérations *Compute* des tâches voisines peuvent être exécutées simultanément, ce qui améliore considérablement les temps d'exécution. Quant aux optimisations appliquées, le blocage temporel améliore dans la plupart des cas le temps de calcul. De plus, la combinaison du blocage temporel avec les transformations de boucles de Pluto fournissent des accélérations élevées. Les deux configurations, utilisant Pluto pour la localité des données, conduisent à des gains de performance substantiels. Cependant, la configuration hybride, utilisant OpenMP pour garantir le parallélisme intra-nœud, a donné de meilleurs résultats en termes de performances sur la plupart des tests. Cette solution hybride est donc un candidat clair pour un choix par défaut automatisé pour les utilisateurs de Dido qui ne souhaitent pas effectuer de calibrage à l'exécution.

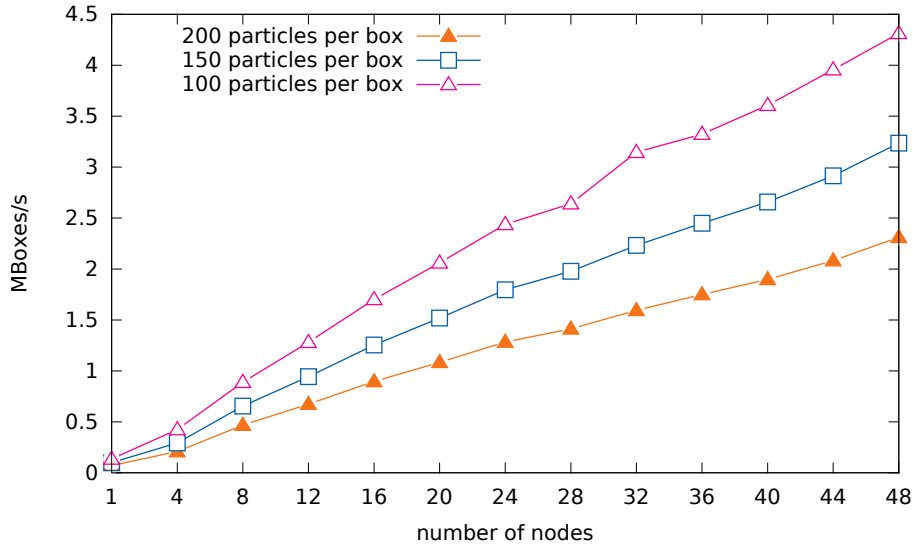


Figure 6 — Scalabilité du Code Généré par Dido pour l'Application de Dynamique Moléculaire

0.7 Conclusions

Nous avons présenté Dido, un langage dédié implicitement parallèle pour la génération de code ORWL de stencils. Dido réalise à la fois des gains en termes de productivité et de performances. Il permet à une grande communauté de programmeurs d'implémenter facilement et en toute sécurité des codes de stencils parallèles et d'exploiter, à moindre coût, les propriétés du modèle ORWL. Cela évite à l'utilisateur tous les détails de parallélisme nécessaires à l'écriture d'un code stencil haute performance. Il répond aux besoins des applications réelles en prenant en charge plusieurs types de données et de conditions de bords. En outre, des expériences ont montré que la productivité et les performances, souvent considérées comme antagonistes, peuvent être réconciliées lors de l'utilisation de Dido. Le code généré est scalable et offre des performances concurrentielles supérieures à celles du code écrit à la main.

De plus, nous montrons que le code ORWL généré est bien structuré et se prête à de différentes optimisations parmi lesquelles le blocage temporel que nous utilisons dans certains cas afin de réduire les coûts de communication et de minimiser les transferts de données. Nous combinons l'optimisation du blocage temporel avec les capacités de réutilisation des données intra-nœud de l'optimiseur de boucle polyédrique Pluto. Cela a considérablement amélioré les performances du code généré. Nous montrons également que Dido a le pouvoir d'expression pour modéliser des applications réelles basées sur des stencils.

La lisibilité du code généré par Dido permet à l'utilisateur de le prendre facilement en main. Il peut donc facilement le modifier. Dido peut ainsi devenir un outil intéractif aidant les utilisateurs à mettre en œuvre de grandes applications comprenant des calculs stencils et à générer du code stencil pour des architectures à mémoire distribuée.

CHAPTER 1

Introduction and Motivation

1.1 Context

Shared memory versus distributed memory are the two predominant low-level abstractions of parallel programming. At first glance, programming on shared-memory multiprocessors may seem easy as processors share a single view of the data. There is consequently no first-hand need for adding specific abstractions to the computational model for communication. Yet, this apparent facility is not met in practice. In general, three major problems persist:

- The existence of several writers or readers poses a threat to *data integrity*.
- Locking strategies, often used to grant access to critical sections, jeopardize the execution *liveness*.
- Shared data accesses from different processing cores can substantially subvert *parallel acceleration*.

OpenMP[1] and POSIX threads[2] are the most commonly used programming interfaces for shared-memory programming that deal with these difficulties. However, the processing power of shared-memory multiprocessors remains limited by the impossibility of scaling shared memory to a large number of processors. For numerous scientific and engineering fields where compute-intensive large-scale simulations are required [CB95, MW04, BBG+98, PPS+13] writing parallel code for distributed memory architectures becomes inevitable. Programming on distributed-memory architectures has its own inherent challenges. One major challenge consists in defining how the processors may coordinate in order to cooperatively solve a common computing problem. Such a coordination involves the definition of a data partitioning scheme in addition to explicit synchronization and communication to manage the data movement between the different processors. MPI[3] (Message Passing Interface) is currently the dominant and most

[1] http://www.openmp.org/
[2] http://pubs.opengroup.org/onlinepubs/9699919799/
[3] http://mpi-forum.org/

widely used paradigm for distributed-memory parallel programming. None of the previously listed computing paradigms by itself is adequate to solve today's challenges on large scale computing platforms which combine shared and distributed memory in hierarchical architectures.

ORWL, see [CG10b], proposes a uniform access model for resources (data, processors, memory, etc.) that can be used in shared memory, distributed-memory or mixed contexts. This inter-task synchronization paradigm targets iterative resource-oriented parallel algorithms. It favors algorithmic control and data consistency by introducing a FIFO-based lock mechanism that handles data-dependencies between tasks and data resources. One of the originalities of ORWL is the proactive announcement of the resources that a task requires for a future computation. A reference implementation [CG10a, GVM14] has proven the validity and performance of the ORWL model.

Similarly to other general purpose parallel programming paradigms for distributed-memory architectures, ORWL code can be complex and tedious to write. Currently, the main hurdle for using ORWL lies in its demanding initialization phase where the programmer has to specify the access scheme between tasks and resources and the initial positions of the tasks in the FIFOs. The latter are decisive for the liveness of the application, and thus should be attributed with a lot of care. Once this tedious but necessary step is done correctly, all subsequent iterations are guaranteed to be deadlock-free and fair. In order to alleviate this burden and enhance programmer productivity, we aim to automatically generate ORWL code, including the attribution of initial handle positions in the FIFOs, without sacrificing the performance. A common approach is to give up generality and tailor a solution for a particular domain, here stencils. The computational pattern of stencil computations consists of iterative sweeps over a data grid, where each grid element is updated as a function of its neighboring elements. As stencil kernels are well structured and computation-intensive, they should considerably benefit from a modeling using ORWL. Eventhough the modeling power of ORWL largely exceeds the stencils, in this work we concentrate on these.

Stencil computations constitute the dominant part of many scientific and engineering applications. For instance, plenty of physical phenomena such as molecular dynamics or seismic waves can be described by elliptic time-dependent partial differential equations (PDEs) and rely on stencil-based numerical discretization methods to be solved. Other stencil computations are at the core of many image processing algorithms and applications such as CT and MRI imaging [CZ08] and object detection [SGRP16] and tracking [GJM16a]. An efficient parallelization and optimization of stencil kernels can greatly enhance the overall performance of those applications. Therefore, a number of recent studies target optimizing stencil computations on both multicore CPUs and GPUs [CM09, PF10, Wol89, DMV+08, NSC+10].

Stencil computations are often parallelized through a decomposition of the data grid into several *blocks* or *tiles*, such that the computation of a specific block requires updated data from neighboring blocks. Conceptually, this enforces the use of so-called shadow regions that surround each block with its updated neighborhood information. These are very difficult to handle since they often need complex case analysis for communi-

cation statements and index calculations. Usually, the burden of the creation and management of the shadow regions lays on the application programmer. Apart from that, the programmer is responsible for handling overlapped computations and communications over block resources. Therefore, they need the support of a powerful inter-task synchronization mechanism such as ORWL, though not without effort as previously mentioned. To alleviate this burden, our approach is to add a layer, based on an implicitly parallel domain-specific language (DSL) [FRR+07] on top of ORWL.

DSLs offer on one hand appropriate domain-specific notations, constructs and abstractions, which improves the expressiveness and thus the productivity in the particular domain they are designed for, compared with general-purpose programming languages (GPL). On the other hand, domain specific frameworks achieve code efficiency thanks to the incorporated domain-specific knowledge. They usually provide acceptable performance and can sometimes reach the performance levels that are attained by hand-coded implementations. Hence, they allow to balance programmability and performance. Not surprisingly, in recent years, DSLs have been widely used in parallel programming to spare the user the details and the complexity related to parallel programming. In particular, numerous research efforts have adopted DSL solutions to optimize stencil computations. Some of them introduce auto-tuning frameworks for multicore architectures [DMV+08] and GPU accelerators [ZM12, HPS12] within DSLs. Others suggest DSL-based stencil compiler transformations to generate efficient code for GPUs and multicore processors[HHV+]. The Pochoir stencil compiler[TCLL11], for example, uses cache-oblivious parallelograms for parallel execution on shared-memory systems to produce high-performance code for stencils.

1.2 Contributions

With this work, we aim to bridge the gap between productivity and high performance by allowing scientific domain experts or average programmers to develop ORWL stencil codes that meet their specific application needs without becoming experts in parallel programming in general, nor in ORWL in particular. We propose a DSL-based solution to make stencil implementations within the ORWL framework simpler and more user-friendly.

Dido [SGM16] is an implicitly parallel domain-specific language for general multidimensional stencil computations that uses ORWL as a communication and runtime backend. It meets the specific needs of the application at a high level of abstraction and achieves both productivity and performance benefits. Our main contributions are:

- We present Dido, a user-friendly interface based on a Domain-Specific Language (DSL) that captures high level stencil abstractions and automatically generates ORWL parallel high-performance stencil code. The generated code can be deployed on shared-memory, distributed-memory or mixed contexts.
- We suggest a pattern for ORWL stencil programs that we call *CompUP*. This pattern

is relevant for different ORWL implementations not only for stencil computations. It ensures expressiveness, deadlock-freeness and better performance for ORWL programs.

- We show that Dido achieves a huge progress in terms of programmer productivity without sacrificing the performance. We expose the complexity of the generated code and the amount of complex details the DSL spares the user.
- We expose the large variability of Dido and its ability to support a wide range of stencil computations and real-world stencil-based applications.
- We show that the well-structured code generated by Dido lends itself to different possible optimizations and study the performance of two of them:
  - We integrate a temporal blocking optimization to the Dido code generator in order to reduce the communication overhead and minimize data transfers.
  - We combine Dido's code generation technique with the polyhedral loop optimizer Pluto to increase data locality and improve intra-node data reuse.
- We present experiments that prove the efficiency and scalability of the generated code that outperforms hand-crafted code.
- We present a benchmark and testing campaign that proves the efficiency and scalability of the Dido generated code on configurations composed up to 768 physical cores. We show that Dido stencil codes are widely scalable. They outperform MPI stencil implementations. On top of that, the two integrated optimizations combined show important performance gains.

1.3 Outline

The remainder of this thesis is organized as follows. In Chapter 2 and Chapter 3, we present the Ordered Read-Write Locks (ORWL) model and the polyhedral model, respectively. In Chapter 4, we describe the stencil computational pattern considered in this work as well as the motivations of our work. Chapter 5 describes the stencil benchmarks and real-world applications considered in this work. An overview of automatic code generation and optimization frameworks for stencil computations is presented in Chapter 6. In Chapter 7, we suggest a code generation pattern for ORWL multi-dimensional stencil computations. Chapter 8 provides a full specification of Dido grammar and architecture. In Chapter 9, we exhibit the structure of Dido generated code, as well as the different measures taken in order to make the complex generated code as clear, readable and efficient as possible. Chapter 10 highlights the different features that Dido guarantees for the generated code. Chapter 11 gives a performance evaluation of Dido generated code. We summarize our work in Chapter 12 and present future directions in Chapter 13.

CHAPTER 2

Ordered Read-Write Locks

Applications requiring parallel computation over a data space usually involve a set of interdependent tasks that present data dependencies over different shared resources, *i.e.*, the output of one task may be the input to one or more other tasks. The data dependencies can be read or write operations that are not necessarily atomic. As an illustration, take the example of block-partitioned iterative matrix computations where tasks are responsible for computing their own blocks, but still need to read data from neighboring blocks. These tasks would compete to acquire a write access to their own blocks and a read-access to neighboring blocks that are usually too big for an atomic wait-free operation on system level.

In [CG10b], PN. Clauss *et al.* came up with a new model that offers an efficient solution to similar problems. The building block of this model is called Ordered Read-Write Locks (ORWL), a special kind of read-write locks. ORWLs have similar semantics to common read-write locks *i.e.*, Concurrent Read and Exclusive Write, but apply in addition a FIFO ordering on the requesting tasks.

In this chapter, we provide a description of the Ordered Read-Write Locks (ORWL) model as presented in [CG10b]. Section 2.1 presents the ORWL model. Section 2.2 describes a formal representation of the data access pattern of ORWL tasks that enables the detection of deadlocks. In Section 2.3, we show that ORWL is particularly suitable for the cyclic data access pattern present in iterative computations. Section 2.4 describes the standalone reference implementation of ORWL. In Section 2.5, we highlight the fact that the ORWL model is a local-view programming paradigm. Section 2.6 outlines the main features that distinguish it from classic Read-Write Locks (RWL).

2.1 ORWL Model

ORWL (Ordered Read-Write Locks) is an inter-task synchronization paradigm for resource-oriented parallel and distributed algorithms. It provides lock-based synchronization methods that allow an implicit deadlock-free and equitable control of protected resources. Here, a resource can be an abstraction of data, hardware or software on which tasks interact. ORWL provides a way to control algorithmically the execution order of tasks based on their data dependencies by introducing a FIFO-based lock mechanism

that handles data-dependencies between threads. It favors algorithmic control and data consistency, while guaranteeing a high degree of parallelism and asynchronicity.

The ORWL model is based on the following features:

1. An association of a lock with each ORWL resource. Lock objects and resources are tightly coupled.
2. An access queue (FIFO) for each lock.
3. A distinction between locking for exclusive writing access and locking for inclusive reading access.
4. A distinction between locks and lock handles (LH) that are user interfaces acting on the locks. Acquisition is granted to lock handles rather than threads.
5. A three-step lock operation by a sequence of *request* (queue insertion), *acquire* (blocking until first in queue) and then *release*.

All of these features combined insure the liveness, equity and efficiency of the application.

Pro-active announcement

The logic behind ORWL is that a task anticipates and pro-actively announces the resources it is going to require for future computation. Through a lock handle, it *requests* a slot in the FIFO queue of the resource (*cf.* Listing 2.1). Such a mechanism is non-blocking, that is, after posting the request, the task is free to continue execution. Only when the access to the resource becomes indispensable for the continuation of the task, this latter attempts to *acquire* it. It is then blocked until the request moves first in the FIFO queue and the access to the associated resource is granted. A task is said to be active and can be executed only when it acquires all the locks it has requested. In other terms, a task is delayed until it detains the first rank in all the required resources waiting queues. Only then, it is allowed to be executed and can perform whatever action it is designed for.

Listing 2.1 — A simple critical section operating on one resource through an `orwl_handle`.

```
1  /*                      announce the access                          */
2  orwl_write_request(&loc, &handle);
3  /*             some operation without the resource                   */
4  ...
5  /*               then, block until access granted                   */
6  orwl_acquire(&handle);
7  /*         some critical operation with locked resource              */
8  ...
9  /*                   then, free the resource                        */
10 orwl_release(&handle);
```

Such a mechanism enables the runtime system to anticipate the access, *e.g.* by doing a data prefetching, at a reduced cost. In particular, it allows to hide access latency that could be caused by slow communication links.

2.2 Synchronization Overlay

For a given interdependent system of tasks, the ORWL model associates a *synchronization overlay* as an abstraction of the data access pattern. Its main role is to specify the exact execution order of the different tasks. It also enables the detection of deadlocks, if any, as explained in Subsection 2.2.3.

A synchronization overlay atop a system of interdependent tasks is defined as follows:

- The data space is maximally partitioned into primitive chunks called *ORWL locations* according to the different data dependencies.
- An *ORWL lock* is associated to each *ORWL location*.
- A *lock handle* (LH) is associated to each lock request.
- Each of the *ORWL locations* can stack any number of read and write lock requests.
- Lock requests in a location waiting queue are numbered bottom-up. This numbering corresponds to the position of the lock request in the resource waiting queue.
- A request can be served only when it holds the lowest rank in the resource waiting queue.
- A task is active and can thereby be executed when it has acquired all its lock requests.

The synchronization overlay is visualized through a data dependency graph as shown in Figure 2.1. Locations are represented on the x-axis. The waiting queues are presented by vertical dotted lines. Exclusive Write Requests (Xreqs) and Inclusive Read Requests (Ireqs) are symbolized by □ and ◯, respectively. Arrows connect Xreqs to Ireqs symbolizing the need for a given write request to hold read-locks on other required locations before getting executed.

2.2.1 Canonical Form

The *canonical form* is a configuration where each ORWL task has exactly one *ORWL location* to which it has requested an exclusive write access, and where each *ORWL location* is required for writing by only one task. In other words, in the *canonical form*, tasks and resources are in a 1-1 correspondence, where each task "owns" its unique resource for which it is responsible, and each resource has its unique task that will be able to modify it. Thereby, in the canonical form, each task posts exactly one *Xreq*.

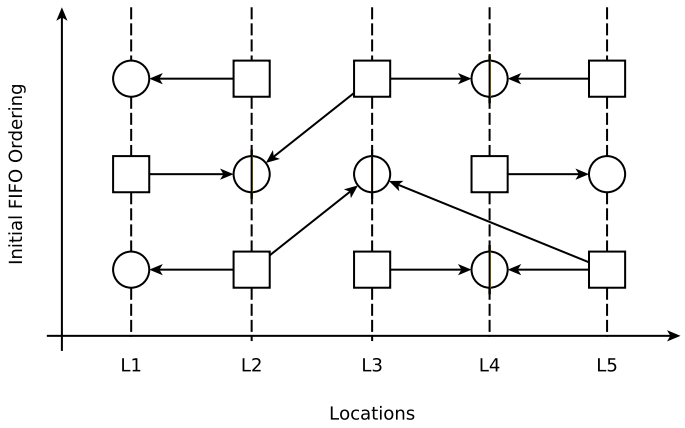
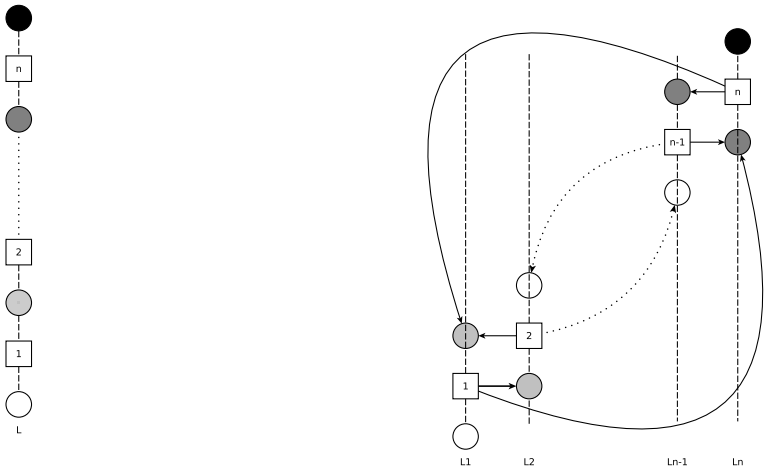


Figure 2.1 — An ORWL synchronization overlay

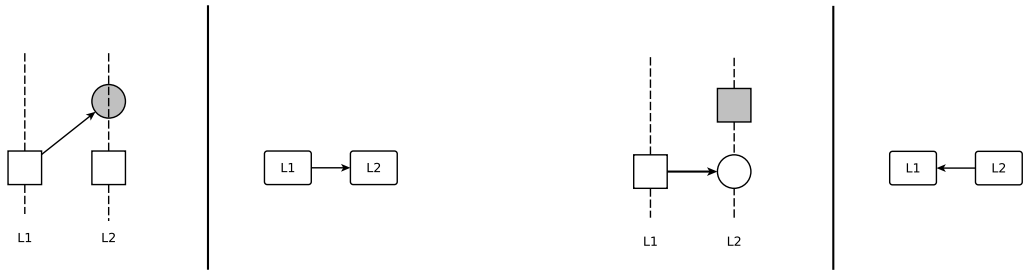


(a) Non-canonical (b) Canonical

Figure 2.2 — An overlay in a non-canonical form and its equivalent overlay in the canonical form after transformation [CG10b]

In its general form, the ORWL model allows multiple tasks to have write access to the same *ORWL location*. It has been proven in [CG10b] that such a general *overlay* system can be transformed into an *overlay* in the *canonical* form while keeping the same execution order of the tasks as shown in Figure 2.2. This *canonical form* can be achieved by splitting the original tasks into auxiliary sub-tasks that keep the same read-write semantics, see [CG10b] for the procedure and the proofs.

Ultimately, given an *overlay* in the *canonical from*, a task is executable if it acquires a



(a) Priority of Xreq over Ireq

(b) Priority of Ireq over Xreq

Figure 2.3 — Delays imposed by the priority order on the lock requests [CG10b]

write access to the resources it owns and a read access to the required resources owned by other tasks.

2.2.2 Delay Digraph

Given an *overlay* in the *canonical form*, the corresponding *delay digraph* is a directed graph that is constructed as follows:

1. Vertices correspond to *locations*.
2. Edges correspond to existing links in the *overlay* between a pair of *locations* L1 and L2.
3. Edge orientations are determined by the link in the *overlay* as follows:
   - L1 → L2 if the Xreq for L1 is connected to an Ireq which is above an Xreq for L2 (*cf.* Subfigure 2.3(a)).
   - L2 → L1 if the Xreq for L1 is connected to an Ireq which is below an Xreq for L2 (*cf.* Subfigure 2.3(b)).

2.2.3 Liveness Conditions

The liveness of an *overlay* depends only on the initial configuration, precisely the initial ordering of the lock-handles. It has been proven in [CG10b] (see Lemma cited below) that, for an *overlay* to be in deadlock, there must be a cycle among its dependencies that is detectable on its *delay digraph* (*cf.* Figure 2.4). If there is no such cycle, the *overlay* is guaranteed to be deadlock-free.

Lemma:

An overlay S in canonical form has a deadlock iff there is a cycle in its delay digraph.

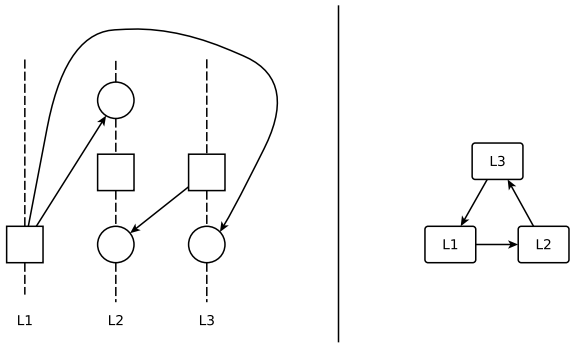


Figure 2.4 — An example of an overlay in the canonical form and its corresponding delay graph presenting a cycle and hence a deadlock

2.2.4 Overlay Initialization

Given a set of tasks $\mathcal{T}$, the conflict graph $C(\mathcal{T})$ is an auxiliary graph that models parallelism between tasks. An edge is drawn between $v, w \in \mathcal{T}$ if v reads the output of w or w reads the output of v. We say that a subset of tasks $\mathcal{T}' \subseteq \mathcal{T}$ is independent if there is no data dependency between any pair of tasks $v, w \in \mathcal{T}'$. Independent sets of tasks may be executed in parallel.

Algorithm 1 : Compute an initial request ordering [CG10b].

Input: A set of tasks $\mathcal{T}$, a set of lock locations $\mathcal{L}$ and for each task $T \in \mathcal{T}$ a list of Xreqs $(X_1, ..., X_w)$ and of Ireqs $(I_1, ..., I_r)$, where the X_i and I_j are locations in $\mathcal{L}$.

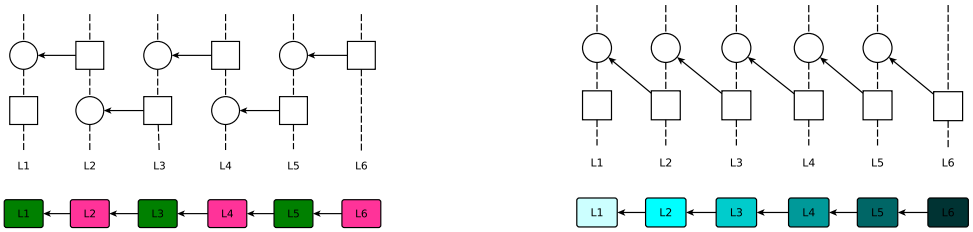
Output: For each $L \in \mathcal{L}$, a priority ordering of the requests for L such that the resulting overlay as a whole is deadlock-free.

Construct an implicit representation of the conflict graph $C(\mathcal{T})$;

Compute a coloring $\mathcal{T}_1$, ..., $\mathcal{T}_x$ of $C(\mathcal{T})$, by partitioning the tasks into independent task sets and attributing a different color to each partition;

Foreach *location* $L \in \mathcal{L}$ *of* T **do**
    initialize $p(L)$ to 0;

Foreach *color* $c = 1, ..., x$ **do**
    Foreach *task* $T \in \mathcal{T}_c$ **do**
        Foreach X_{req} X *of* T , $X = X_1, \ldots, X_w$ **do**
            increment $p(L)$, set the priority of X to the new value and increment $p(L)$ again;
        Foreach I_{req} I *of* T , $I = I_1, ..., I_r$ **do**
            set the priority of I to $p(L)$;



(a) The result of a 2-coloring

(b) One color class per location

Figure 2.5 — Two different initializations of the same set of requests [CG10b].

Algorithm 1 provides a strategy to construct a deadlock-free *overlay*. It consists of partitioning the tasks into independent task sets that do not have conflicts, and distinguishing them by attributing a different color to each partition. For a given task, lock initial requests are then placed according to the position of its color class. Figure 2.5 shows, *e.g.*, two possible initializations of the same set of requests following Algorithm 1. It has been proven in [CG10b] that Algorithm 1 generates an *overlay* that is deadlock-free.

2.3 Iterative Tasks

The pro-active locking in ORWL enables a thread or a process to define several handles on the same lock, and thereby to newly request a lock by means of one handle while still actively holding a lock via another handle. Thereby, iterative tasks may insert their request for the next iteration in the FIFO while still holding a lock for the current one. This is a big advantage for iterative computations that access data in a cyclic pattern, and the library provides specific tools that implement this pro-active iteration scheme.

The type `orwl_handle2` presents a pair of `orwl_handle` that are bound to the same resource and used in alternation. At the start of each iteration, one of the two handles is bound to the resource to grant the access for the current iteration, while the other is inactive. At the end of the iteration, before releasing the lock, a new request for the next iteration is automatically posted through the inactive handle. This guarantees the reservation of the resource for the next iteration at the same initial FIFO ordering, before releasing the lock to grant access to other tasks operating on the same resource. Listing 2.2 shows an example of a request-acquire-release sequence over a resource accessed iteratively.

2.4 ORWL Library and Runtime

The ORWL model comes with a standalone reference implementation, `liborwl` [GVM14], that uses standard languages and interfaces (C and POSIX). It can be used in shared, distributed or mixed contexts. It is solely based on sockets for inter-node communication

Listing 2.2 — A critical section operating on one resource, accessed iteratively, through an orwl_handle2.

```
/*          bind a pair of handles to the resource                */
orwl_write_request2 (&loc, &handle2);
/*            some operation without the resource                   */
...
/*              then, block until acces granted                     */
orwl_acquire2 (&handle2);
/*        some critical operation with locked resource              */
...
/*       free resource + new request for next iteration             */
orwl_release2 (&handle2);
```

and on threads for intra-node sharing of data. In fact, Taktuk[1] is responsible for deploying the remote execution on available nodes as well as transporting files and commands. It sets up an interconnection network by collecting node properties (IP address, port number) and sending them to all nodes. This allows a point-to-point communication between the different tasks.

The runtime is decentralized: after an initial starting phase, no global task scheduling or explicit synchronization is necessary. Data transfer is triggered by events, such as the termination of a specific task, and is performed as much as possible concurrently to computation.

2.5 Local-View Programming Paradigm

An ORWL application is often composed of a set of equivalent concurrent tasks. Each task has a set of owned resources on which it executes its specific program code. An ORWL program is then specified as a local description of one task that we call the *main task*. The *main task* can refer to its own resources (local) or those of other tasks (remote). The tasks interact through communication and synchronization operations via the locks they subsequently acquire for these resources. This includes all operations executed on the locally owned resources. The tasks are realized by threads (local execution) and processes (remote execution) and the operations on the resources are implemented in shared memory, when possible, or through network communications on the socket level.

2.6 ORWL Features

Implementations of classic read-write locks (RWL), do not guarantee the access order of tasks in case of multiple simultaneous access requests over the same resource. To avoid starvation, these implementations tend to prioritize write requests even if these have

[1] http://taktuk.gforge.inria.fr/

been preceded by pending read-requests. This has many disadvantages. It is first a serious issue for the application accuracy. Applications should therefore not rely on the access order. In addition, possible lock inversion may lead to deadlocks that are tedious to avoid. Besides, writing tasks could alternate over one lock leaving the other reading tasks endlessly waiting. It is thus difficult to guarantee an equitable progression for all tasks. This explains the limited use of RWL despite its availability in all modern OS.

Unlike classic read-write locks, ORWL allows to retrace the exact execution order of tasks on a given resource. This order is fully determined by the initial configuration, and namely by the initial priorities attributed to the different lock-handles. This guarantees both accuracy and liveness for the application. ORWL guarantees, in addition, an equitable progression of all tasks. Since the order specified in the initialization phase is strictly respected, readers and writers, that have reserved a slot in the resource waiting queues are guaranteed an access. This prevents tasks from endlessly seizing or waiting for a lock and therefore guarantees the equity among tasks.

Experiments have also shown that ORWL is a powerful synchronization tool that guarantees efficiency for a wide range of applications, and several series of benchmarks have demonstrated outstanding performance in different computational contexts [CG10a, GVM14, SGM16]. It also presents a valid choice for out-of-core computation [CG10a].

To sum up, ORWL has many distinguishing features such as guaranteeing efficiency, accuracy, liveness and equity among tasks, which makes of it an excellent choice for programming parallel and distributed applications.

2.7 Initialization Phase

In the iterative setting, ORWL can guarantee the crucial properties of liveness and equity for all tasks, although this comes not without effort. As evoked in Subsection 2.1, the initial FIFO positions of the locking requests are decisive. They must be attributed with a lot of care in order to avoid cyclic dependencies that may lead to a deadlock. Consequently, the initialization phase where the programmer specifies the initial access order of the tasks to the resources is tedious to implement and error prone. As an example, Listing 2.3 corresponds to the initialization phase of a triangle of tasks, *Task 0*, *Task 1* and *Task 2* where each task of the three communicates with the two others. Each task has thus a write access to two local locations and a read access to two remote locations. Despite the simplicity of the example, it is obvious that the initialization phase is not trivial and can easily result in errors.

On the other hand, once this key step is done correctly, all subsequent iterations are guaranteed to be deadlock-free and fair. One of the main motivations of this work is to alleviate this programming task and automate the initialization phase, including the attribution of initial FIFO positions.

Listing 2.3 — Example of the initialization phase: a triangle of tasks, *Task 0, Task 1* and *Task 2*, each communicating with the two others.

```
ORWL_LOCATIONS_PER_TASK( // ORWL Location Declarations
  orwl_location_0,
  orwl_location_1,
);

ORWL_LOCATIONS_PER_TASK_INSTANTIATION();

static void orwl_taskdep_init_0(orwl_handle2 hdl_in[static 2],
                                orwl_handle2 hdl_out[static 2]) {
  // the internal state used for p99_rand
  p99_seed * const seed = p99_seed_get();
  // an ORWL server thread responsible for receiving incoming
  // connections and dispatching the data to callback threads
  // that will handle the requests.
  orwl_server * const server = orwl_server_get();
  orwl_write_insert(&hdl_out[0], ORWL_LOCATION(0,0), 0, seed, server);
  orwl_write_insert(&hdl_out[1], ORWL_LOCATION(0,1), 0, seed, server);
  orwl_read_insert(&hdl_in[0], ORWL_LOCATION(1,1), 1, seed, server);
  orwl_read_insert(&hdl_in[1], ORWL_LOCATION(2,0), 1, seed, server);
  orwl_schedule(); // synchronization
};

static void orwl_taskdep_init_1(orwl_handle2 hdl_in[static 2],
                                orwl_handle2 hdl_out[static 2]) {
  p99_seed*const seed = p99_seed_get();
  orwl_server*const server = orwl_server_get();
  orwl_write_insert(&hdl_out[0], ORWL_LOCATION(1,0), 0, seed, server);
  orwl_write_insert(&hdl_out[1], ORWL_LOCATION(1,1), 0, seed, server);
  orwl_read_insert(&hdl_in[0], ORWL_LOCATION(2,1), 1, seed, server);
  orwl_read_insert(&hdl_in[1], ORWL_LOCATION(0,0), 1, seed, server);
  orwl_schedule(); // synchronization
};

static void orwl_taskdep_init_2(orwl_handle2 hdl_in[static 2],
                                orwl_handle2 hdl_out[static 2]) {
  p99_seed*const seed = p99_seed_get();
  orwl_server*const server = orwl_server_get();
  orwl_write_insert(&hdl_out[0], ORWL_LOCATION(2,0), 0, seed, server);
  orwl_write_insert(&hdl_out[1], ORWL_LOCATION(2,1), 0, seed, server);
  orwl_read_insert(&hdl_in[0], ORWL_LOCATION(0,1), 1, seed, server);
  orwl_read_insert(&hdl_in[1], ORWL_LOCATION(1,0), 1, seed, server);
  orwl_schedule(); // synchronization
};
```

CHAPTER 3

Polyhedral Model

In this work, we use PLuTo[BHRS08], an automatic source-to-source loop-optimizer based on the *polyhedral model*, to increase data locality and improve intra-node data reuse. The *polyhedral model*, also called *polytope model*, is a mathematical abstraction that offers a representation capturing the execution of programs in a mathematical form suitable for analysis and transformation using machinery from linear algebra and linear programming. It offers a powerful platform for compile-time program optimization. Unlike typical compilers where intermediate representations abstract the input code as *abstract syntax trees* (AST) or *control-flow graphs* (CFG), polyhedral compilers reason about individual statement iterations.

The polyhedral model represents the *iteration space* as a set of integer points inside a *parametric polyhedron* on which operations corresponding to loop transformations are performed. The transformed polytopes are then translated into equivalent, but optimized, loop nests.

This chapter provides an overview of the polyhedral model. In Section 3.1, we present the basic mathematical notions behind the polyhedral model. Section 3.2 shows how affine loop nests are represented in this model. In Section 3.3, we present an overview of the most commonly used polyhedral tools and libraries.

3.1 Mathematical Background

In the following, $\mathbb{K}$ denotes an Euclidean space.

Definition 1 *(Affine function). A function $f : \mathbb{K}^m \rightarrow \mathbb{K}^n$ is affine if there exists a matrix $A \in \mathbb{K}^{m \times n}$ and a vector $\overrightarrow{b} \in A^n$ such that:*

$$\forall \overrightarrow{x} \in \mathbb{K}^m, f(\overrightarrow{x}) = A\overrightarrow{x} + \overrightarrow{b}$$

Definition 2 *(Affine hyperplane). An affine hyperplane of an n-dimensional space $\mathbb{K}^n$ is an affine subspace of dimension $n-1$. For $\overrightarrow{a} \in \mathbb{K}^n$ with $\overrightarrow{a} \neq \overrightarrow{0}$ and a scalar $b \in \mathbb{K}$, an affine hyperplane is the set of all vectors $\overrightarrow{x} \in \mathbb{K}^n$ such that:*

$$\overrightarrow{a}.\overrightarrow{x} = b,$$

For instance, a point, a line and a plane are hyperplanes in 1-, 2- and 3-dimensional Euclidean spaces, respectively.

Definition 3 *(Affine half-space). An affine hyperplane of equation $\overrightarrow{a}.\overrightarrow{x} = b$ divides the space into two half-spaces, defined by the inequalities:*

$$\overrightarrow{a}.\overrightarrow{x} \leq b \text{ and } \overrightarrow{a}.\overrightarrow{x} \geq b$$

where $\overrightarrow{a} \in \mathbb{K}^n (\overrightarrow{a} \neq 0)$ and $(\overrightarrow{b} \in \mathbb{K})$.

Definition 4 *(Convex Polyhedron) The intersection of a finite number of affine halfspaces defines a convex polyhedron, each half-space providing a face of the polyhedron. A polyhedron $P \subset \mathbb{K}^n$ can be expressed as a set of m affine constraints in a constraint matrix $A \in \mathbb{K}^{m \times n}$ and a constraint vector $\overrightarrow{b} \in \mathbb{K}^m$ as:*

$$P = \{\overrightarrow{x} \in \mathbb{K}^n \mid A\overrightarrow{x} + \overrightarrow{b} \geq 0\}$$

Definition 5 *(Parametric Polyhedron) A parametric polyhedron denoted $P(\overrightarrow{p})$ is a polyhedron parametrized by a vector $\overrightarrow{p} \in \mathbb{K}^p$ such that:*

$$P(p) = \{\overrightarrow{x} \in \mathbb{K}^n \mid A\overrightarrow{x} + B\overrightarrow{p} + \overrightarrow{b} \geq 0\}.$$

where $A \in \mathbb{K}^{m \times n}$ is a constraint matrix, $B \in \mathbb{K}^{m \times p}$ is a coefficient matrix and $\overrightarrow{b} \in K^m$ a vector of contants.

3.2 Polyhedral Representation of Loop Nests

The most common use of the polyhedral model is for loop nest transformation in program optimization. The polyhedral model is applicable to any sequence of arbitrarily nested loops with affine bounds and accesses, also called *affine loop nests*. These form the compute-intensive core of linear algebra kernels and stencil computations. For example, the left loop nest in Listing 3.1 can be represented using the polyhedral model as all of the loop bounds, data access expressions and control statements are affine functions. On the contrary, the right loop cannot be represented because of the array indirection present in the second loop upper bound (`C[i]`) that cannot be analyzed statically using the polyhedral model.

The polyhedral model allows the representation of memory references of individual statement iterations without handling the semantics of the instructions inside each statement. It represents statements in loop nests as a combination of three essential constructs: the *iteration domain*, the *access function* and the *schedule function*. In the following Subsections, we detail these constructs.

Listing 3.1 — Affine and non-affine loop nest examples.

```
for (i = 0; i <= N; i++)                    for (i = 0; i <= N; i++ )
  for (j = i; j <= N; j++ )                    for (j = 0; j <= C[i]; j++ )
    if (i+j-P>=0)                               A[i][j] = B[j]; //S2
      A[i][j] = B[i+j-P]; // S1
```

3.2.1 Iteration Domain

Definition 6 *(Iteration Vector). Let S be a statement of a program. An iteration vector of a statement S is an n-dimensional vector containing a possible combination of loop iterator values surrounding the statement S, where n is the depth of the loop nest enclosing the statement.*

The iteration vector represents thus a dynamic instance of a statement. For example, vectors $(0, p)$, $(1, p-1)$, ..., $(N-1, N-1)$ are possible iterator vectors for $S1$ (see Listing 3.1 left).

Definition 7 *(Iteration Space or Domain). The set of all iteration vectors for a statement S is called the iteration domain. It is denoted by* $\mathscr{D}^S$.

The iteration domain presents a compact way to represent all the instances of a given statement. It can be expressed as a system of linear constraints. For example, the iteration domain of the one statement in Listing 3.1 left, *i.e.*, `S1= {A[i][j]=B[i+j-P];}`, can be expressed as:

$$D_{S_1}(\mathtt{N},\mathtt{P}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} 0 \le i \le N \\ i \le j \le N \\ P \le i + j \end{array} \right\}$$

The iteration domain may be represented graphically as in Figure 3.1. Each integer point in the graphical representation stands for one instance of the statement.

3.2.2 Access Function and Schedule Function

In the polyhedral model, memory references accessed by each instance, corresponding to are represented as accesses to multidimensional arrays through linear functions of the enclosing loop iterators. These are the key for performing precise dependency analysis.

Definition 8 *(Access Function). For a statement S at depth d accessing a m-dimensional array, its access function is defined as:* $f\{\vec{x}\} = M\vec{x} + \vec{m}$, *where* $M \in Z^{d \times m}$. *The access function maps each point of the iteration domain with an array access.*

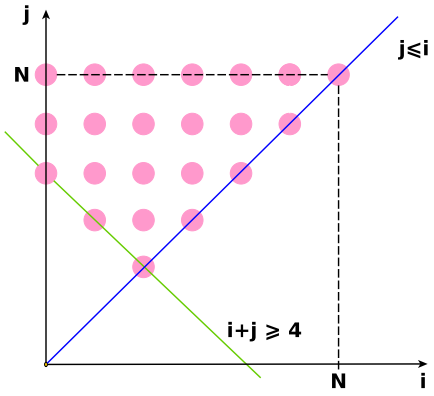


Figure 3.1 — Iteration Domain Graphical Representation.

Definition 9 *(Schedule Function). The scheduling function of a statement S, known also as the schedule of S, is a function that maps each dynamic instance of S to a logical date, expressing the execution order between statements:*

$$\forall x \in \mathscr{D}^S, \theta_S(\overrightarrow{x}) = T.\overrightarrow{x} + \overrightarrow{t}$$

Definition 10 *(SCoP). A maximal set of consecutive statements in a program with convex polyhedral iteration domains is called a static control part, or SCoP for short.*

More details about the polyhedral model and its uses in program representation are given in [Bas04].

3.3 Polyhedral Tools

The use of the polyhedral model as a program representation allows the use of a wide range of powerful tools and libraries that implement related operations such as Clan [Bas08], ISL [Ver10], Pluto [Bon09], and Cloog-ISL [Bas13] for polyhedral extraction, dependence testing, automatic transformation, and code generation, respectively. In this section, we enumerate the most commonly used polyhedral tools and libraries.

3.3.1 Polylib

Polylib (Polyhedral Library) [Loe99] is a polyhedral library that operates on both parameterized or non parameterized unions of polyhedra to compute intersections, differences, unions, convex hulls, simplifications, images and preimages, in addition to some input and output functions. It can also be used to compute Ehrhart polynomials in order to

count the vertices in a parameterized polyhedron. VisualPolylib is a visualization environment that enables the interactive use of Polylib.

3.3.2 OpenScop

OpenScop (SCoPLib) [Bas11] is a portable polyhedral specification format that is widely used to simplify the exchange between different polyhedral tools.

3.3.3 Clan

Clan (Chunky Loop ANalyzer) [Bas08] is a polyhedral parser for high level programs written in C, C++, C# or Java. It extracts the polyhedral loop nests and translates them to a polyhedral representation (OpenScop). The output can be then manipulated by other tools for dependence analysis, program transformation, parallelization, *etc.*

3.3.4 Candl

Candl (Chunky ANalyzer for Dependencies in Loops) [Bas12] is a dependency-analysis library. It extracts the dependency graph and the associated dependency polyhedra from a loop nest encoded in the OpenScop representation. It can also perform violation dependence analysis to check whether a given transformation respects the program dependencies. Thus, it is useful to build program transformations that respect the original program semantics.

3.3.5 ISL

ISL (Integer Set Library) [Ver10] is a thread-safe C library that provides operations to manipulate sets and relations of integer points bounded by linear constraints. It is mainly used in the polyhedral model for program analysis and transformation and for manipulating unions of polyhedra.

ISL supports a large range of operations to manipulate sets such as intersection, union, set difference, emptiness check, convex hull, affine hull, integer projection, computing the lexicographic minimum using parametric integer programming, coalescing and parametric vertex enumeration, *etc.*

3.3.6 CLooG

CLooG (Chunky Loop Generator) [Bas02] is a polyhedral library that has been originally written to generate the code resulting from optimizing compilers based on the polytope model. It can generate efficient code for a given polyhedron, *i.e.*, it finds a code (*e.g.*, C, FORTRAN...) that reaches each integral point of a union of parameterized polyhedra.

CLooG embeds many optimizations that aim at avoiding control overhead to produce highly efficient code. Additionally, it provides flags that enable the users to control the code generation mechanisms for efficiency or for readability. CLooG is used in major projects, *e.g.*, Pluto and Graphite, to generate the output code. It is also embedded in the GCC compiler for its advanced loop optimization features.

3.3.7 Pluto

PLuTo[BHRS08] is an automatic source-to-source loop-optimizer based on the polyhedral model. It is widely used for its ability to simultaneously provide synchronization-free coarse-grained parallelism and improve data locality for arbitrarily nested static affine loops. Pluto is mostly renowned for its parallelization and efficient tiling and diamond tiling capabilities, but it also supports a wide range of loop transformations such as fission, fusion, skewing, interchanging, reversal, unrolling, etc. Pluto takes as input sequential affine loop nests written in C and generates parallel OpenMP C code.

First, it uses Clan to parse and scan the input C code, extract the polyhedral loop nests and translate them to the OpenScop format. Second, Candl/ISL computes dependencies. Then, the Pluto transformation framework takes as input the polyhedral domains and dependence polyhedra and computes the polyhedral loop transformations. These are provided in an OpenScop representation to Cloog that generates finally the transformed code. Pluto automatically inserts OpenMP pragmas at the right place of the loop nest. The code generated by Pluto is OpenMP parallel code for shared-memory multicores. It is also optimized for locality and made amenable to auto-vectorization. Additional options are also provided to further tune tile sizes, unroll factors, outer loop fusion structures, etc.

In this work, we use the Pluto library for intra-node data locality optimization. Details are given in Chapter 9.

CHAPTER 4

Stencil Computations

Stencil kernels appear in a wide range of scientific and engineering applications ranging from numerical and PDE (Partial Differential Equations) solvers to computational physics [TU90, BRHS92, NKV94], as well as image processing [CHZ11, CZ08].

This Chapter presents an overview of stencil computations. In Section 4.1, we describe the computational pattern considered in this work. Section 4.2 highlights the diversity of stencil computations by exposing the different characteristics that distinguish them. Section 4.3 outlines the challenges in stencil programming on both shared and distributed memory architectures. In Section 4.4, we exhibit a motivating example to highlight the complexity of writing stencil code for distributed-memory architectures in general, and within ORWL in particular. We conclude in Section 4.5 by exposing the motivations of our work.

4.1 Computational Pattern

Stencil computations constitute a widely used computational pattern that performs global sweeps over a multi-dimensional regular grid, realizing nearest neighbor computations. The computation grid is a contiguous subset of $\mathbb{Z}^n$ in a Cartesian coordinate system. At each iteration, each grid element is updated following a function of a subset of its neighboring elements from current or previous iterations. In this work, we consider stencil computations performing iterative point-wise updates over an n-dimensional grid, according to a function similar to the following computation:

$$
\begin{aligned}
&R^{\varphi}[i_1][i_2]...[i_n] = \\
&\sum_{\nu=0}^{\mu}\left(\sum_{k} C_k[i_1][i_2]...[i_n] \times A^{\varphi-\nu}[i_1 \pm h^{\nu}_{k,1}][i_2 \pm h^{\nu}_{k,2}]...[i_n \pm h^{\nu}_{k,n}]\right. \\
&\left.+ \sum_{m} \alpha_m \times A^{\varphi-\nu}[i_1 \pm \ell^{\nu}_{m,1}][i_2 \pm \ell^{\nu}_{m,2}]...[i_n \pm \ell^{\nu}_{m,n}]\right)
\end{aligned}
$$

Here, φ is the current iteration and μ is the number of previous iterations involved in the computation. The center element and its neighboring elements from previous iterations $A^{\varphi-\nu}$, $\nu = 0,\ldots,\mu$, are weighted by coefficient grids C_k and scalar constants α_m.

The computation can involve as many coefficient grids C_k and scalar constants α_m as neeeded. The ± notation is a short-hand that we use here to save space ($A^{\varphi-\nu}[i\pm1][j\pm1]$ corresponds to $A^{\varphi-\nu}[i+1][j+1]+A^{\varphi-\nu}[i+1][j-1]+A^{\varphi-\nu}[i-1][j+1]+A^{\varphi-\nu}[i-1][j-1]$).

The new element A^{φ} is deduced from intermediate value R^{φ} by any set of statements. We refer to A as the *main data*, that is the grid that undergoes the computation. We presume that there is only one such *main data*, but that there can be multiple coefficient grids that we call *auxiliary data*. The *auxiliary data* are constrained to have the same topology and size as the *main data*. They must also be accessed at the same position $[i_1][i_2]...[i_n]$ as the center element. Unlike the coefficient grids, the *main data* grid can be accessed at any position. $h^{\nu}_{k,d}$ and $\ell^{\nu}_{m,d}$ present offsets separating the accessed neighboring elements from the center element. We call the maximum of these offsets the *halo* of the computation:

$$halo = \max_{\nu,k,d,m} \{h^{\nu}_{k,d}, \ell^{\nu}_{m,d}\}.$$

It has to be noted however, that the function above is valid only for inner elements and not for boundary elements. The most common boundary conditions for stencil computations are evoked in Subsection 4.2.4.

4.2 Stencil Characteristics

Despite their apparent simplicity and similarity, stencil computations are indeed diverse and can be distinguished by different aspects and characteristics. These have a direct impact on their efficient parallelization. For instance, some stencil computations are iterative, *e.g.*, *Laplacians*, and others are not, *e.g.*, image convolution and corner detection which is usually used in computer vision programs to extract certain features from an image. In this work, we specifically target iterative stencil computations as these can better benefit from a modeling using ORWL. In the following, we outline a number of important stencil characteristics that have a direct impact on their parallelization.

4.2.1 Operators

Stencils include a large panel of computations, involving the most commonly used differential operators, *i.e.*, *divergence*, *gradient* and *Laplacian*. Our suggested framework does not support stencils that have vector-valued input or output structure, *e.g.*, *divergence* or *gradient* operators. In this work, we focus on iterative stencil computations where the input and output grids have the same structure. *Laplacians*, *e.g.*, follow this computational pattern. They are commonly used to solve iteratively elliptic or time-dependent PDEs.

4.2.2 Structure

Stencil computations can be classified according to their structure or topology, including their dimension, *halo* and neighborhood as follows.

Stencil computations can be classified according to their structure or topology, including their dimension, halo and neighborhood as follows.

The dimension of the main data is one of the main stencil characteristics. The higher the dimension is, the more challenging the parallelization and optimization are. In this work, we support multi-dimensional stencil computations.

4.2.2.1 Dimension

The dimension of the *main data* is one of the main stencil characteristics. The higher the dimension is, the more challenging the parallelization and optimization are. In this work, we support multi-dimensional stencil computations.

4.2.2.2 Neighborhood

Stencil computations, also called *structured grid computations*, exhibit a static dependence pattern. Hence, the neighboring relationship remains constant during time. The most common neighborhood types for stencil computations are the *von Neumann* neighborhood and the *Moore* neighborhoods. As shown in Figure 4.1, while the *von Neumann* neighborhood is composed of only the elements that are orthogonally surrounding the center element, the *Moore* neighborhood includes in addition the elements on the corners.

More formally, a *von Neumann* neighborhood of range *halo* on an n-dimensional grid G is defined by:

$$\aleph^{(vN)}_{i_1,...,i_n} = \{j = (j_1, j_2, ..., j_n) \in G \quad | \quad |j_1 - i_1| + ... + |j_n - i_n| \leq halo\}$$

A *Moore* neighborhood of range *halo* on a n-dimensional grid G is defined by:

$$\aleph^{(M)}_{i_1,...,i_n} = \{j = (j_1, j_2, ..., j_n) \in G \quad | \quad |j_1 - i_1| \leq halo \quad and \quad ... \quad and \quad |j_n - i_n| \leq halo\}$$

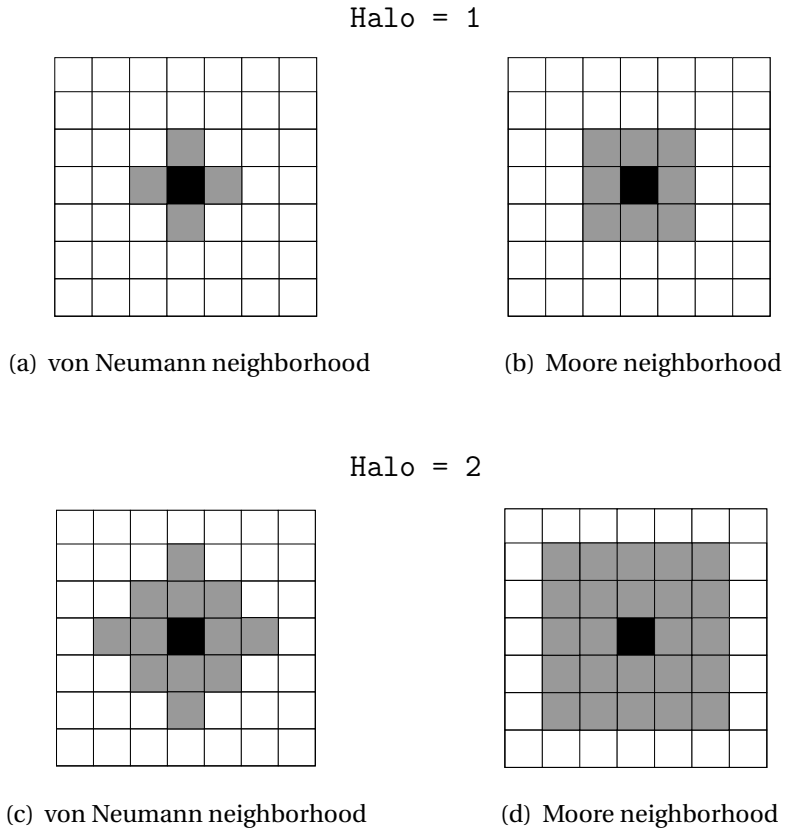
Other stencil neighborhoods can also exist. The dependence pattern can be symmetric or asymmetric.

4.2.3 Grid Traversal

The three most common stencil grid traversal methods are *Jacobi*, *Gauss-Seidel*, and *Gauss-Seidel Red-Black* (GSRB) iterations.

4.2.3.1 Jacobi

Jacobi-like iterations require at least two copies of the *main data* grid swapping their roles after each iteration. To update one grid element, required elements from input grids corresponding to previous iterations are read, and the result is written in the output grid as



(a) von Neumann neighborhood (b) Moore neighborhood

(c) von Neumann neighborhood (d) Moore neighborhood

Figure 4.1 — Moore and von Neumann neighborhoods for a 2D stencil.

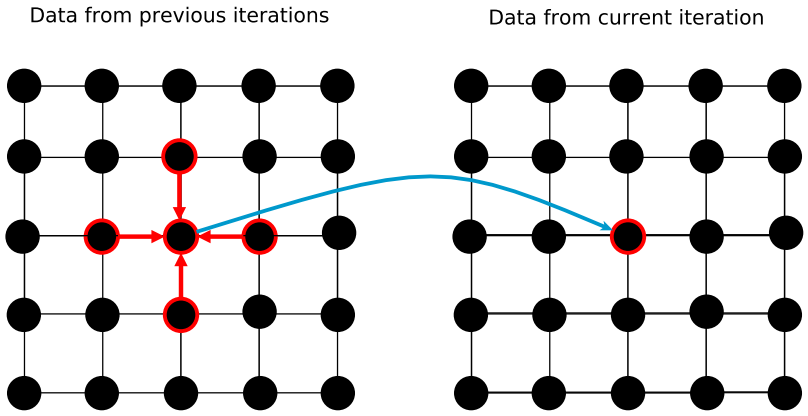
shown in Figure 4.2(a). Hence, each output grid element can be computed independently from other elements. Therefore, *Jacobi*-like iterations are easily parallelizable. However, requiring distinct input and output data grids increases memory storage and bandwidth.

4.2.3.2 Gauss-Seidel

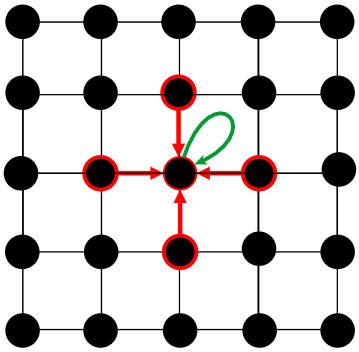
As shown in Figure 4.2(b), *Gauss-Seidel*-like iterations require only one single copy of the *main data* grid that undergoes in-place sweeps. One element update requires both already updated elements and other elements that have not yet been updated in the current iteration. The inherent dependency imposes an order to respect when computing elements, which significantly lowers the amount of available parallelism during one iteration. On the other hand, *Gauss-Seidel* iterations consume less memory than *Jacobi* iterations as they do not require several copies of the *main data* grid as for *Jacobis*.

4.2.3.3 Gauss-Seidel Red-Black

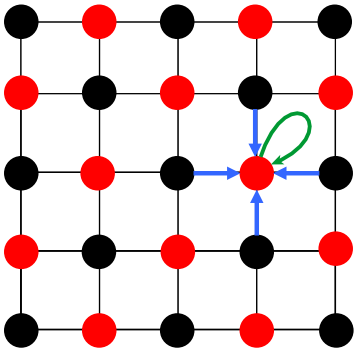
Similarly to *Gauss-Seidel* iterations, *Gauss-Seidel Red-Black* (GSRB) require only one single copy of the *main data* grid that undergoes in-place sweeps. However, GSRB defines two dependent iteration domains. In fact, points are divided into two categories, *i.e.*, so



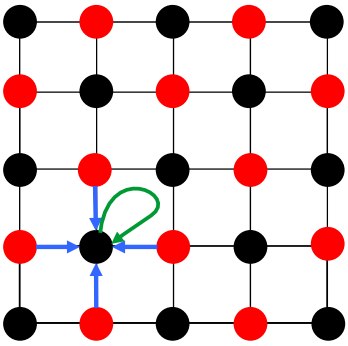
(a) Jacobi iterations



(b) Seidel iterations



(c) GSRB iterations: Red phase



(d) GSRB iterations: Black phase

Figure 4.2 — Grid traversal methods of stencil computations: *Jacobi*, *Gauss-Seidel*, and *Gauss-Seidel Red-Black* (GSRB) iterations.

called Red and Black, and the computation consists of two alternating phases. As shown in Figure 4.2(c), the red phase computes the red elements of the grid, which are every second point in the axis directions. The black phase computes the black elements from the updated red ones using the same stencil as shown in Figure 4.2(d). A black grid point can be updated only when all required red points have been updated. Once the required black points have been updated, a red point computation can restart. Since the same color points are independent from each other, their updates can be performed in parallel. A straightforward GSRB implementation would sweep twice over the grid, which is costly in terms of memory bandwidth. A classic solution to handle this problem is to have two wavefronts, *i.e.*, a leading one for red points and a trailing one for black points. Besides, GSRB iterations present non-linear conditionals as depicted in Listing 4.1. As a result, they are not handled by the automatic loop optimizer Pluto [BHRS08].

Listing 4.1 — Gauss-Seidel Red-Black (GSRB) iteration code.

```
1  // Red phase
2  for (i=1; i<n-1; i++)
3    for (j=1; j< n-1; j++)
4      if ((i + j) % 2 == 1)
5        u[i][j] = f (u[i+1][j], u[i-1][j], u [i][j+1], u[i][j-1]);
6  // Black phase
7  for (i=1; i<n-1; i ++)
8    for (j=1; j<n-1; j ++)
9      if ((i+j) % 2 == 0)
10       u[i][j] = f (u[i+1][j], u[i-1][j], u [i][j+1], u[i][j-1]);
```

4.2.4 Boundary Conditions

Separate boundary treatment, often imposed by PDE boundary conditions, is commonly required. The boundary conditions can be periodic or non-periodic. For periodic boundary conditions, the data grid is rolled up as an n-dimensional torus. Thus, the index calculations in each dimension are computed modulo the data grid size. As for non-periodic boundary conditions, the most commonly used are *Dirichlet* and *Neumann* [MF46]. While *Dirichlet* boundary conditions apply time-dependent functions, *Neumann* boundary conditions specify a time-constant value that often corresponds to the value the derivative should take on the boundary. Applications may however require further customization of boundary conditions in order to improve the problem simulation accuracy.

4.3 Stencil Programming Challenges

At first glance, the efficient parallelization of stencil computations appears to be straightforward and easy to achieve because of the high regularity of the computational pattern. Namely, stencils operate on regular Cartesian grids and are often uniformly defined as nested loops with affine bounds and accesses over the entire grid exhibiting a static and locally contained dependence pattern. This suggests massive parallelism, good potential

for locality optimization and easily achievable code efficiency. However, efficient parallelization of stencil computations remains challenging on both shared and distributed memory architectures.

4.3.1 Low Arithmetic Intensity

Stencil computations are memory-bandwidth bound with low constant arithmetic intensity. The latter is a kernel-specific metric that is used to predict and analyze the performance of algorithms and compute kernels. It is defined as the total number of floating-point operations per grid point divided by the number of data elements that need to be transferred from memory to the compute unit (CPU or GPU) in order to perform one grid point update [DJ05].

For a given stencil kernel, both the number of floating-point operations per grid point and the number of memory references are constant. Hence, unlike, *e.g.*, matrix multiplication , the arithmetic intensity is constant and independent from the problem size. Besides, the number of floating-point operations is often low compared to the number of transferred data elements. In fact, following the hardware data loading pattern, when reading one element from memory, the entire data cache line is loaded. This assumption is valid for both cache-based CPU architectures and shared memory CUDA-programmed GPUs. Hence, when loading the center point of a stencil, the halo layer including the neighboring points is automatically made available. Roughly speaking, for one grid element computation, the entire n-dimensional domain surrounding the element is brought to memory and the result is written back.

Stencil code efficiency is often impaired by the high memory I/O costs and low computational intensity. In fact, the performance is often limited by the available memory bandwidth, leading to idle processing cores. Hence, optimizing data movements in stencil codes is crucial to achieve high performance. ORWL proactive resource announcement could be a promising solution to minimize the cost of data transfers.

4.3.2 Boundary Irregularities

Stencil computations are pleasantly regular and their parallelization may therefore seem to be intuitive. However, stencil-based real-world applications are more challenging as they may, *e.g.*, present irregularities in terms of boundary conditions. If not well handled, boundary conditions can easily become a bottleneck in the performance of the overall stencil computation. The impact of, *e.g.*, making a test at every point to determine if it falls on the boundary can tremendously degrade the stencil code performance.

4.3.3 Distributed Memory Programming

Stencil computations are at the core of multiple scientific and engineering fields where compute-intensive large-scale simulations are required [WYTX13, Rap04]. Therefore,

writing parallel code for distributed memory architectures becomes sometimes inevitable. However, programming on distributed-memory architectures is particularly challenging, tedious and error-prone. In addition to the above mentioned challenges, stencil programming on distributed-memory architectures implies the definition of a data partitioning scheme, explicit synchronization and communication to manage the data movement between the different processors. In fact, an intuitive method to parallelize stencil computations is to decompose the *main data* grid into a set of *blocks* or *tiles*. While the computation in each block that is sufficiently far from the edges is independent from other blocks, the computation near the edges and corners depends on a subset of neighboring blocks, that are computed by different tasks. Thus, communication between tasks is needed in order to exchange updated values between blocks after each iteration. A specific mechanism must ensure the synchronization of overlapping computations and communications that access a protected resource. Here, the inter-task synchronization model ORWL, presented in Chapter 2 could be very helpful. In Section 4.4, we present a modeling of stencil computations using ORWL.

4.4 Stencils within ORWL

By the nature of their regular structure, stencil computations should considerably benefit from a modeling with ORWL: a block decomposition can be used to distribute the work to nodes and processors, and the limited and controlled overlap of access between neighboring blocks can be easily modeled by an access to shared resources.

Stencil modeling, within ORWL, implies the definition of a number of data locations and the attribution of an ORWL Lock to each of the defined data locations. A classic simplistic solution would be to define a number of *shadow locations* in addition to the *main location* that is composed of the block itself. These *shadow locations* consist in buffers where, at the end of each iteration, the newly computed data of block edges are saved. They are then available to neighboring tasks for reading. The number of ORWL data locations depends on the neighborhood type and shape of the stencil computation.

Figure 4.3 shows an example of a simple decomposition of a 2D matrix into four neighboring blocks (MT0, MT1, MT2 and MT3): Each task consists of one compute and 8 update operations acting on 9 locations that are the *main location* and 8 *shadow locations*: S, N, E, W, SE, NE, SW, NW. The compute operation reads (R) the data it requires through handles from the *shadow locations* of the remote neighboring blocks. These are first copied into the corresponding edges of the *main location* block. Then, the compute operation executes the computation kernel and writes (W) the results to the *main location*. The update operations, one for each *shadow location*, are then used to read (R) the newly computed values from the *main location* and write (W) them in the *shadow locations* to make them available for neighboring tasks. An excerpt from the corresponding ORWL code and initialization phase in particular is provided in Listing 4.2.

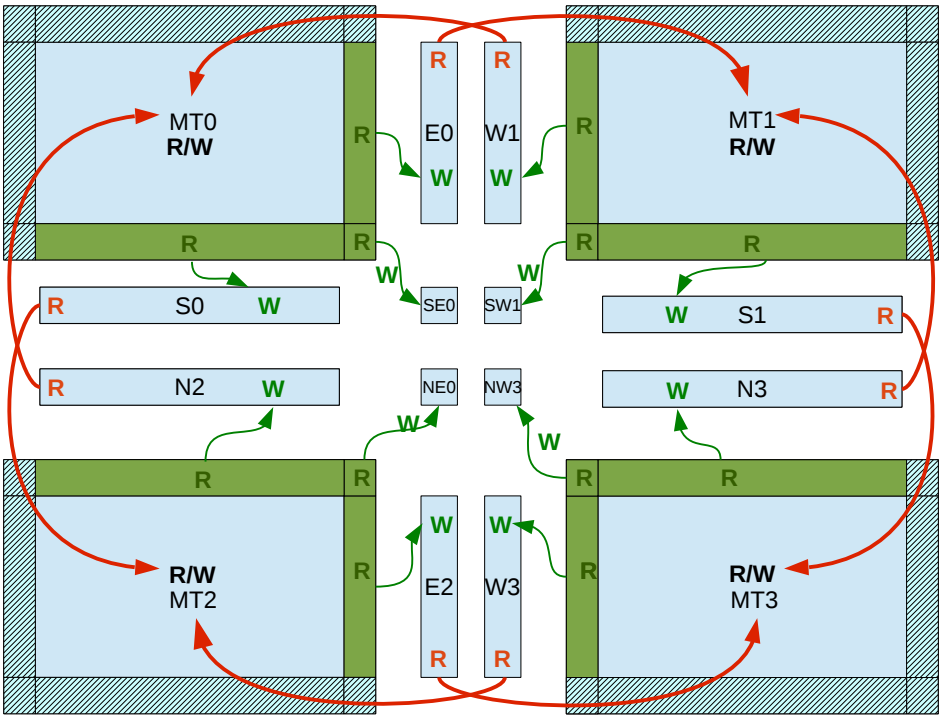


Figure 4.3 — An example of a 2D stencil ORWL modeling with a decomposition into four blocks

4.5 Motivations

As mentioned in Chapter 2, to take full advantage of ORWL properties, the programmer has to go through a tedious but necessary initialization phase. They have to carefully specify the access scheme between tasks and resources and to assign the initial positions of lock handles in the resource FIFOs as in Listing 4.2. The code corresponding to the initialization phase is long, complex and error-prone. Mistakes at this level may lead to deadlocks. Our principal motivation for developing our DSL, called *Dido*, is to alleviate this burden by automating the generation of this initialization phase in a fail-safe manner.

Furthermore, without the aid of the DSL, the programmer is responsible for handling the updates of the halo regions. They have to define the ORWL data locations explicitly, allocate their associated buffers and write the update operations. These programming tasks are by far longer and more involved than the computational kernel they serve. Additionally, it is very likely that the programmer makes mistakes when writing the indices in the complex update operations that have to be specified by using the halo margin offsets in each direction of the problem grid. The more dimensions the *main data* has, the more complex the indices become and the higher is the risk of making mistakes. With our DSL, we spare the programmer from writing complex indices and error-prone parts, by fully generating them.

Listing 4.2 — Initialization phase code of a 2D stencil ORWL modeling.

```
orwl_handle2 here = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 handle[task_neighb_amount] = {P99_DUPL(TASK_NEIGHB_AMOUNT,
                                                    ORWL_HANDLE2_INITIALIZER)};
/* Take the local lock in write mode */
orwl_write_insert(&here, myloc, 0, seed);
/* Take the distant locks in read mode*/
if (is_border(borders, borderP0)) {
   orwl_read_insert(&handle[NN],
         relative_task(row-1,  col, S, global_cols), 0, seed);
   if (is_border(borders, borderPP)) {
      orwl_read_insert(&handle[NEN],
            relative_task(row-1,  col, SE, global_cols), 0, seed);
   }
   if (is_border(borders, borderPM)) {
      orwl_read_insert(&handle[NWN],
            relative_task(row-1,  col, SW, global_cols), 0, seed);
   }
}
if (is_border(borders, borderM0)) {
   orwl_read_insert(&handle[SS],
         relative_task(row+1,  col, N, global_cols), 0, seed);
   if (is_border(borders, borderMP)) {
      orwl_read_insert(&handle[SES],
            relative_task(row+1,  col, NE, global_cols), 0, seed);
   }
   if (is_border(borders, borderMM)) {
      orwl_read_insert(&handle[SWS],
            relative_task(row+1,  col, NW, global_cols), 0, seed);
   }
}
if (is_border(borders, border0M)) {
   orwl_read_insert(&handle[WW],
         relative_task(row,  col-1, E, global_cols), 0, seed);
   if (is_border(borders, borderMM)) {
      orwl_read_insert(&handle[SWW],
            relative_task(row,  col-1, SE, global_cols), 0, seed);
   }
   if (is_border(borders, borderPM)) {
      orwl_read_insert(&handle[NWW],
            relative_task(row,  col-1, NE, global_cols), 0, seed);
   }
}
if (is_border(borders, border0P)) {
   orwl_read_insert(&handle[EE],
         relative_task(row,  col+1, W, global_cols), 0, seed);
   if (is_border(borders, borderMP)) {
      orwl_read_insert(&handle[SEE],
            relative_task(row,  col+1, SW, global_cols), 0, seed);
      }
   if (is_border(borders, borderPP)) {
      orwl_read_insert(&handle[NEE],
            relative_task(row,  col+1, NW, global_cols), 0, seed);
      }
   }
}
orwl_schedule(myloc, 1, srv);
```

Furthermore, it is difficult for non-experienced developers to achieve good parallel performance results, be it by using ORWL or any other programming framework. Dido also allows us to incorporate our ORWL domain-specific knowledge in a code generation framework that helps the user achieve high performance, in addition to productivity benefits.

CHAPTER 5

Benchmarks & Real-World Applications

In this work, we suggest a framework that generates parallel stencil code for distributed-memory architectures. It supports a wide range of iterative multi-dimensional stencil computations. This Chapter gives an overview of the benchmarks and real-world applications that we have considered to evaluate both the expression power and efficiency of our suggested framework. Section 5.1 presents the benchmarks that we have considered as well as their characteristics. In section 5.2, we describe two applications that we have used in this work to prove the validity of our approach.

5.1 Benchmarks

In this section, we present the benchmarks that we have considered in this work to validate our approach and prove the efficiency of the generated code. In the following, the superscript t denotes the discrete time step or iteration number, the subscripts $x, y, z, etc.$ denote the spatial coordinates, ∇^2 denotes *Laplacian* and the $\pm$ notation is a short-hand that we use here to save space.

5.1.1 Heat Transfer Equations

The heat equation $\frac{\partial}{\partial t}u = \kappa\nabla^2 u$ describes the temperature variation over time given initial temperature distribution and boundary conditions. If we assume that the heat conduction coefficient κ is constant and that there are no heat resources, the equation might be solved by applying an explicit finite difference scheme on a uniform mesh of points.

- **2D Heat Transfer:**

The 2D heat equation can be solved using ADI (Alternating Directions Implicit) [DOR+99] method. It consists of combining two implicit half-steps in one timestep,

Listing 5.1 — Livermore Kernel 23.

```
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < M-1; ++j ) {
    q = data [i-1][j] * zb [i][j]
      + data [i] [j-1] * zv [i][j]
      + data [i][j+1] * zu [i][j]
      + data [i+1] [j] * zr [i][j]
      + zz [i][j] - data [i][j];
    data [i][j] += 0.175 * q ;
  }
}
```

one in each direction (`x` or `y`). It results in the following *Jacobi-style* equation:

$$\begin{aligned} u_{x,y}^{t+1} &= u_{x,y}^{t} \\ &+ c_x.(u_{x-1,y}^{t} + u_{x+1,y}^{t} - 2u_{x,y}^{t}) \\ &+ c_y.(u_{x,y-1}^{t} + u_{x,y+1}^{t} - 2u_{x,y}^{t}) \end{aligned}$$

- **3D Heat Transfer:**

The 3D heat equation discretization results in the following *Jacobi-Like* equation:

$$u_{x,y,z}^{t+1} = c_0.u_{x,y,z}^{t} + c_1.(u_{x\pm1,y,z}^{t} + u_{x,y\pm1,z}^{t} + u_{x,y,z\pm1}^{t})$$

5.1.2 LINPACK Livermore

The *Livermore Kernel 23*, a classic benchmark taken from LINPACK, has always been considered as a reference benchmark for ORWL in previous works [CG10a, GVM14]. As shown in Listing 5.1, it is a 5-point Stencil. Each element is updated by the element at the same position from the previous iteration and 4 neighbors offset by 1 on each direction. This stencil algorithm is cache-unfriendly. It has the particularity that data grows quickly in memory, as it requires a set of five coefficient grids (`zb`, `zv`, `zu`, `zr` and `zz`) in addition to the *main data* grid, which makes the overall memory footprint bigger.

5.1.3 Wave Equations

The classic linear wave equation $\frac{\partial^2}{\partial t^2}u - c^2 \triangle u = 0$ is a second-order partial differential equation that arises in multiple fields such as acoustics, electromagnetics, and fluid dynamics. It describes the physics of waves including water waves, sound waves and light waves.

- **2D Wave Equation**

A second-order-in-time discretization of the 2D wave equation results in the following 2D second-order stencil. In addition to data from two previous timesteps, an auxiliary data grid $v_{x,y}$ corresponding to the inverse of velocity squared is required.

$$u^{t+1}_{x,y} = 2u^{t}_{x,y} - u^{t-1}_{x,y} + v_{x,y}.(c_0.u^{t}_{x,y} + c_1.(u^{t}_{x\pm1,y} + u^{t}_{x,y\pm1}))$$

- **3D Wave Equations**

A second-order in time discretization of the 3D wave equation results in a 3D order-k stencil where k = 2, 4, 6, 8, 10 or 12.

$$u^{t+1}_{x,y,z} = 2u^{t}_{x,y,z} - u^{t-1}_{x,y,z} + v_{x,y,z}.(c_0.u^{t}_{x,y,z} + \sum_{i=1}^{k/2} c_i.(u^{t}_{x\pm i,y,z} + u^{t}_{x,y\pm i,z} + u^{t}_{x,y,z\pm i}))$$

where $v_{x,y,z}$ is the inverse of velocity squared.

5.1.4 Gauss-Seidel Methods

Solving Laplace equation using Gauss Seidel method results in the following equations:

- **2D Gauss-Seidel**

$$u^{t+1}_{x,y,z} = c_0.u^{t}_{x,y,z} + c_1.(u^{t+1}_{x-1,y} + u^{t+1}_{x,y-1} + u^{t}_{x+1,y} + u^{t}_{x,y+1})$$

- **3D Gauss-Seidel**

$$u^{t+1}_{x,y,z} = c_0.u^{t}_{x,y,z} + c_1.(u^{t+1}_{x-1,y,z} + u^{t+1}_{x,y-1,z} + u^{t+1}_{x,y,z-1} + u^{t}_{x+1,y,z} + u^{t}_{x,y+1,z} + u^{t}_{x,y,z+1})$$

- **Gauss-Seidel Red-Black (GSRB) Laplacian**

Gauss-Seidel Laplacians, resulting from high-order discretization of the 3D Laplacian, are often used as smoothers in multigrid methods. The following 3D 25-point stencil equation corresponds to two iterations of a red-black Gauss-Seidel iteration collapsed into one.

$$u'_i = a u_i + b \sum_{|j|_1=1} u_{i+j} + \sum_{r=1}^{2} c_r \sum_{\substack{|j|_1=2 \\ j_k\in\{0,\pm r\} \\ k=1,2,3}} u_{i+j}, \qquad i,j \in Z^3$$

where $j = (j_1, j_2, j_3) \in Z^3$ is an offset vector, $a = \alpha + 6\beta^2$, $b = \alpha\beta$, $c_1 = 2\beta^2$ and $c_2 = \beta^2$.

5.1.5 27-point 3D stencil

Eventhough the simple 7-point 3D stencil is fairly common, there are many instances with compute-intensive stencils that require more neighboring points. For instance, the NAS Parallel MG (Multigrid) benchmark utilizes a 27-point stencil to calculate the Laplace operator for a finite volume method [BBB+91].

5.1.6 4D+ Jacobis

The framework suggested in this work supports multi-dimensional stencils. For that, we consider as benchmarks 4D 9-point Jacobi and 5D 11-point Jacobi .

5.1.7 Summary

Table 5.1 presents a summary of the characteristics of the considered benchmarks.

5.2 Real-World Applications

The expressive power of our suggested framework exceeds the scope of benchmarks. It also targets real-world stencil-based applications. In this work, we consider two real-world applications that we describe in the following sections.

5.2.1 Cell Nuclei Recognition in Breast Cancer Images

Breast cancer is the most common cancer affecting women on a worldwide scale, with over 1.6 million new cases per year.Early detection greatly increases the chances of successful treatment. Nowadays, diagnosis is mostly performed through a microscopic slide analysis of biopsy tissue. Several criteria are observed in order to grade the cancer on a scale of 1 to 3. One of these criteria is the size of cell nuclei that beyond a certain limit is considered as anomalous. Given the large number of stained biopsies that have to be analyzed every day in health-care institutions, efficient methods for automatic nuclei detection can be very helpful to fight the disease and monitor its progression.

A previous work [AFB13] has suggested an algorithm that automatically detects cell nuclei abnormalities by analyzing scans of biopsy slides (*cf.* Figure 5.1). The suggested algorithm is based on the Marked Point Process (MPP) which is a promising tool usually used to extract objects in remote sensing. The Marked Point Process (MPP) is a stochastic process that models objects that are random in number, random in geometry parameters and randomly localized. The algorithm involving a birth-death process is composed of five main steps as follows:

Table 5.1 — Benchmark Characteristics Summary

Benchmark	# of points	Operations per Point	Memory Accesses	Arithmetic Intensity	Grid Traversal	Neighborhood	Boundary Conditions
2D Heat ADI	5-pt	4(+), 2(-), 4(×)	1 (out), 1 (in)	5.0	Jacobi	von Neumann	non-periodic
3D Heat	7-pt	6(+), 2(×)	1 (out), 1 (in)	4.0	Jacobi	von Neumann	non-periodic
Livermore	5-pt	5(+) 1(-) 5(×)	1 (out), 6 (in)	1.57	Jacobi	von Neumann	non-periodic
2D Wave	5-pt	5(+), 1(-), 4(×)	1 (out), 3 (in)	2.5	Jacobi	von Neumann	non-periodic
3D order-k Wave	$(3k+1)$-pt	$(1+3k)$ (+), 1 (-), 4 (×)	1 (out), 3 (in)	$(3k+6)/4$	Jacobi	von Neumann	non-periodic
2D Gauss-Seidel	5-pt	4(+), 2(×)	1 (in/out)	6.0	Seidel	von Neumann	non-periodic
3D Gauss-Seidel	7-pt	6(+), 2(×)	1 (in/out)	8.0	Seidel	von Neumann	non-periodic
3D GSRB	25-pt	24(+), 4(×)	1 (in), 1 (out)	14.0	Jacobi	Moore	non-periodic
3D 27-pt Jacobi	27-pt	26 (+), 2(×)	1 (in), 1 (out)	14.0	Jacobi	Moore	periodic
4D 9-pt Jacobi	9-pt	8 (+), 2(×)	1 (in), 1 (out)	5.0	Jacobi	von Neumann	periodic
5D 11-pt Jacobi	10-pt	10 (+), 2(×)	1 (in), 1 (out)	6.0	Jacobi	von Neumann	periodic

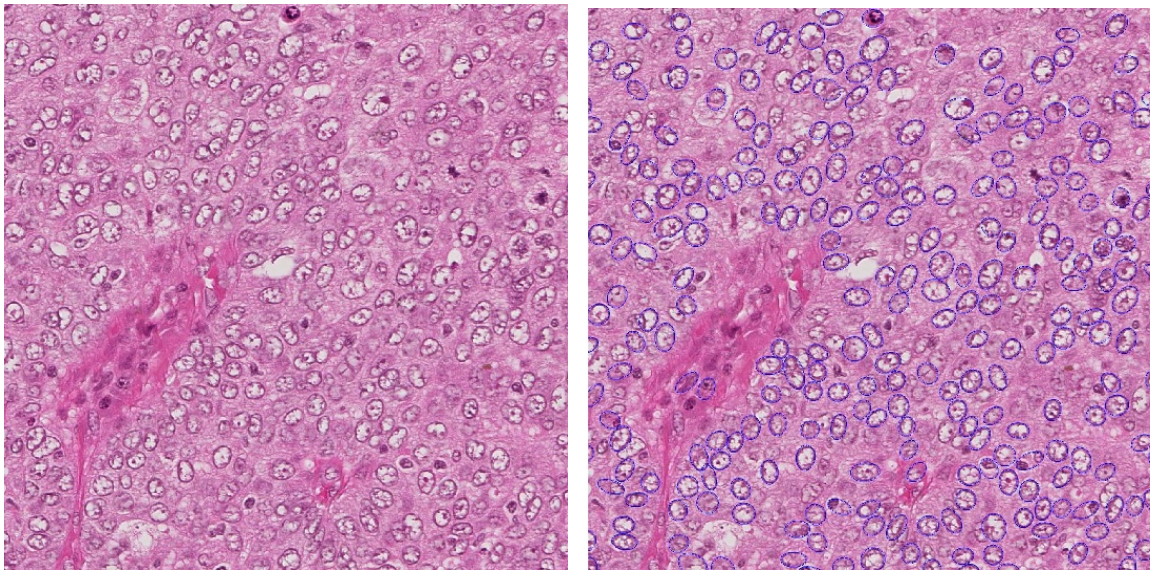


Figure 5.1 — Example of cell nuclei detection using the Marked Point Process [AFB13]

1. *Birth:* Generation of *birth points* that are randomly distributed throughout the image following a *birth coefficient* which is a probability that decreases at each iteration to favor convergence. A cell nuclei has approximately the shape of an ellipse. Thus, every point created in the *birth step* is randomly given a major axis, a minor axis and an inclination angle.

2. *Data-Fidelity Computation:* For each ellipse, a *data fidelity value*, also called *attachment*, is computed depending on its geometry parameters. It is also necessary to calculate the *Bhattacharyya distance* [ATR98] to measure the intensity difference between the inner and outer borders of an ellipse.

3. *Cover Map Update:* Based on the *data fidelity values*, a *cover map* indicating the number of ellipses overlapping each pixel is drawn. A map value of a given pixel inside an ellipse is updated when it is higher than the ellipse data fidelity value.

4. *Death:* First, the less attached ellipses are killed. Then, a random filter is applied. It keeps only a few percent of the surviving ones. This filter depends on both the *data fidelity value* and the *death probability* which is a coefficient that increases at each iteration to favor convergence.

5. *Convergence Step:* Steps 1 to 4 are iteratively repeated. At each iteration, surviving ellipses are competing with the newly created ones. Convergence is reached when all born ellipses in one superstep are killed within the same iteration.

In [AFB13], Avenel *et al.* have parallelized the suggested algorithm for nuclei detection on CPU using OpenMP and on GPU using CUDA. Though, because of memory limitations, a single GPU is not able to support the analysis of a complete biopsy slide that

may reach $10^5 \times 10^5$ pixels at full resolution. Distributing the algorithm becomes thus inevitable.

The main challenge in distributing the algorithm lies in the *Cover Map Update* step as it consists of a nearest-neighbor computation, hence a stencil. The *cover map* containing *data fidelity values* is distributed among nodes. Updates of neighboring blocks newly computed values are needed.

A handwritten implementation using Ordered Read-Write Locks (ORWL) paradigm has been suggested in [SGRP16]. In this work, we prove that Dido has the expression power to handle such applications. We succeeded in generating a code that is equivalent to the handwritten code used in [SGRP16]. Unfortunately, no performance results are provided for this application.

5.2.2 Molecular Dynamics

Molecular dynamics (MD) is a widely spread method used to simulate a particle system and analyze the dynamical properties of particles. It is heavily used in multiple areas such as biology [Ber96], materials [KLH+04] and chemistry [Rap04]. It presents the particularity of providing an individual particle view of the simulation system, which is not reachable through laboratory experiments. MD simulations study the trajectory of particles by numerically time-integrating Newton's equation of motion [HDS96] defining the force on one particle as an interaction of all others.

Large-scale MD simulations may involve billions of particles and thus require a high computation power only available on distributed-memory clusters. For that, writing MD code for distributed-memory architectures becomes often inevitable. The MD algorithm is composed of seven main steps as follows:

1. Compute particle positions.
2. Test the periodicity and assign particles to cells.
3. Compute the partial velocities.
4. Compute the forces.
5. Compute the kinetic energy.
6. Compute and normalize the velocities.
7. Compute the instantaneous physical quantities.

Step 2 and step 4 of the MD algorithm, *i.e.*, force computations and periodicity test and assignment to cells, present nearest-neighbor dependencies.

In this work, we use Dido, an implicitly parallel domain-specific language for stencil computations, to automatically generate MD distributed code. In our implementation,

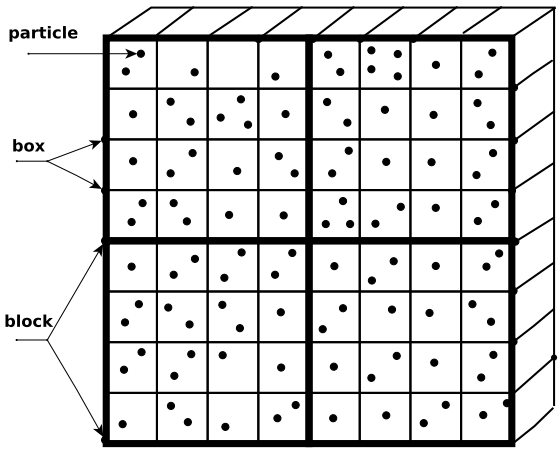


Figure 5.2 — Particle-Box-Block Partitioning

we consider three-body forces, which are more compute-intensive than two-body forces only. We also consider that beyond a certain distance called the cutoff distance Rc, two particles are considered as non-interactive. This means that only pairs and triplets of particles within Rc are taken into account, reducing the computation of the forces to a $O(N)$ complexity where N is the total number of particles.

We uniformly divide the spatial simulation domain into boxes where each box contains the particles that physically belong to its space as shown in Figure 5.1. The main data consists then of a grid of boxes through which it is necessary to go in order to access the particles. In several steps of the MD algorithm, the same attribute of several particles is accessed at once. For this reason, we chose to organize a box of P particles as a structure of arrays (SOA), *i.e.*, several separate arrays of length P, one array for each particle attribute. Each box is then able to manage its own attributes independently. This modeling makes parallelization easier. Since particles may enter and leave the box, this latter has the behavior of a list. However, the number of particles in a box quickly stabilizes and oscillates around a mean value. For each particle, there are in total fourteen physical quantities: three spatial coordinates, velocity in each dimension, acceleration in each dimension, forces in each dimension in addition to mass and temporary quantities.

As particles beyond cutoff distance Rc are considered as non-interactive, each particle belonging to one box interacts only with particles belonging to its 27 neighboring boxes. However, the edges of each box must be slightly longer than the cutoff distance. As depicted in Figure 5.2, we combine several boxes together forming blocks. Each block is a sub-grid of the *main data* grid. The system is considered as infinite by imposing periodic boundary conditions, *i.e.*, when a particle enters/leaves the simulation domain, an image particle leaves/enters respectively, in a way that the number of particles remains constant.

In this work, we succeeded in automatically generating MD parallel code for distributed-memory architectures. Performance results are given in Chapter 11.

CHAPTER 6

Automatic Code Generation and Optimization Frameworks for Stencil Computations

Numerous research efforts have focused on improving stencil computation performance, either automatically or manually. The stencil computation methods and frameworks suggested in previous works fall into three categories:

- Hand-coded implementations that strive to achieve the highest possible performance for one specific stencil kernel.
- Implementations that focus on tuning particular parameters for achieving the highest possible performance.
- Implementations that target user-friendliness and ease of programming by building domain-specific frameworks.

In fact, parallel programming turns out to be considerably more difficult than sequential programming, be it on shared-memory or distributed-memory architectures. In addition, achieving portability and performance is more and more challenging given the diversity of the current and emergent parallel architectures. As a result, domain-specific frameworks have been widely used in recent years to spare the user the details and complexity related to parallel programming. In particular, numerous research efforts have adopted domain-specific solutions to optimize stencil computations. Some of the suggested frameworks automatically generate stencil optimizations and transformations for GPUs and multicore processors. Others present auto-tuning frameworks for stencil computations. A few other frameworks target distributed-memory architectures. Some other works focus on stencil-like image processing applications.

Existing domain-specific frameworks have taken two approaches according to the form of the user input. One approach consists in building a completely new programming language, *i.e.*, a DSL. The other consists in adding a small set of extensions, *e.g.*,

pragma annotations, on an existing general-purpose language to build an an embedded DSL While external DSLs enable a maximally optimized language design that is tailored to the domain-specific needs, embedded DSLs are limited to the traditional programming language basic syntax. On the other hand, embedded DSLs leverage existing compilers rather than developing and maintaining a separate compiler as for external DSLs. They also inherit syntax, operators, and operator precedence that are well-defined, well-documented, and well-understood by the general-purpose language programmers.

In this chapter, we provide an overview of automatic code generation and optimization frameworks for stencil computations. Section 6.1 describes domain-specific frameworks that automatically generate stencil optimizations and transformations for GPUs and multicore processors. Section 6.2 presents domain-specific languages for stencil-based image processing applications. In section 6.3 provides an overview of the autotuning frameworks for stencil computations. Section 6.4 describes the few available domain-specific frameworks that generate code for distributed-memory architectures.

6.1 Stencil Code Optimization Frameworks

Multiple research works have targeted stencil code optimizations in order to improve data locality and parallelism, optimize data movements, and control the synchronization frequency and communication volume when applicable. The optimizations vary from blocking and tiling [DMV+08, SC04, RT00, WCO+08, KW01, DHK+00] to more advanced techniques such as cache oblivious, time skewing, wavefront or overlapped tiling [WLSG+06, NSC+10, DKW+09, FS05, SL99, Won00, WHZ+09, ZWN+08, GKV12, ZGG+12, KBB+07, Mic09].

Despite these numerous research efforts and the proven gains in performance, using these achievements is not straightforward as they introduce significant complexities in the stencil code, which adds a difficulty layer to the parallelism. Writing the code becomes too complex and error-prone for average programmers. In this section we provide an overview of the most common stencil optimizations and the state-of-the-art frameworks that enable the automatic generation of those optimizations for both multiprocessors and GPU architectures.

6.1.1 Stencil Optimizations

Stencil computations are often bound by capacity misses in cache. However, the stencil data access pattern is regular and logically-related data are often near in memory. Therefore, loop tiling [Xue12], also called loop blocking, is a key transformation for stencil computations. Loop tiling techniques have proven to offer the potential to generate large performance speedups by enhancing data locality and parallelism, and reducing cache misses [GAK03, HCC+09, LS04, RKRS07, RT00].

Other techniques aim at increasing temporal locality, *e.g.*, time tiling that consists in

tiling along the time dimension, which also enhances data locality and performance. To make time-tiling of stencil computations legal, loop skewing [Won00] is often required. Different time tiling schemes for stencil computations on a variety of hardware architectures, *i.e.* GPUs and shared memory CPUs, are discussed in [DMV+08, CSN+10, MS11, WHZ+09]. Time-tiling is often also coupled with ghost-zone optimizations, such as overlapped tiling and split tiling in order to preserve inter-tile concurrency [Won00]. However, combining parallelization and locality optimization techniques often results in pipelined startup of tasks, leading to idle cores during parallelized execution. Diamond tiling is one of the techniques used to avoid pipelined startup and allow concurrent start of tiles [BBP17]. It also ensures load balancing between two synchronization points.

Other more advanced optimization techniques for stencil computations exist. For instance, cache-oblivious algorithms [FLPR99] can be designed to optimally use a CPU cache without having its size as a predefined parameter. Unlike explicit blocking which optimally divides a problem into blocks that fit a given cache size, cache-oblivious algorithms rely on divide and conquer algorithms to recursively divide the problem until reaching a subproblem size that fits into the cache. One way to avoid machine-specific tuning, is to build cache-oblivious algorithms.

6.1.2 Stencil Code Optimization Frameworks for Multiprocessors

Pochoir [TCK+11] is a compiler and runtime system for implementing multidimensional stencil computations on multicore processors. The Pochoir compiler allows programmers to write stencil specifications in a domain-specific language embedded in C++ and translates it into high-performing Cilk code. Behind Pochoir, there is TRAP, a cache-oblivious divide-and-conquer algorithm based on recursive trapezoidal decompositions introduced by Frigo and Strumpen [FLPR99, FS05]. TRAP employs, in addition, a hyperspace-cut strategy that yields asymptotically more parallelism without sacrificing cache-efficiency. It consists of applying parallel space cuts simultaneously to as many dimensions as possible, instead of only one parallel space cut at a time. Pochoir applies, in addition, multiple optimizations such as code cloning, loop index calculations and coarsening the base case of recursion. However, Pochoir is limited to shared-memory architectures and does not achieve as much speedup over the loop code in cases where the stencil contains conditionals or high ratio of memory accesses compared to floating-point operations.

6.1.3 Stencil Code Optimization Frameworks for GPU accelerators

Given the high regularity and locally contained dependence pattern of stencil computations, these computations can achieve high performance benefits from a parallelization on GPU accelerators. However, writing parallel code for these architectures presents a higher degree of complexity and remains challenging even for highly skilled experts. Low-level programming models are commonly used to write GPU programs. The two most common models are CUDA [Nvi10] and OpenCL [SGS10]. Since GPU architectures are

massively threaded, performance on these accelerators can easily be degraded in case of branch divergence or lack of memory coalescing. Hence, efficient GPU programs typically involve the scheduling of a large number of concurrent threads to hide memory latency. In particular, stencil computations require special attention to data movements. Locality optimizations are thus required to keep data close to processor cores and ensure the overlapping of communications and computations. Several recent efforts aimed to provide high-level abstractions to make GPU programming easier without sacrificing performance. The suggested frameworks can be divided into two groups, *i.e.*, DSLs and embedded DSLs.

6.1.3.1 Embedded DSLs

PSkel [PRG15] is an API for high-level stencil programming implemented as a C++ template library. It is based on parallel skeletons and provides abstractions for developing parallel stencil computations on CPU-GPU systems. PSkel enables the programmer to partition tasks and assign data and computation across CPU and GPU. It also provides templates for manipulating input and output data, specifying stencil masks, encapsulating memory management, computations, and runtime details.

Mint [UCB11] also embodies a domain-specific approach for stencil methods. It consists in a source-to-source translator that generates optimized CUDA C from annotated C code. The user has only to annotate their traditional C source with intuitive Mint directives. Thanks to the incorporated domain-specific knowledge, Mint generates efficient CUDA C code that delivers performance that is competitive with hand-optimized CUDA.

In [GCK+13], Grosser *et al.* present a static code generation approach for stencil computations that is implemented as a prototype extension of the PPCG tool [VCJC+13], an existing polyhedral code generator targeting GPUs. The framework applies split tiling over both time and parallel loops to take advantage of the parallelism and local memory resources of modern GPUs, without the need for skewing or redundant computations. The underlying algorithm deduces index-set splitting from dependence vectors, then applies tiling. The generated code keeps intermediate results of several iterations of the time loop in shared memory, which significantly reduces the pressure on the global memory bandwidth and therefore increases the computational throughput.

Physis [MNSM11] is also a source-to-source translator that targets GPU-based heterogeneous systems. It takes as input annotated C code and generates C for CPU execution, CUDA code for GPU acceleration and MPI for node-level parallelization. It also generates automatic optimizations such as computation and communication overlapping.

Previous work also includes Nebo [MKCK00] that targets partial differential equations in particular. Nebo [EMBS17] is a declarative domain-specific language embedded in C++ solving partial differential equations for transport phenomena such as computational fluid dynamics on structured meshes. It supports single-thread execution, multi-thread execution, and many-core GPU-based execution. Eventhough Nebo handles data parallelism, it does not provide memory management or inter-node communication. It also

keeps data transfers between CPU and GPU to the user.

6.1.3.2 DSLs

Holewinski et al. [HPS12] introduce a DSL and its associated compiler algorithms that automatically generate stencil code for GPU accelerators, including both nVidia and AMD devices. The code generation follows a predefined scheme that uses overlapped tiling to generate time-tiled efficient code.

In [HHV+, HVF+13], Henretty *et al.* present a DSL for stencil computations combined with its multi-target compiler that generates code for GPUs and short-vector SIMD ISAs (e.g., SSE, AVX). Given the difference between these two types of architecture, different target-specific optimization strategies are apllied by the compiler. Split tiling is used for multi-core CPUs, while overlapped tiling is used for GPUs, achieving data reuse and parallelism along spatial and time dimensions. Efficient stencil implementations on short-vector SIMD ISAs are particularly challenging as they involve arithmetic operations on physically contiguous data elements. In fact, vector operations on SIMD devices require the loading of contiguous data elements from memory into different slots in different vector registers, and the execution of identical and independent operations on the components of vector registers. The suggested DSL overcomes this problem by combining tiling with dimension-lifting-transpose (DLT) transformation [HSP+11], in a way that operands that need to be combined are located in the same slot of different vectors.

6.2 Image Processing Frameworks

Many image processing applications can be viewed as pipelines consisting of several interconnected processing stages that follow the stencil computational pattern. A number of previous works suggest domain-specific languages for this particular domain, *i.e.*, image processing applications.

Polymage [MVB15] is a domain-specific language for image processing applications based on the polyhedral model. It allows the user to express the commonly used image processing computation patterns in the form of point-wise operations, stencils, up-sampling, down-sampling, histograms, and time-iterated methods. Functions within Polymage are defined by a list of cases where each case construct takes a condition and an expression as arguments. Conditions are used to specify constraints involving variables, function values, and parameters. The compiler takes as input the program specified in the Polymage syntax and the names of the live out functions. The image processing pipelines are represented internally as a directed acyclic graph (DAG), where each node represents a stage specified by the user and the edges represent the dependencies. The polyhedral representation is then extracted from the specification. After selecting a tiling strategy based on a set of criteria, the transformed C++ code is generated. Finally, automatic tuning is performed to choose the right tile size.

Halide [RKBA+13] is a domain-specific language and compiler for image processing

pipelines. It targets multiple architectures, *i.e.*, x86, ARM and GPUs. It offers trade-offs between locality, parallelism, and redundant recomputation in stencil pipelines. Halide compiler requires, in addition to the problem specification, a scheduling representation. Given the input specification, a loop synthesizer based on range analysis is used for data parallel pipelines to produce a complete loop nest and corresponding allocations. Then, bounds of each dimension are obtained in a recursive manner, by interval analysis of the expressions in the caller which index that dimension. The compiler performs later sliding window optimization and storage folding to improve data reuse when possible. The final code is generated after vectorization and unrolling optimizations. An autotuner based on stochastic search is finally applied to automatically find a fitting schedule.

These frameworks target image processing and provide performance and programmer productivity benefits for this particular applicative domain. When compared to our approach, Polymage and Halide are specialized and limited to image processing applications, while Dido is more general. Dido has the expression power to model image processing applications, *e.g.* Cell Nuclei Recognition, in addition to other stencil kernels such as PDE solvers. Unlike Dido syntax, Polymage and Halide specifications are also complicated and not intuitive. Additionally, they do not also support distributed-memory architectures

6.3 Autotuning Frameworks

Given the growing diversity and complexity of nowadays micro-architectures, architecture-specific tuning is often necessary to fully leverage the computation power of the current machines. However, manual tuning is costly in time and effort and requires a deep understanding of the specific architecture. Given the breadth of architectures, stencil kernels and problem sizes, creating a separate hand-tuned stencil code is both time-consuming and error-prone. Automatic tuning, also called autotuning, could alleviate this burden. Autotuners are often designed to maximize performance metrics, but can also be designed to maximize other metrics, *e.g.* power efficiency. For instance, in [TLCS11], software and hardware facilities are used to tune applications for several combinations of power and performance.

Building an auto-tuner requires extensive domain-specific knowledge in order to determine which code transformations are legal and potentially useful. Additionally, building a code generator that generates the selected optimizations can be challenging, especially when combining different optimizations for a co-parameter space. However, the one-time cost invested in building an autotuner can easily be compensated by the performance portability. Auto-tuners are often designed to be scalable and support any number of cores, making them achieve good performance on nowadays architectures, but also other architectures supporting similar programming models, *e.g.* manycore architectures.

Autotuning consists in automatically searching and selecting the code variant that achieves the best performance among a set of possible versions. In other terms, a benchmarking executable is built and run on the target machine. Autotuning can be per-

formed either offline (*e.g.* ATLAS [TCC+09] and FFTW [FJ05]) or at runtime (*e.g.* Active Harmony [ŢCH+02]). The code versions can range from determining the best configuration for an optimizing compiler [PE06], to re-parameterizations that have an impact on performance-related tunables, *e.g.*, cache tiling factors and loop unrolling factors as for Patus [CSB11] to code transformations, which can be achieved by a source-to-source transformation framework such as CHiLL [CCH08].

Autotuning had proved successful in creating high performance codes for multiple scientific computing kernels, including dense and sparse linear algebra and discrete transforms. Among the most reknown production-quality auto-tuning libraries and code generators, there is ATLAS [TCC+09] for dense linear algebra, OSKI [VDY05] and PHiPAC [WD98] for sparse linear algebra and FFTW [FJ05] and SPIRAL [VDY05] for spectral algorithms. Eventhough the computation structure of stencil computations maps well to current hardware architectures, meticulous architecture-specific tuning is still required to fully leverage the computational power of the platform. Previous efforts have successfully developed stencil-specific auto-tuners. These can be divided into two categories, *i.e.*, DSLs and embedded DSLs.

6.3.1 Embedded DSLs

In [KCO+10], Kamil et al. suggest a source-to-source multi-target framework for auto-parallelizing and auto-tuning multi-dimensional stencil loops. It takes ordinary Fortran 95 as input, and produces Fortran, C, or CUDA code. It targets both multicore architectures and GPGPUs. The tool uses an intermediate abstract syntax tree (AST) representation of the Fortran stencil problem specification to explore possible auto-tuning transformations. The framework achieves perfomance gains over the reference sequential implementation. However, the generated code uses GPU device memory only and does not take advantage of shared memory. Additionally, the suggested framework does not support distributed-memory architectures.

Datta et al. [DMV+08] developed an optimization and auto-tuning framework for stencil computations, targeting multi-core systems, NVidia GPUs, and Cell SPUs It generates stencil optimizations, including core blocking, thread blocking, register blocking, NUMA-aware data allocation, array padding, software prefetching, cache bypass, SIMDization, and common subexpression elimination.

6.3.2 DSLs

PATUS [CSB11] is a code generation and auto-tuning framework for stencil computations on multicore processors and GPUs. In PATUS, the user describes the stencil kernel in a C-like syntax, and either chooses one of the parallelization and optimization predefined strategies, or specifies a customized strategy. PATUS then generates C code from strategy templates and optimizes strategy-dependent parameters for the specific hardware architecture in use. In fact, the code generator creates in a first step a benchmark harness from

a strategy-specific templated implementation. Then, the autotuner repeatedly runs the executable and measures the runtime of the stencil kernel, while varying the autotuning parameters specified in the strategy definition. The aim is to find the optimal configuration with minimal running time for the strategy-dependent parameters in addition to internal parameters such as loop nest unrolling and padding factors. Thanks to the incorporated domain-specific knowledge, Patus enables code optimization beyond the abilities of current compilers. It can thus be considered as an experimentation toolbox for parallelization and optimization strategies. However, PATUS has some limitations. First, it is restricted to shared memory architectures, that are multicore CPU and single-GPU systems. It supports traditional CPU architectures and NVIDIA GPUS using OpenMP and CUDA respectiveley for parallelization. Additionally, unlike Dido, it is limited to shared-memory, Jacobi iterations, Dirichlet-type boundary conditions.

6.4 Distributed-Memory Frameworks

Despite the high regularity of their computational pattern, programming stencil computations on distributed-memory architectures remains challenging. It involves the definition of a data partitioning scheme in addition to explicit synchronization and communication to manage the data movement between the different processors. This can be tedious, complex and error-prone. There is a significant need in alleviating this burden by automatically generating stencil code for distributed-memory architectures.

One of the few stencil code generation frameworks that support distributed-memory architectures is presented in [DRRB13]. It consists in a source-to-source transformer that translates sequential affine loop nests to parallel code using the asynchronous Message Passing Interface (MPI) primitives for communication. It is implemented as an extension for Pluto [Bon09]. It generates code for distributed memory clusters as well as CPU and GPU heterogeneous systems where CPUs act as orchestrators of data movement between compute devices. The communication code generation is based on non-trivial data dependency partitioning techniques described in [Bon13]. This method statically determines the data to be transfered between compute devices in oder to avoid both unnecessary and duplicate data from being communicated. In fact, the data dependencies are classified into three types; i.e, Read-after-Write (RAW), Write-after-Read (WAR), and Write-after-Write (WAW). The framework generates the communication code using only WAW dependences. Bondhugula proved in [Bon13] that these are sufficiant for efficient communication code generation and result in the minimum communication volume to preserve program semantics for a vast majority of cases. The tool relies on polyhedral tools such as Clan [Bas08], ISL [Ver10], Pluto, and Cloog-isl [Bas13] to extract the polyhedral representation and data dependencies, transform the input code and generate the output code. The input code is tiled and parallelized using Pluto not only to improve locality and increase the parallelism granularity, but also to reduce the frequency of communication and the bound buffer sizes. Polylib [Loe99] is used to implement the polyhedral operations and ISL [Ver10] is used to eliminate transitive dependences and compute

last writers or the exact dataflow. The framework is scalable and achieves performance that is close to manually written code and exceeds it in some cases.

Previous work also includes ParAgent [MKCK00], a source-to-source code generation framework that targets finite difference methods (FDM) in particular. It takes as input Fortran-77 programs based on the explicit time-marching FDM and produces parallel programs for distributed-memory computers. It uses static analysis to help minimize communication between compute nodes.

In [LAB+14], Christian Lengauer et al. present an ongoing project that aims to realize a platform based on a domain-specific approach targeting stencil computations for exascale systems. ExaStencils project follows Wirth's notion of stepwise refinement [Wir71], *i.e*, the platform is composed of multiple layers of abstraction, starting from the mathematical statement of the stencil computation to the code to be executed on the target platform. Each layer is associated with a domain-specific code generation and optimization step. ExaStencils follows a path of refinement steps and makes choices at the different layers relying on the combined knowledge base of domain experts, mathematicians, and software and hardware specialists. It recommends suitable combinations of configuration option such as algorithmic components, alternatives of data structures, and parameter values based on a machine-learning approach. The aim is to detect and handle explicitly interactions among those configuration options through a small number of concrete stencil-code variants. Then, a weaving algorithm applies optimizations according to the made choices. Among the suggested optimizations, ExaStencils uses the polyhedron model for automatic loop parallelization and optimization. However, Exastencils is for the moment prototyped but still not implemented. The preliminary version generates C++ code with OpenMP and CUDA. The prototype works only for shared-memory architectures. The version targeting distributed memory architectures is still not implemented.

When compared to our approach, all of the previously enumerated frameworks for distributed memory-architectures are embedded domain-specific languages, while Dido is a domain-specific language. Additionally, all of them use MPI as a message passing programming model, whereas Dido uses ORWL as a programming backend.

Beyond the scope of stencils, *i.e.*, structured grids, Listz is a domain-specific language for solving partial differential equations (PDEs) on unstructured meshes. Unlike affine partitioning, Listz uses the logic of mesh-topology rather than affine transformations to automate the analysis. It captures the data-parallelism of the mesh, the locality of the PDE stencil, and the synchronization of dependencies that occur between phases of an application. It provides language statements for interacting with an unstructured mesh, and storing data at its elements. The program analyses results in a partitioning method based on message passing and a scheduling method based on graph coloring. The Liszt compiler then generates native code for multiple runtimes. Listz targets different architectures*i.e.*, clusters, SMPs and GPUs. It therefore supports different parallel programming models, *i.e.*, MPI, pthreads, and CUDA. However, Liszt supports for the moment only a small subset of applications. It lacks support for implicit methods and linear solvers.

CHAPTER 7

Dido Code Generation Pattern

The aim of this work is to automate the generation of multi-dimensional stencil code with ORWL as communication and runtime back-end. To this end, we needed to define a generalized use pattern that fulfills the liveness properties for ORWL stencils. For this purpose, deep knowledge of the domain semantics was needed. We had first to analyze ORWL 2D and 3D stencil implementations and extract the constraints they must meet. Then, the extracted pattern was generalized for multi-dimensional stencils.

In this Chapter, we suggest a pattern that ensures deadlock-freeness, liveness and efficiency for ORWL multi-dimensional stencil programs. The suggested pattern consists of the combination of a non-trivial data partitioning scheme with an iterative operation form that we call *CompUp*. Section 7.1 introduces the *CompUp* form. In Section 7.2, we present the overlapped data partitioning as the default data partitioning scheme for Dido. In Section 7.3, we address the inter-node data locality by applying the temporal blocking optimization for a class of stencil computations.

7.1 CompUp Form

To enhance the expressiveness of our tool and make code generation easier, we agreed to express iterative ORWL programs in a form that we named *CompUp*. Here, as shown in Figure 7.1, we cast an ORWL program in the form of three types of iterative operations: *Compute*, *Local Update* and *Global Update*.

1. The *Compute* operation performs the computation as specified by the application. It reads the locally accessible data that is imported by *Global Update* operations and saved in local buffers. Then, it executes the computation kernel and writes the results in the *main location*.

2. The *Local Update* operation ensures the data transfer between different resources of the same task. It reads the updated data from the *main location* and stores it in local buffers to make it available for neighboring tasks.

3. The *Global Update* operation performs the inter-task communication between the *main task* and neighboring tasks. It reads the data in the remote neighboring tasks' buffers, updated by their own *Local Update* operations, and writes it on local buffers, making it available for the next *Compute* operation.

In order to meet the *canonical form* constraints, careful efforts have been made to associate one single operation to each location, and thus one exclusive write access to each location. We add additional constraints on the initial positions that the previously enumerated operations take in the data location FIFOs. First, each operation should have the same initial position over all the resources it needs. Second, we impose that these initial positions of priorities follow the order above. Namely, the *Compute* operation has priority over the others. Then, the *Local Update* comes second to save the computed results on local buffers. The last priority is assigned to the *Global Update* operation. To sum up, the request orderings in the FIFOs are initially arranged with the positions indicated by the enumeration above, and then regenerated cyclicly as the iterations progress. In the following, we prove that by adding these constraints to the *CompUp* form, the subsequent computation is guaranteed to be *deadlock-free*.

In addition to the liveness guarantees that it provides, the *CompUp* form enables the automatic attribution of the FIFO initial positions. It has to be mentioned that the *CompUp* form is valid for different ORWL iterative implementations, not only for stencil computations, *e.g.*, graph processing. But, these are not the object of this work.

Proof

Apart from the assigned priorities, the *CompUp* form is made of three types of iterative operations: *Compute*, *Local Update* and *Global Update*. One single operation, and thus one exclusive write access, is associated to each location. Operations of the same type, *i.e.*, *Compute*, *Local Update* or *Global Update*, do not have any conflicts. As a result, operations can be trivially partitioned into three sets. We attribute one color to each set as shown in Figure 7.1. Algorithm 1 given in Chapter 2 generates the FIFO request ordering suggested above for the *CompUp* form operations. It also guarantees that the resulting *overlay* is deadlock-free. This can be easily verified by observing the corresponding *delay digraph* depicted in Figure 7.2. The *delay digraph* does not contain any cycles on its dependencies. As evoked in Subsection 2.2.3, this implies that the corresponding *overlay* is deadlock-free.

7.2 Overlapped Data Partitioning

As mentioned in Chapter 4, stencil parallelization within ORWL implies the definition of a number of *shadow locations*, in addition to the *main location* that undergoes the computation. Instead of the standard block partitioning presented in Section 4.4, we extend the *main location* to include the halo region elements. Each block is then enlarged by twice

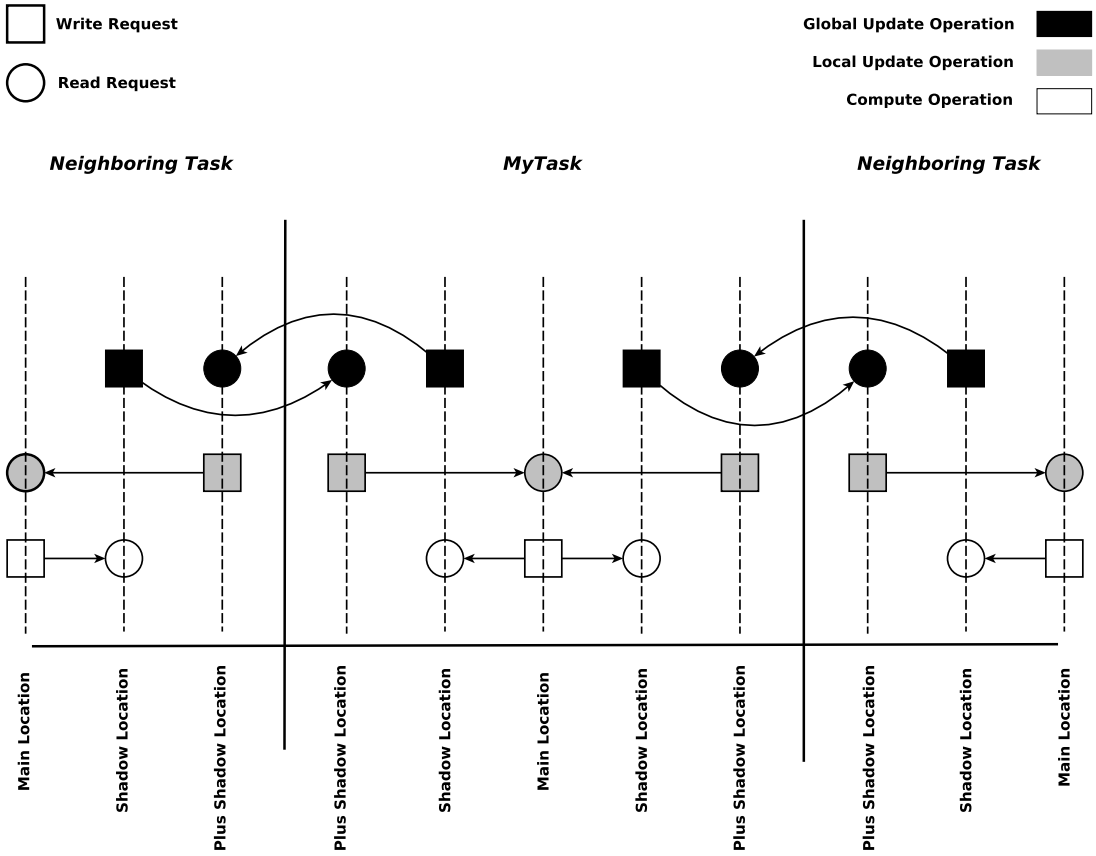


Figure 7.1 — CompUp Form Overlay: FIFO request ordering of the *Compute*, *Local Update* and *Global Update* operations

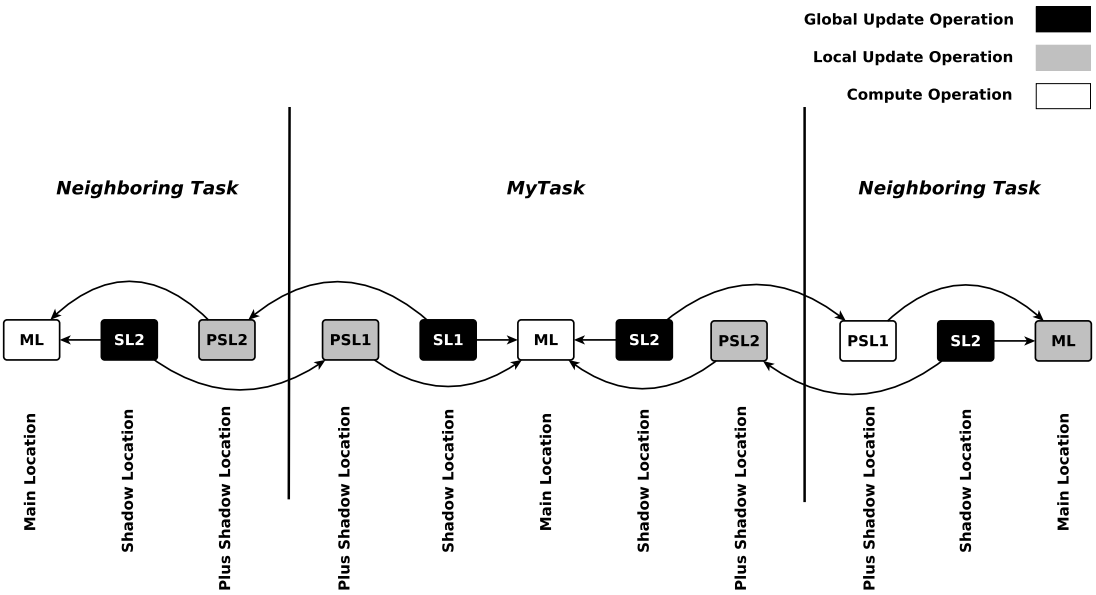
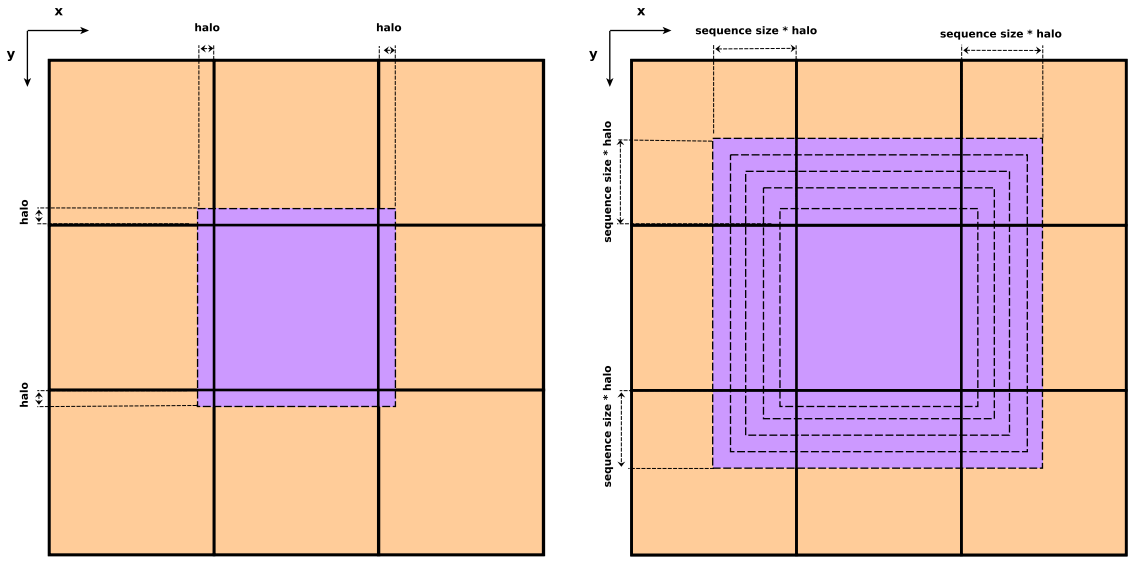


Figure 7.2 — CompUp Form Delay Digraph



(a) Overlapped Data Partitioning (b) Temporal Blocking Data Partitioning

Figure 7.3 — Dido main data partitioning schemes for a 2D stencil

the *halo* offset on each dimension as shown in Figure 7.3(a). The overlapped data partitioning greatly simplifies the *Compute* operation code and more precisely the computation of the frontier elements of each block. On one hand, it improves code locality and spares the analysis and specific treatment details that would have been, otherwise, necessary for their computation (*cf.* Listing 4.2). On the other hand, it helps avoid the complex indexing relative to the *halo* region updates. The computation part in the *compute* operation is thus reduced to one simple case that is applied on all the block elements. No special treatment of the edges is necessary. This considerably simplifies the code, making it clearer and easier to understand, to write, and thus to generate. Further details are given in Chapter 9. Preliminary experiments have shown that the overlapped data partitioning has no negative impact on performance. The overlapped data partitioning is the default data partitioning scheme applied by the Dido code generator.

Example

Figure 7.4 provides an example combining the *CompUp* form with the overlapped data partitioning for a 2D stencil with *von Neumann* neighborhood. Unlike the modeling presented in Section 4.4, each task consists of one *Compute* operation, 4 *Local Update* operations and 4 *Global Update* operations acting on 9 locations:

- The *main location* as shown in Figure 7.4 is extended and includes the halo region elements.
- 4 of the 8 *shadow locations* (NN, SS, EE, WW) present locations where the newly com-

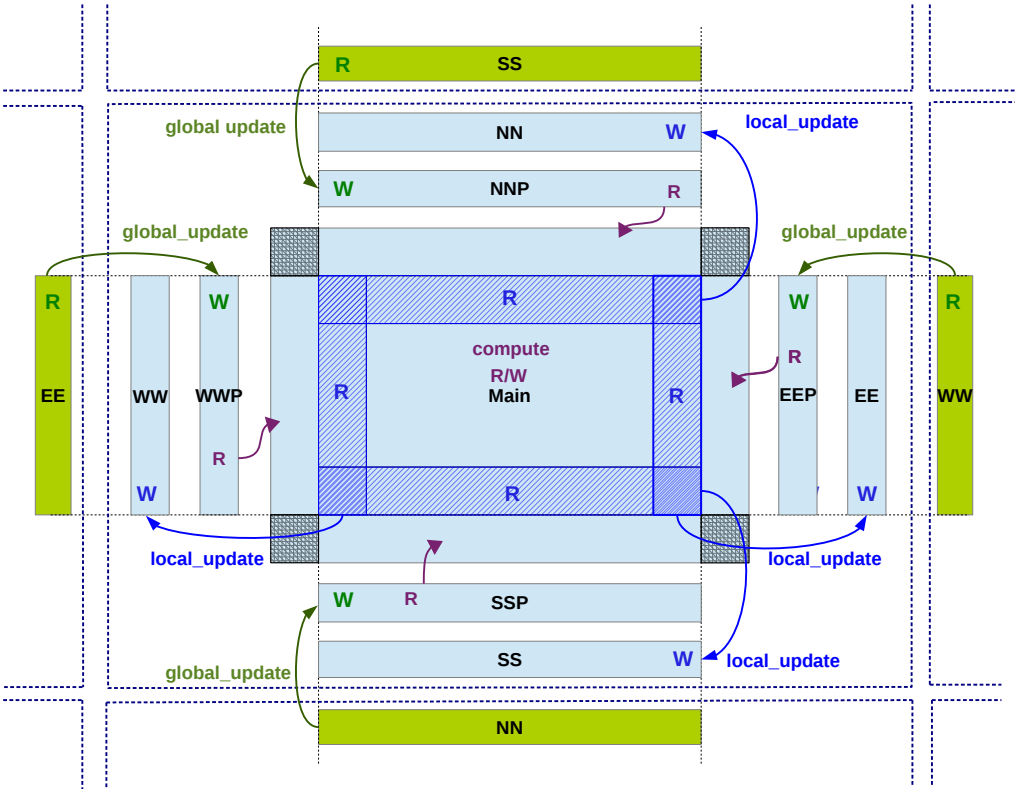


Figure 7.4 — Compute - Local Update - Global Update (CompUp) modeling combined with overlapped data partitioning for a 2D stencil example with von Neumann neighborhood

puted data of the local block edges is exported, saved and made available to neighboring tasks for reading.

- The remaining 4 *shadow locations* (NNP, SSP, EEP, WWP) are needed to store updated data that is imported from neighbors.

As shown in Figure 7.2, there are three types of operations:

- *Compute*: It has read/write access to the main location and read access to the *shadow* locations (NNP, SSP, EEP, WWP). At each iteration, it updates first the halo margin elements on the *main location*. Then, it executes the computation kernel.

- *Local Update*: copies updated data from *main location* and stores it in local buffers. The updated data is then made available for neighboring tasks and ready to be read without interrupting the main block computation. It has read access to the main location and write access to the corresponding *shadow location*.

- *Global Update*: reads the data from the *shadow locations* of the neighboring tasks and writes it into local buffers.

7.3 Temporal Data Locality

Stencil computations are memory-bound with a low arithmetic intensity. Namely, the quotient between the number of floating-point operations and the number of memory references is small. Reducing the amount and frequency of frontier data exchange between different compute nodes can have a noticeable impact.

Temporal blocking is one of the most widely used optimizations to reduce the communication overhead [DMV+08, CSN+10, MS11, WHZ+09]. It consists of partitioning the iteration space into sequences of a fixed size. Such a sequence is composed of local iterations that are computed while the data is kept on local memory. Communications are only performed at the end of each sequence. To this end, in each dimension each data block is enlarged by twice the halo multiplied by the sequence size as depicted in Figure 7.3(b). To compute the first local iteration of a sequence, all elements of the extended block are required. The results are stored locally. For the next iteration, no further communication is required since all elements are already present. After each local iteration, the loop domain shrinks by twice the halo.

Temporal blocking considerably improves the temporal data locality and reduces both communication and synchronization overhead. However, it introduces redundant computations and extra loading of data for the halo regions. Thus, there is a tradeoff between the additional computations and loads, and the gains in terms of bandwidth, waiting time and communication and synchronization overhead. In fact, the overhead of temporal blocking can be compensated only when the local share of the problem size is large enough. To conserve the semantics of the application and avoid redundant data copies, the size of an iteration sequence has to be odd.

Dido applies temporal blocking optimization while generating code for 2D and 3D stencils only if specified by the user as will be further explained in Chapter 9. Otherwise, the default data partitioning scheme is applied as specified in Section 7.2. For higher dimensions, preliminary performance tests have shown that, in most of the cases, the overhead of temporal blocking is too important and counteracts any potential gain in performance that we could achieve by improving temporal data locality. For 2D and 3D stencils, we opt for an iteration sequence size of 5 and 3, respectively. We have run tests for different stencil configurations and most of them are not deteriorated by using temporal blocking. On the contrary, performance gains are achieved and will be presented in Chapter 11, below.

Besides temporal locality and the resulting gains in efficiency and performance, temporal blocking offers good optimization opportunities that can be used by Pluto as will be explained in Section 9.1.

CHAPTER 8

Dido: Grammar & Architecture

Dido is an implicitly parallel domain-specific language for general multidimensional stencil computations that uses ORWL as a communication and runtime back-end. It offers a user-friendly interface that captures high-level stencil specifications and automatically generates ORWL parallel high-performance code for distributed-memory architectures.

In this chapter, we discuss the different design choices that we have made in order to meet the needs of real-world applications. Section 8.1 depicts the internal Dido architecture. In Section 8.2, we present the complete grammar of Dido syntax. Section 8.3 exhibits structural and execution parameters that need to be specified within Dido.

8.1 Architecture

Internally, Dido is made of four main components that are the lexer, the parser, the abstract syntax tree (AST) and the code generator. The user provides specifications written in the introduced language. These are parsed in order to extract stencil features that are transformed, in a second step, into an abstract syntax tree (AST). Given the internal representation of the stencil put in the form of an AST, Dido generates ORWL code following the *CompUp* form and the defined pattern presented in Chapter 7. The internal workflow of the framework is depicted in Figure 8.1. Dido is implemented using *OCaml* [LDG+08] lexer and parser generators (*ocamllex*, *ocamlyacc*).

8.2 Dido Grammar

Dido takes as input a file that contains a set of parameters specifying the stencil computation topology and size in a concise and trivial syntax. Listing 8.1 depicts the complete Extended Backus-Naur Form (EBNF) grammar of Dido stencil specification syntax.

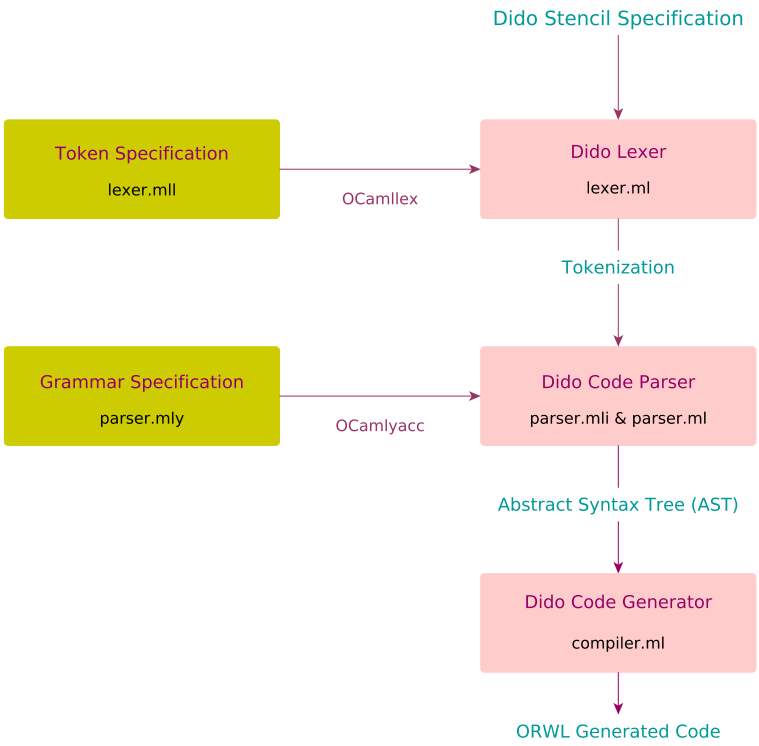


Figure 8.1 — A high-level overview of the Dido framework architecture.

8.3 Parameter Specification

Dido is intended to generate ORWL code following the pattern we have described in Chapter 7. Therefore, additional parameters have to be provided by the user, specifying the topology and size of their specific problem instance. In order to save compile time and efforts, the DSL is divided into two parts:

- The first part encompasses *structural parameters* of the application. Those parameters define the topology of the problem and enable the generation of the most complex parts of the ORWL code. This includes data locations, operations, handle initializations, initial positions in the FIFOs and critical sections. The compilation time of the ORWL code, generated by this part of the DSL, could be moderately long.

- The second part consists of the *execution parameters* for a particular problem instance. They can be provided within the DSL, if they are fixed, or as parameters at runtime.

The code resulting from the *structural* part of the DSL can be used for different problem instances. The user has the option to generate all the code from scratch or only generate

Listing 8.1 — Grammar of the Dido Language.

```
program             = app-name, structural-params, execution-params;
app-name            = "ORWL_Application = ", name, ";";
name                = string;

(* Structural Parameters *)
structural-params = main-data, auxiliary-data, application;
main-data           = "Main_Data = {",
                      md-name, "(", md-dim, "D) in (", dimlist, ");",
                      "}";
md-name             = string;
md-dim              = natural number;
dimlist             = dim, {",",dim};
dim                 = string;

auxiliary-data      = "Auxiliary_Data = {", auxlist, "}";
auxlist             = [aux, {",",aux}, ";"];
aux                 = string;

application         = "Application = {", kernel, types, halo,
                      iter-halo, boundary-conditions,
                      neighbourhood, grid-traversal, data-element-type,
                      "}";
kernel              = "kernel = ", kernel-fun, "in", kernel-file-name;
kernel-fun          = string;
kernel-file-name    = string, ";";
types               = "types = ", type-file-name, ";";
type-file-name      = string;
halo                = "halo = ", halo-value, ";";
halo-value          = natural number;
iter-halo           = "iteration_halo = ", iter-halo-value, ";";
iter-halo-value     = natural number;
boundary-conditions = "boundary_conditions = ",
                      ("periodic" | "non-periodic"), ";";
neighbourhood       = "neighb = ",
                      ("Moore" | "von-Neumann" | shapelist), ";";
shapelist           = shape, {",",shape};
shape               = "(", coord, {",",coord}, ")";
coord               = natural number;
grid-traversal      = "grid_traversal = ", ("Jacobi" | "Seidel"), ";";
data-element-type = "data_element_type = ", string, ";";

(* Execution Parameters *)
execution-params  =  "Execution_Parameters = {",
                     ["iterations_number = ", iter-nmb, ";",
                     md-name, sizelist, "into", blocksizelist, ";",
                     "number_nodes = ", node-nmb, ";",
                     "number_tasks_per_node = ", tpn-nmb, ";"],
                     "init_file =", init-file, ";"
                     "}";
sizelist            = "[", size, "]", {"[", size, "]"};
blocksizelist       = "[", blocksize, "]", {"[", blocksize, "]"};
iter-nmb            = natural number;    init-file    = string;
node-nmb            = natural number;    tpn-nmb      = natural number;
size                = natural number;    blocksize    = natural number;
```

the main function depending on the execution parameters specified in the *instantiation* part.

8.3.1 Structural Parameters

Listings 8.2, 8.3 and 8.4 show Dido specifications of 2D wave, 2D seidel and 3D molecular dynamics (MD) stencils, respectively.

- **Main & Auxiliary Data:**

The *main data*, that is the grid that undergoes the computation, is specified by its name (here `wave_grid`, `seidel_grid` or `box_grid`) and dimension (`3D` or `2D`). The user has also to provide names for the problem axes (`x`, `y` and `z`) (*see* Line 3 of Listings 8.2, 8.3 & 8.4). These will be used for the generation of, *e.g.*, the ORWL location and task identifiers. As mentioned in Chapter 4, we presume that there is only one *main data* grid, but there can be multiple coefficient grids, here called *auxiliary data*. These are specified by their names (*see* Line 6).

- **Computation Halos:**

To obtain solutions with the desirable accuracy, Dido supports high-order stencils in both time and space. Hence, the user has to specify the `halo` of the computation (*see* Line 11) as well as the discretization order in time `iteration_halo` (*see* Line 12) which corresponds to the number of previous iterations needed for the computation. The halo values are compile-time constants.

- **Kernel:**

Dido enables a natural description of the parallel computation, where the compute kernel itself is specified in the form of a sequential function. The user is therefore not exposed to any of the parallelism details as they just have to describe the sequential computation for one central stencil point. In order to broaden the spectrum of the supported applications, we have made the design choice to specify the sequential kernel function on the level of C code rather than within the DSL (*cf.* Listing 8.5). Hence, the kernel can be any arbitrary C code, which enables the user to combine different types of statements and easily reuse legacy code. The kernel can therefore be a simple call to an existing predefined function, or it can also contain hundreds of lines of code (*cf.* molecular dynamics (MD) with around 800 lines of code). A reference to the kernel function as well as the kernel header file should be given within the DSL (*see* Line 9).

The only constraints that a user has to respect when writing the kernel function is to follow a predefined order for the the kernel function arguments (*cf.* Listing 8.6) as these have to meet the function call at the level of the generated code (*See* Line 15 in Listing 9.10).

Listing 8.2 — ORWL 2D Wave stencil specification within Dido.

```
ORWL_Application = wave_2D;
Main_Data = {
  wave_grid (2D) in (x,y);
}
Auxiliary_Data = {
  velocity;
}
Application = {
  kernel = compute_wave_2D in "kernel-wave-2D.h";
  types = "types-wave-2D.h";
  halo = 1;
  iteration_halo = 2;
  boundary_conditions = non-periodic;
  neighb = von Neumann;
  grid_traversal = Jacobi;
  data_element_type = float;
}
Execution_Parameters = {
  init_file = "init-wave-2D.h";
  iterations_number = 100;
  wave_grid [10000][10000] into [1000][1000];
  number_nodes = 50;
  number_tasks_per_node = 2;
}
```

Listing 8.3 — ORWL 2D Seidel stencil specification within Dido.

```
ORWL_Application = seidel_2D;
Main_Data = {
  seidel_grid (2D) in (x,y);
}
Auxiliary_Data = {
  (* empty *)
}
Application = {
  kernel = compute_seidel_2D in "kernel-seidel-2D.h";
  types = "types-seidel-2D.h";
  halo = 1;
  iteration_halo = 1;
  boundary_conditions = non-periodic;
  neighb = (1,0),(0,1),(-1,0),(0,-1); (* or von Neumann *)
  grid_traversal = Seidel;
  data_element_type = float;
}
Execution_Parameters = {
  init_file = "init-seidel-2D.h";
  iterations_number = 100;
  seidel_grid [10000][10000] into [1000][1000];
  number_nodes = 25;
  number_tasks_per_node = 4;
}
```

Listing 8.4 — ORWL 3D MD application specification within Dido.

```
ORWL_Application = MD_3D;
Main_Data = {
  box_grid (3D) in (x, y, z);
}
Auxiliary_Data = {
 (* empty *)
}
Application = {
  kernel = compute_MD in "kernel-MD.h";
  types = "types-MD.h";
  halo = 1;
  iteration_halo = 1;
  boundary_conditions = periodic;
  neighb = Moore;
  grid_traversal = Jacobi;
  data_element_type = box;
}
Execution_Parameters = {
  init_file = "init-MD.h";
  iterations_number = 100;
  box_grid [1000][1000][1000] into [250][250][250];
  number_nodes = 16;
  number_tasks_per_node = 4;
}
```

Listing 8.5 — Molecular Dynamics Kernel

```
void compute_MD (size_t n, box box_grid_out[][n+2][n+2],
                           box box_grid_in[][n+2][n+2]) {
  compute_positions (box_grid_in[k][j][i]);
  periodicity (n, box_grid_out, box_grid_in);
  compute_partial_velocities (box_grid_in[k][j][i]);
  compute_forces(n, box_grid_out);
  compute_kinetic_energy (n, box_grid_out);
  compute_velocities (box_grid_out[k][j][i]);
  normalization_velocities (n, box_grid_out);
  compute_physical_quantities ();
}
```

The user has also to make sure that the specified kernel is coherent with the parameters given within Dido, *i.e.*, `halo` and `iteration halo`, so that the access does not fall outside the computation domain. Otherwise, a compilation error will be inevitably generated when compiling the generated code.

- **Neighbourhood:**

Dido does not extract the stencil neighborhood from the kernel since the latter is specified in a separate header file that is not parsed by the DSL. Therefore, the user has to

Listing 8.6 — Kernel function arguments order for a 2D wave kernel example

```
1  void compute_wave_2D (size_t n, float wave_out[][n+2],
2                                  float wave_in_1[][n+2],
3                                  float wave_in_2[][n+2],
4                                  float v[][n+2]) {
5    size_t y,x;
6    for (y = 1; y < n+1; y++) {
7      for (x = 1; x < n+1; x++) {
8        wave_out[y][x] = 2 * wave_in_1[y][x] - wave_in_2[y][x]
9                       + v[y][x] * (c0 * wave_in_1[y][x]
10                      + c1 * (wave_in_1[y][x-1] + wave_in_1[y][x+1]
11                            + wave_in_1[y-1][x] + wave_in_1[y+1][x]));
12     }
13   }
14 }
```

provide the neighborhood of his computation within Dido input code. This can be the classic *von Neumann* or *Moore* neighborhoods that they can specify directly (*see* Line 14). Alternatively, they can refine the neighborhood shapes by specifying the exact accessed neighboring points as in Listing 8.3, in order to avoid redundant communications.

- **Grid Traversal:**

In this work, we address both *Jacobi-like* and *Seidel-like* methods. The DSL enables the user to specify the iteration types, *i.e.*, *Jacobi* or *Seidel* (*see* Line 15). These have a considerable impact on the generated code as will be further explained in Chapter 9.

- **Data Types and Scalar Constants:**

Stencils supported by Dido are not constrained to classic basic numeric types, *i.e.*, *floats*, *doubles*, *etc.* The stencil grid elements can be of any data type whether it is a basic type or an aggregate data type. The user has a free hand to specify all the necessary types, data-structures and scalar constants that may be necessary for their application. These are defined on the level of C code in a separate header file. A reference to the type header file and the main data element type is included in the Dido code (*cf.* Line 10 & 16, respectively). This header file will be included in the generated ORWL code as shown in Figure 9.1. As an illustration, take the example of molecular dynamics (*see* Listing 8.7) where *particle* and *box* are both aggregate types (Systems of Arrays).

Listing 8.7 — Particle and Box type definitions in the Molecular Dynamics (MD) application as specified in the type header file `types-MD.h` included in *Dido* specification

```
1  typedef struct particle {
2    float X;//positions
3    float Y;
4    float Z;
5    float VX;//velocities
6    float VY;
7    float VZ;
8    float massp;
9    float ipab;
10 } particle;
11
12 typedef struct box {
13   float ppX[NPP];//positions
14   float ppY[NPP];
15   float ppZ[NPP];
16   float ppxD[NPP];//accelerations
17   float ppyD[NPP];
18   float ppzD[NPP];
19   float ppVX[NPP];//velocities
20   float ppVY[NPP];
21   float ppVZ[NPP];
22   float ppFX[NPP];//forces
23   float ppFY[NPP];
24   float ppFZ[NPP];
25   float ppmass[NPP];
26   float ppipab[NPP];
27   int nbpart;
28   particle var_x_n[];//vector of particles to displace in x direction
29   particle var_x_p[];
30   ...
31   int numb_x_n;//number of particles to displace in x direction
32   int numb_x_p;
33   ...
34 } box;
```

Listing 8.8 — Neumann boundary conditions.

```
1 float dido_boundary (size_t x, size_t y,
2                      size_t global_x, size_t global_y,
3                      size_t t) {
4   float border_value=0;
5   if ((x == global_x-1)||(y == global_y-1))
6     border_value=10;
7   return border_value;
8 }
```

Listing 8.9 — Dirichlet boundary conditions.

```
1 float dido_boundary (size_t x, size_t y,
2                      size_t global_x, size_t global_y,
3                      size_t t) {
4   return 10 + 0.2 * t;
5 }
```

- **External libraries:**

Dido enables the user to utilize any library or header file that they find necessary for their application. This can be easily done by including them in the types header file or in the kernel header file. For instance, in the Cell Nuclei Recognition application, the *CImg* image processing library [Tsc04] is used to load, save, display and process the image. It was sufficient to include this library in the code in order to take full advantage of its useful functions.

- **Boundary conditions:**

To comply with real application requirements and provide better simulation accuracy, Dido supports both *periodic* and *non-periodic* boundary conditions (*see* Line 13). In case of *non-periodic* boundary conditions, we allow the user to customize the boundary conditions according to their application needs. It can be one of the commonly used conditions such as *Neumann* (constant value for borders), *Dirichlet* (a specific function to compute element values on the edges) or any other border specification described through a C function called *dido_boundary* in the kernel header file. Listing 8.9 and Listing 8.8 show Dido specifications of a Dirichlet and a Neumann boundary condition, respectively.

8.3.2 Execution Parameters

The second part of Dido specifies the execution parameters for a particular problem instance. These consist of the initialization files, the convergence criteria and the problem size and repartition among available nodes.

- **Initialization:**

For a particular problem instance, the user has to provide a header file (*see* Line 19) that contains initialization functions for the *main* and *auxiliary data*. The user can use their own routines, just like they did for handwritten applications.

The following naming conventions have to be respected when specifying the init functions:

- `initialization` for the *main data*.
- `initialization_{auxiliary-data-name}` for *auxiliary data*.

- **Stopping Criteria:**

A stencil computation is often repeated either for prefixed number of iterations or until the computation stabilizes. Thus, the user has to provide a stopping criterion that can be either a maximum number of iterations within the DSL (*cf.* Line 20), or a convergence condition at the level of C code.

- **Problem Size and Repartition:**

The user has also to specify the global size of the problem as well as the data repartition into blocks. These can be specified either within the Dido input (*see* Line 21) or as parameters at runtime. Finally, they have to specify the number of nodes on which the computation is deployed as well as the number of tasks per node (*see* Line 22-23).

CHAPTER 9

Dido: Code Generation and Optimization

Given a stencil specification that does not exceed a few lines of trivial code, Dido generates hundreds of lines of ORWL parallel code. In addition to the defined pattern presented in 7, the quality of the generated code is crucial for performance. Careful attention has been dedicated to make the complex generated code as clear, readable and efficient as possible.

In this chapter, we exhibit the structure as well as the details of Dido generated code. Section 9.1 describes how Dido code is generated. Section 9.2 presents the structure of the generated code. In section 9.3, we describe the generated code components,*i.e.*, Locations, tasks, operations, *etc.* Section 9.4 shows the impact of the different stencil characteristics on the generated code. In section 9.5, we describe how Pluto can be used for data locality optimization of Dido generated code.

9.1 Code Generation

As shown in Listing 9.1, the code is generated through a simple call to the Dido executable (`./dido`) with the name of the file containing the stencil specifications as argument (*e.g.* `MD-3D.dido`). The user can also add additional options as following:

- `--temp`: to apply the temporal blocking optimization to the generated code as described in Section 7.3. It is recommended to use this option for 2D and 3D stencil computations with basic type data elements (*e.g.* `int`, `float`, `double`, *etc*). Without this option, the generated code follows the default pattern specified in Section 7.2.

- `--output`: to specify the output directory. The default would be the directory containing the Dido executable.

- `--mainonly`: to only generate the main function as specified in the *instantiation* part of Dido code, instead of regenerating all of the applicative code from scratch.

Listing 9.1 — Generating ORWL code from a Dido stencil specification.

```
./dido MD-3D.dido --temp --output=outdirectory --mainonly
```

9.2 Structure of the Generated Code

As shown in Figure 9.1, after running the Dido code generator, Dido generates five different files including the *Makefile*. The four other files contain applicative ORWL code as following:

- `{ApplicationName}-def.h` and `{ApplicationName}-def.c` comprise the declaration and definition of the ORWL tasks, locations and operation threads.
- `{ApplicationName}-main.c` file contains the main function of the ORWL program.
- `{ApplicationName}-tasks.c` file encompasses the communication code as well as the different functions and *CompUp* operations.

The generated code consists of C code with includes of both P99[1] and ORWL libraries (*cf.* Listing 9.2).

9.3 Generated Code Components

9.3.1 Locations and tasks

The number of ORWL data locations and neighboring tasks depends on the dimension n of the *main data* and the computation neighborhood. In the case of a *Moore* neighborhood, the number of neighboring tasks is:

$$|neigboring\ tasks| = \sum_{k=1}^{n} 2^k * C_n^{n-k}$$

Otherwise, the number of neighboring tasks is reduced. It corresponds to $2 * n$ in the case of *von Neumann* neighborhood. Following the *CompUp* form, the number of *shadow locations* is twice the number of tasks, in addition to the *main location*:

$$|locations| = 2 * |neigboring\ tasks| + 1$$

In order to guarantee a minimum of discoverability and readability for the complex generated code, the naming conventions for the ORWL tasks and locations have been carefully chosen. The attributed names encode the type of the object as prefix, *i.e.*, LOCATION or TASK, as well as the direction. For the direction, each axis name given by the user is followed by a suffix that encodes the direction in that particular dimension. There are three possible direction suffixes as following:

[1] `https://www.p99.gforge.inria.fr/`

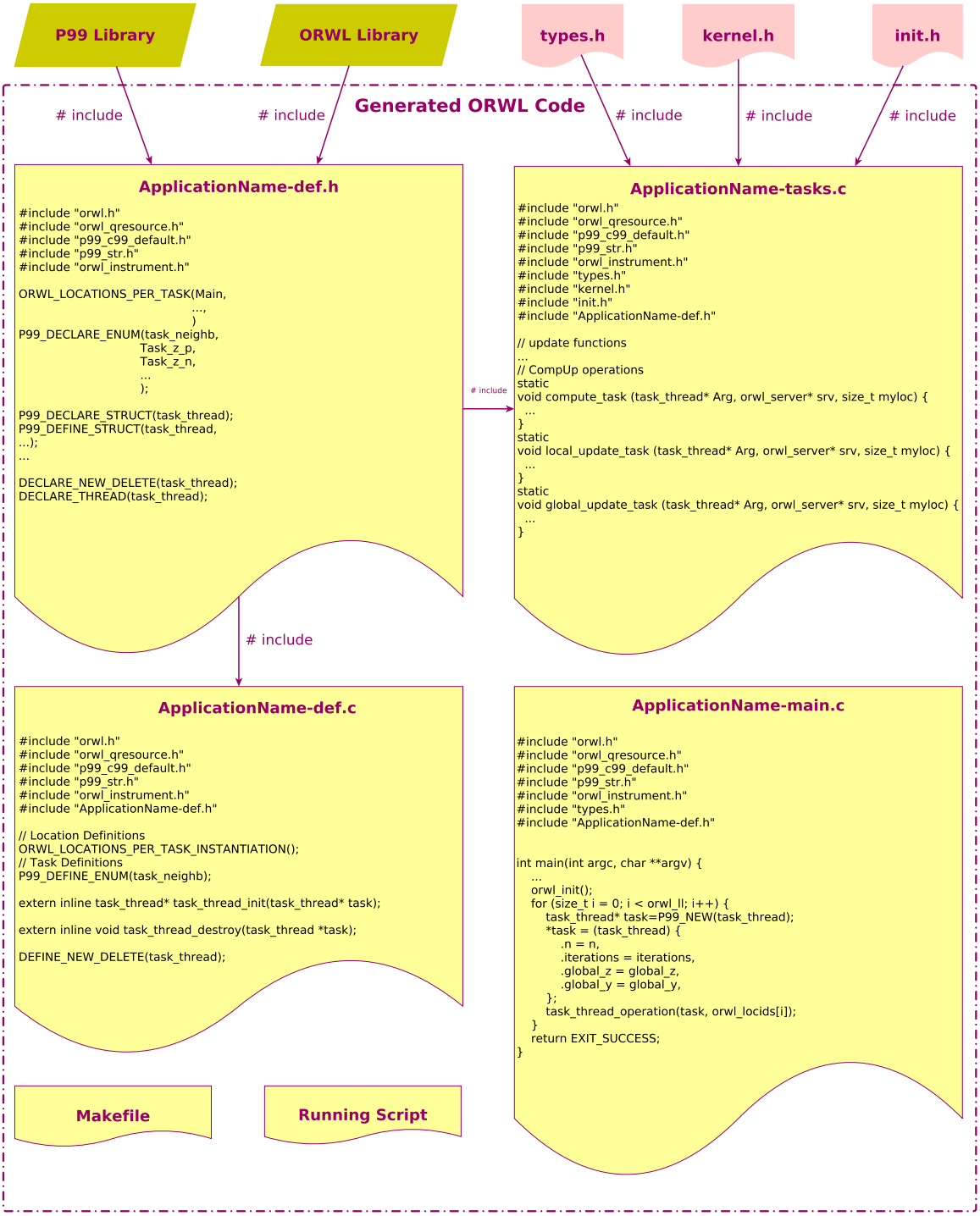


Figure 9.1 — Structure of Dido generated code

Listing 9.2 — Head of the generated code files with includes of P99 and ORWL libraries

```
#include "orwl.h"
#include "orwl_qresource.h"
#include "p99_c99_default.h"
#include "p99_str.h"
#include "orwl_instrument.h"
```

- *p* suffix corresponds to positive directions, *i.e.*, the directions in which the indices grow in the *main data* grid.
- *n* suffix corresponds to negative directions, *i.e.*, the directions in which the indices decrease in the *main data* grid.
- *k* suffix indicates that the size of the location in that direction is equal to the block size.

To sum up, the neighboring tasks and shadow location names follow the pattern:

$$\{TASK|LOCATION\}\_dim_1\{p|n|k\}...\_dim_n\{p|n|k\}\{\varnothing|\_PLUS\}$$

The $\_PLUS$ suffix indicates that the location serves for the *Global_Update* operation to locally save neighboring task updated data as described in Section 7.1.

For face *shadow locations*, *i.e.*, locations that have the same size as the *main data* grid in all dimensions except one dimension dim, the location and task names are reduced to:

$$\{TASK|LOCATION\}\_dim\{p|n\}\{\varnothing|\_PLUS\}$$

Figure 9.2 depicts the location names for a 2D stencil example following the naming conventions specified above. Despite the effort spent on the naming conventions, the code remains complex and can not be easily written by hand as further shown in Listing 9.3 and Listing 9.4.

9.3.2 An ORWL Program: a Loop over Locations

As an immediate result of the canonical form as evoked in Chapter 2, the main function of an ORWL program is nothing but a loop over all ORWL locations as shown in Listing 9.5. For each location, an operation is automatically generated. That operation is then instantiated at run time by a separate thread. The type and structure of the operation, whether it is a *Global Update*, *Local Update* or *Compute*, depends on the identifier of the associated location as depicted in Listing 9.6.

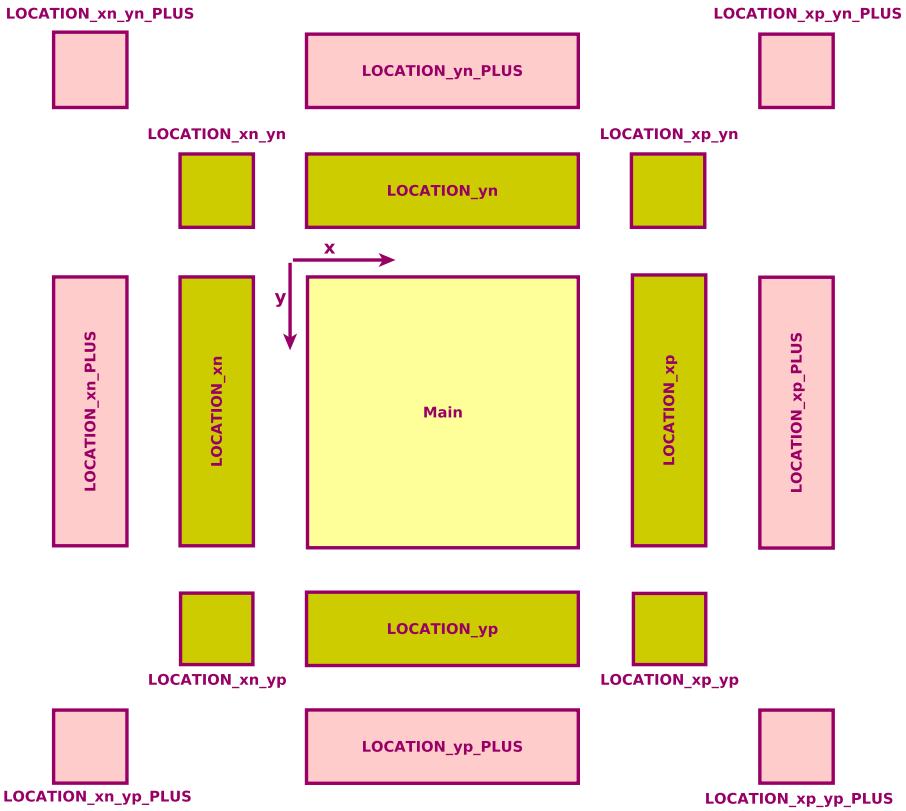


Figure 9.2 — Location names for a 2D Stencil with *Moore* neighborhood and (x, y) as axis names.

9.3.3 Operations

Excerpts from the *Compute*, *Local Update* and *Global Update* operations, corresponding to a 2D wave as specified in Listing 8.2, are depicted in Listing 9.7, Listing 9.8 and Listing 9.9, respectively. These show that all operations, apart from their type, have common features. Each operation consists of four main steps:

1. **Data Buffer Initialization:** Buffers corresponding to locations are initialized.

2. **Lock Initialization Step:** Lock objects and handles are initialized. Initial requests to the specified locations are inserted for each handle. The request orderings in the FIFOs are initially arranged with the positions indicated by the *CompUp* form as specified in Section 9.7.

3. **Initialization Iteration:** An introductory iteration, that distributes all control and data to the appropriate operations is executed.

4. **Computation Iterations:** The proper computations (or data copies) are run in an iteration loop.

Listing 9.3 — ORWL Task declarations for a 27-pt 3D Stencil with (x, y, z) as axis names

```
P99_DECLARE_ENUM(task_neighb,
                 TASK_zk_yp_xp, TASK_zk_yp_xn, TASK_zk_yn_xp,
                 TASK_zk_yn_xn, TASK_zp_yk_xp, TASK_zp_yk_xn,
                 TASK_zn_yk_xp, TASK_zn_yk_xn, TASK_zp_yp_xk,
                 TASK_zp_yn_xk, TASK_zn_yp_xk, TASK_zn_yn_xk,
                 TASK_zp_yp_xp, TASK_zp_yp_xn, TASK_zp_yn_xp,
                 TASK_zp_yn_xn, TASK_zn_yp_xp, TASK_zn_yp_xn,
                 TASK_zn_yn_xp, TASK_zn_yn_xn,
                 TASK_zp,       TASK_zn,       TASK_yp,
                 TASK_yn,       TASK_xp,       TASK_xn
                 );
```

Listing 9.4 — ORWL Location declarations for a 27-pt 3D Stencil with (x, y, z) as axis names.

```
ORWL_LOCATIONS_PER_TASK(MAIN_LOCATION,
              LOCATION_zk_yp_xp, LOCATION_zk_yp_xn, LOCATION_zk_yn_xp,
              LOCATION_zk_yn_xn, LOCATION_zp_yk_xp, LOCATION_zp_yk_xn,
              LOCATION_zn_yk_xp, LOCATION_zn_yk_xn, LOCATION_zp_yp_xk,
              LOCATION_zp_yn_xk, LOCATION_zn_yp_xk, LOCATION_zn_yn_xk,
              LOCATION_zp_yp_xp, LOCATION_zp_yp_xn, LOCATION_zp_yn_xp,
              LOCATION_zp_yn_xn, LOCATION_zn_yp_xp, LOCATION_zn_yp_xn,
              LOCATION_zn_yn_xp, LOCATION_zn_yn_xn,
              LOCATION_zp, LOCATION_zn, LOCATION_yp,
              LOCATION_yn, LOCATION_xp, LOCATION_xn,
              LOCATION_zk_yp_xp_PLUS, LOCATION_zk_yp_xn_PLUS,
              LOCATION_zk_yn_xp_PLUS, LOCATION_zk_yn_xn_PLUS,
              LOCATION_zp_yk_xp_PLUS, LOCATION_zp_yk_xn_PLUS,
              LOCATION_zn_yk_xp_PLUS, LOCATION_zn_yk_xn_PLUS,
              LOCATION_zp_yp_xk_PLUS, LOCATION_zp_yn_xk_PLUS,
              LOCATION_zn_yp_xk_PLUS, LOCATION_zn_yn_xk_PLUS,
              LOCATION_zp_yp_xp_PLUS, LOCATION_zp_yp_xn_PLUS,
              LOCATION_zp_yn_xp_PLUS, LOCATION_zp_yn_xn_PLUS,
              LOCATION_zn_yp_xp_PLUS, LOCATION_zn_yp_xn_PLUS,
              LOCATION_zn_yn_xp_PLUS, LOCATION_zn_yn_xn_PLUS,
              LOCATION_zp_PLUS, LOCATION_zn_PLUS, LOCATION_yp_PLUS,
              LOCATION_yn_PLUS, LOCATION_xp_PLUS, LOCATION_xn_PLUS
              );
```

These steps are synchronized by an ORWL barrier (*cf.* Line 28 in Listing 9.7) and a scheduling step (*cf.* Line 37 in Listing 9.7). The latter ensures that the initial FIFO positions of all handles of the different operations are inserted consistently at all locations. Thereafter, the execution order of tasks, based on their data dependencies, is fixed globally, and the operations can run concurrently without the need for any further global synchronization.

9.3.4 Update Functions

The generated code includes four update functions used to update *shadow region* buffers with newly computed data as following:

Listing 9.5 — Main function of ORWL genrated parallel code.

```
int main(int argc, char **argv) {
  ...
  orwl_init();
  for (size_t i = 0; i < orwl_ll; i++) {
    task_thread* task = P99_NEW(task_thread);
    *task = (task_thread) {
      .n = n,
      .iterations = iterations,
      .global_y = global_y,
    };
    task_thread_operation(task, orwl_locids[i]);
  }
  return EXIT_SUCCESS;
}
```

Listing 9.6 — Operation thread instantiation based on associated location ID.

```
DEFINE_THREAD(task_thread) {
  size_t myloc = orwl_myloc;
  orwl_server *const srv = orwl_server_get();
  orwl_locations task1 = ORWL_LOCAL(myloc);
  if (task1 == MAIN_LOCATION) {
      compute_task(Arg, srv, myloc);
  } else {
    if (task1 <= LOCATION_xn) {
      update_local_task (Arg, srv, myloc, task1);
    } else  {
      update_global_task (Arg, srv, myloc, task1);
    }
  }
}
```

- `update_edge_in_matrix` (*See* Listing 9.12) called by the *Compute* operation (*cf.* Line 10 in Listing 9.10).
- `update_edge_from_matrix` (*See* Listing 9.13) called by the *Local Update* operation (*cf.* Line 35 & Line 45 in Listing 9.8).
- `update_edge_from_edge` and `update_border` (*See* Listing 9.15 & Listing 9.14) called by the *Global Update* operation (*cf.* Line 37 & Line 41 in Listing 9.9).

As depicted in Listings 9.12, 9.13, 9.15 & 9.14, inside these update functions, the unit-stride loop index is incremented first, then the middle indices are incremented, and finally the least-contiguous index is incremented. In other terms, the inner loop corresponds to the unit-stride loop index and the outer loop corresponds to the least-contiguous index. To sum up, we stream contiguously through memory. Additionally, the iterator names are followed by suffixes that encode the dimension using the axis names provided by the user at the level of Dido code. This guarantees a minimum of discoverability for the update functions code.

Listing 9.7 — Structure of a *Compute* operation for a 2D wave stencil computation: second-order in time *Jacobi-like* iterations with *von Neumann* neighborhood

```
static
void compute_task (task_thread* Arg, orwl_server* srv, size_t myloc) {
  ORWL_THREAD_USE(task_thread, n, iterations, global_y);
  p99_seed *const seed = p99_seed_get();
  size_t mtask = ORWL_TASK(myloc); // main task id and coordinates
  size_t const global_x = ORWL_TASK(orwl_nl) / (global_y);
  size_t const x = mtask / (global_y);
  size_t const y = (mtask %  (global_y)) ;
  /*****              Data Buffer Initialization                *****/
  size_t const np2 = n+2;           // data buffer sizes
  typedef float fline [np2];        // data buffers
  typedef float mline [np2];
  mline * mtrx = P99_MALLOC (float[np2][np2]);
  mline * mtrx1 = P99_MALLOC (float[np2][np2]);
  mline * mtrx2 = P99_MALLOC (float[np2][np2]);
  float (*velocity)[][np2] = P99_MALLOC (float[np2][np2]);
  for (size_t i_y = 0; i_y < n; i_y++) {   // initialization
    for (size_t i_x = 0; i_x < n; i_x++) {
      (*velocity)[i_y + 1][i_x + 1] =
                    initialization_velocity (n*x + i_x, n*y + i_y);
    }
  }
  ...
  mline (* matrix [3]) = {mtrx, mtrx1, mtrx2};
  orwl_global_barrier_wait(myloc, 1, srv); // synchronization
  /*****               Lock Initialization Step                 *****/
  orwl_handle2 LWrite = ORWL_HANDLE2_INITIALIZER;
  orwl_handle2 buffer_y_n_Read = ORWL_HANDLE2_INITIALIZER;
  ...
  orwl_write_insert(&LWrite, myloc, 0, seed);
  orwl_read_insert(&buffer_y_n_Read,
        ORWL_LOCATION(ORWL_TASK(myloc), LOCATION_y_n_PLUS), 0, seed);
  ...
  orwl_schedule(myloc, 1, srv);
  /*****               Initialization Iteration                 *****/
  ORWL_SECTION(&buffer_y_n_Read, 1, seed) {
   //empty
  }
  ...
  ORWL_SECTION(&LWrite, 1, seed) {
   orwl_truncate(&LWrite, sizeof(void*));
   float (** mtrxP)[np2] = orwl_write_map(&LWrite);
   *mtrxP = matrix[0];
  }
  /*****                Computation Iterations                  *****/
  ... // see Listing 9.10
  // disconnect handles and free buffers
  orwl_disconnect(LWrite, 1, seed);
  orwl_disconnect(buffer_y_n_Read, 1, seed);
  ...
  free(mtrx);
  ...
}
```

Listing 9.8 — Structure of a *Local Update* operation for a 2D wave stencil computation: second-order in time *Jacobi-like* iterations with *von Neumann* neighborhood

```
static void local_update_task(task_thread *Arg, orwl_server* srv,
                              size_t myloc, orwl_locations task1) {
  ORWL_THREAD_USE(task_thread, n, iterations);
  p99_seed *const seed = p99_seed_get();

  /*****             Data Buffer Initialization                *****/
  typedef float const line [n+2];
  typedef float fline [n+2];
  size_t t_x;
  switch (task1) {
  case LOCATION_y_n:
    t_x = n+2;
    break;
  case LOCATION_x_n:
    t_x = 1;
    break;
    ...
  }
  orwl_global_barrier_wait(myloc, 1, srv);

  /*****              Lock Initialization Step                 *****/
  orwl_handle2 buffer_Write = ORWL_HANDLE2_INITIALIZER;
  orwl_handle2 main_Read = ORWL_HANDLE2_INITIALIZER;
  orwl_write_insert(&buffer_Write, myloc, 1, seed);
  orwl_read_insert(&main_Read,
           ORWL_LOCATION(ORWL_TASK(myloc), MAIN_LOCATION), 1, seed);
  orwl_schedule(myloc, 1, srv);

  /*****              Initialization Iteration                 *****/
  ORWL_SECTION(&buffer_Write, 1, seed) {
    orwl_truncate(&buffer_Write, sizeof(fline));
    ORWL_SECTION(&main_Read, 1, seed) {
      line *const *matrixP = orwl_read_map(&main_Read);
      fline *frontier = orwl_write_map(&buffer_Write);
      update_edge_from_matrix(task1, t_x, n, frontier, *matrixP);
    }
  }

  /*****               Computation Iterations                  *****/
  for (size_t iter = 0 ; iter <iterations; iter++){
    ORWL_SECTION(&buffer_Write, 1, seed){
      ORWL_SECTION(&main_Read, 1, seed){
        line *const *matrixP = orwl_read_map(&main_Read);
        fline *frontier = orwl_write_map(&buffer_Write);
        update_edge_from_matrix(task1, t_x, n, frontier, *matrixP);
      }
    }
  }
  orwl_disconnect(&main_Read, 1, seed);
  orwl_disconnect(&buffer_Write, 1, seed);
}
```

Listing 9.9 — Structure of a *Global Update* operation for a 2D wave stencil computation: second-order in time *Jacobi-like* iterations with *von Neumann* neighborhood and *non-periodic Neumann* boundary conditions

```
static void global_update_task(task_thread *Arg, orwl_server* srv,
                               size_t myloc, orwl_locations task1) {
  ORWL_THREAD_USE(task_thread, n, iterations, global_y);
  p99_seed *const seed = p99_seed_get();
  size_t mtask = ORWL_TASK(myloc); // main task id and coordinates
  ...
  bool border = false;

  /*****             Data Buffer Initialization                *****/
  ...
  orwl_global_barrier_wait(myloc, 1, srv);

  /*****              Lock Initialization Step                 *****/
  orwl_handle2 there_buffer_Read = ORWL_HANDLE2_INITIALIZER;
  orwl_handle2 here_buffer_Write = ORWL_HANDLE2_INITIALIZER;
  orwl_write_insert(&here_buffer_Write, myloc, 2, seed);
  switch (task1) {
  case LOCATION_y_p_PLUS :
    if (y + 1 < global_y) {
      orwl_read_insert(&there_buffer_Read,
            relative_task(x, y+1, LOCATION_y_n, global_y), 2, seed);
    } else {
      border = true;
    }
    break;
  ...
  }
  orwl_schedule(myloc, 1, srv);

  /*****               Initialization Iteration                 *****/
  ORWL_SECTION(&here_buffer_Write, 1, seed) {
    orwl_truncate(&here_buffer_Write, sizeof(fline));
    if (!border) {
      ORWL_SECTION(&there_buffer_Read, 1, seed) {
        line *neighb_frontier = orwl_read_map(&there_buffer_Read);
        fline *frontier = orwl_write_map(&here_buffer_Write);
        update_edge_from_edge(task1, n, frontier, neighb_frontier);
      }
    } else {
      fline *frontier = orwl_write_map(&here_buffer_Write);
      update_border(task1, t_x, n, x, y,
                    global_x, global_y, 0, frontier);
    }
  }

  /*****                Computation Iterations                 *****/
  ... // see Listing 9.11
  orwl_disconnect(&there_buffer_Read, 1, seed);
  orwl_disconnect(&here_buffer_Write, 1, seed);
}
```

Listing 9.10 — Computation iteraions inside a *Compute* operation for a 2D wave stencil computation: second-order in time *Jacobi-like* iterations with *von Neumann* neighborhood

```
for (size_t iter = 0 ; iter < iterations ; iter++) {
  size_t inbr = iter % 3;
  float  (*const wave_grid)[np2] = matrix[inbr];
  float  (*const wave_grid_iter_minus_1)[np2] = matrix[(inbr+1) % 3];
  float  (*const wave_grid_iter_minus_2)[np2] = matrix[(inbr+2) % 3];
  // Data Updates Followed by Computation
  ORWL_SECTION(buffer_y_n_Read, 1, seed){
    fline *frontier = orwl_read_map(&buffer_y_n_Read);
    t_x = n+2;
    update_edge_in_matrix(
      ORWL_LOCAL(ORWL_LOCATION(ORWL_TASK(myloc), LOCATION_y_n_PLUS)),
                t_x, n, frontier, wave_grid_iter_minus_1);
  }
  ...
  compute_wave_2D(n, wave_grid, wave_grid_iter_minus_1,
                  wave_grid_iter_minus_2, *velocity);
  ORWL_SECTION(&LWrite, 1, seed) {
    float (** matrixP)[n+2] = orwl_write_map(&LWrite);
    *matrixP = wave_grid;
  }
}
```

Listing 9.11 — Computation iteraions inside a *Global Update* operation for a 2D wave stencil computation: second-order in time *Jacobi-like* iterations with *von Neumann* neighborhood and *non-periodic Neumann* boundary conditions

```
if (!border) {
  for (size_t iter = 0 ; iter < iterations; iter++){
    ORWL_SECTION(&here_buffer_Write, 1, seed){
      ORWL_SECTION(&there_buffer_Read, 1, seed){
        line *neighb_frontier = orwl_read_map(&there_buffer_Read);
        fline *frontier = orwl_write_map(&here_buffer_Write);
        update_edge_from_edge(task1, n, frontier, neighb_frontier);
      }
    }
  }
} else {
  for (size_t iter = 0 ; iter < iterations; iter++){
    ORWL_SECTION(&here_buffer_Write, 1, seed){
      fline *frontier = orwl_write_map(&here_buffer_Write);
      update_border(task1, t_x, n, x, y,
                    global_x, global_y, iter, frontier);
    }
  }
}
```

Listing 9.12 — Update Edge in Matrix function

```
static
void update_edge_in_matrix(orwl_locations task, size_t t_x,
                           size_t const n, float frontier[][t_x],
                           float jacobi[][n+2]) {
  size_t const np2m1 = n + 1;
  assert(frontier);
  switch (task) {
  case LOCATION_y_n_PLUS :
    for (size_t i_y = 0; i_y < 1; i_y++) {
      for (size_t i_x = 0; i_x < n+2; i_x++) {
        jacobi[i_y][i_x] = frontier[i_y][i_x];
      }
    }
    break;
  case LOCATION_y_p_PLUS :
    for (size_t i_y = 0; i_y < 1; i_y++) {
      for (size_t i_x = 0; i_x < n+2; i_x++) {
        jacobi [i_y + np2m1][i_x] = frontier[i_y][i_x];
      }
    }
   break;
   ...
  }
}
```

Listing 9.13 — Update Edge from Matrix function

```
static
void update_edge_from_matrix(orwl_locations task, size_t t_x,
                             size_t const n, float frontier[][t_x],
                             float const jacobi[][n+2]) {
  size_t const np2m1 = n + 1;
  assert(frontier);
  switch (task) {
  case LOCATION_y_n :
    for (size_t i_y = 0; i_y < 1; i_y++) {
      for (size_t i_x = 0; i_x < n+2; i_x++) {
        frontier[i_y][i_x] = jacobi[i_y + 1][i_x];
      }
    }
    break;
  case LOCATION_y_p :
    for (size_t i_y = 0; i_y < 1; i_y++) {
      for (size_t i_x = 0; i_x < n+2; i_x++) {
        frontier[i_y][i_x] = jacobi[np2m1 - 1 + i_y][i_x];
      }
    }
    break;
    ...
  }
}
```

Listing 9.14 — Update Border Function

```
static
void update_border(orwl_locations task, size_t t_x, size_t const n,
                   size_t x, size_t y,
                   size_t global_x, size_t global_y,
                   size_t t, float frontier[][t_x]) {
  assert(frontier);
  switch (task) {
  case LOCATION_y_n_PLUS :
    for (size_t i_y = 0; i_y < 1; i_y++) {
      for (size_t i_x = 0; i_x < n+2; i_x++) {
        frontier[i_y][i_x] = dido_boundary (x*n+i_x, y*n+i_y,
                                            global_x, global_y, t);
      }
    }
    break;
  ...
  case LOCATION_x_n_PLUS :
  for (size_t i_y = 0; i_y < n+2; i_y++) {
    for (size_t i_x = 0; i_x < 1; i_x++) {
      frontier[i_y][i_x] = dido_boundary (x*n+i_x, y*n+i_y,
                                          global_x, global_y, t);
    }
  }
  break;
  ...
  }
}
```

Listing 9.15 — Update Edge from Edge function

```
static
void update_edge_from_edge(orwl_locations task, size_t const n,
            float here_frontier[][n+2], float const frontier[][n+2]) {
  assert(frontier);
  for (size_t i_y = 0; i_y < 1; i_y++) {
    for (size_t i_x = 0; i_x < n+2; i_x++) {
      here_frontier[i_y][i_x] = frontier[i_y][i_x];
    }
  }
}
```

9.4 Generated Code Variability

Dido supports a wide range of stencil computations with different characteristics such as grid traversal and boundary conditions. This section gives an overview of the impact of those characteristics on the generated code.

Listing 9.16 — Waiting iterations inside operations for *Seidel-like* grid iterations

```
/*****                    Waiting iterations                        *****/
for (size_t wi = 0; wi<x+y; wi ++) {
  ORWL_SECTION(&buffer_y_n_Read, 1, seed) {
    //empty
  }
  ORWL_SECTION(&buffer_y_p_Read, 1, seed) {
    //empty
  }
  ORWL_SECTION(&buffer_x_n_Read, 1, seed) {
    //empty
  }
  ORWL_SECTION(&buffer_x_p_Read, 1, seed) {
    //empty
  }
  ORWL_SECTION(&LWrite, 1, seed) {
    //empty
  }
}
```

9.4.1 Grid Traversal

Dido handles both *Jacobi*-like and *Gauss-Seidel*-like grid traversals. Listing 9.7 depicts the compute operation code corresponding to *Jacobi*-like iterations. Instead of two copies of the *main data* grid swapping their roles after each iteration, only one single copy of the *main data* grid is required by *Gauss-Seidel*-like iterations. Additionally, the inherent dependency imposes an order to respect when computing elements. On the level of the generated code, this is handled by adding waiting iterations (*cf.* Listing 9.16) just after the *initialization iterations* and before *computation iterations* creating a wavefront.

9.4.2 Boundary Conditions

As specified in Section 8, Dido supports both *periodic* and *non-periodic* boundary conditions. These have an impact on the generated code, and more particularly, on the *Global Update* operation code. The other types of operations, *i.e. Compute* and *Local Update*, are not affected by the boundary condition treatment. Listing 9.17 and Listing 9.9 show the *Global Update* operations corresponding to a 2D *Jacobi-like* stencil with *periodic* and *non-periodic* boundary conditions, respectively.

As evoked in Subsection 4.3, boundary conditions can easily become a bottleneck in the stencil code performance, especially if a test is made at every point to determine if the stencil point falls on the boundary. Therefore, we devoted significant attention in order to encapsulate all necessary treatment in the initialization phase so that the computation iterations remain test-free. In case of *non-periodic* boundary conditions, a control is made in the *lock initialization* step inside the *Global Update* operation to check whether the corresponding block edge falls on the boundary (*cf.* Line 17-27 in Listing 9.9) If it is

the case, the `update_boder` function is called in the *initialization iteration*. It is this update function that calls the `dido_boundary` function provided by the user (*cf.* Line 11 in Listing 9.14). Following the same scheme, one treatment or the other is performed inside the *computation iterations* as shown in Listing 9.11. Subsequently, the *computation iterations* are test-free.

As for *periodic* boundary conditions, the treatment is completely confined in the *lock initialization* step. The lock handles are initialized appropriately in a way to form a torus as shown in Listing 9.17. No specific treatment has to be made at the level of the computation iterations. The computation iterations are consequently test-free (*See* Line 40-48 in Listing 9.17).

9.4.3 Temporal Blocking

As mentioned in Chapter 7, the overlapped data partitioning is the default data partitioning scheme for Dido generated code. Each block is then enlarged by twice the *halo* offset on each dimension as shown in Listing 9.7. However, if the user chooses to apply the temporal blocking optimization to the generated code as described in Section 7.3, each data block is enlarged by twice the halo multiplied by the sequence size on each dimension.

9.5 Pluto for Data Locality Optimization

Stencil computations can be written as simple nested loops with linear affine bounds and accesses. However, naive loop implementations often suffer from poor cache locality. To improve locality and intra-node data reuse, we combined our code generation technique with the polyhedral loop optimizer *Pluto*. We have chosen *Pluto* over other code generation and optimization frameworks and Stencil DSLs as it is a source-to-source code transformer and can therefore be directly applied. However, we had to decide how to interleave *Pluto* generated code with *ORWL*, and came up with two different solutions:

The first consists of using Pluto to optimize data locality on each core, *i.e*, at the task level. Here, each compute node is associated with one *ORWL* process which is composed of several tasks. These tasks operate on different data blocks in parallel. We use *Pluto* to optimize the kernel served by these tasks. So, here *Pluto* is not used as a parallelizer, but rather as a data locality optimizer: the parallelism is only *ORWL*-based and the generated code is exclusively *ORWL* code. In particular, this solution does not use *OpenMP*.

The second solution is to use code that is parallelized by *Pluto* at the node level. Here, we place one ORWL process consisting of one task/block per node. For this solution, we rely on *Pluto* for data locality but also for coarse-grained parallelism. This provides a hybrid solution where *ORWL* and *OpenMP* are both used to guarantee parallelism on different levels.

Pluto does not alter in any way the accuracy of Dido generated code. Per default, its data dependency detection guarantees that operations are performed in exactly the same

Listing 9.17 — *Global Update* operation of a 2D Jacobi stencil with *periodic* boundary conditions

```
static void update_global_task(task_thread *Arg, orwl_server* srv,
                               size_t myloc, orwl_locations task1) {
  ...
  /*****               Lock Initialization Step                 *****/
  orwl_handle2 there_buffer_Read = ORWL_HANDLE2_INITIALIZER;
  orwl_handle2 here_buffer_Write = ORWL_HANDLE2_INITIALIZER;
  orwl_write_insert(&here_buffer_Write, myloc, 2, seed);
  switch (task1) {
  case LOCATION_y_p_PLUS :
    if (y + 1 < global_y) {   // x and y are the block coordinates
      orwl_read_insert(&there_buffer_Read,
        relative_task(x, y+1, LOCATION_y_n, global_y), 2, seed);
    } else {
      orwl_read_insert(&there_buffer_Read,
        relative_task(x, 0, LOCATION_y_n, global_y), 2, seed);
    }
    break;
  case LOCATION_y_n_PLUS :
    if (y > 0) {
      orwl_read_insert(&there_buffer_Read,
       relative_task(x, y-1, LOCATION_y_p, global_y), 2, seed);
    } else {
      orwl_read_insert(&there_buffer_Read,
       relative_task(x, global_y-1, LOCATION_y_p, global_y), 2, seed);
    }
    break;
    ...
  }
  orwl_schedule(myloc, 1, srv);
  /*****               Initialization Iteration                 *****/
  ORWL_SECTION(&here_buffer_Write, 1, seed) {
    orwl_truncate(&here_buffer_Write, sizeof(fline));
    ORWL_SECTION(&there_buffer_Read, 1, seed) {
      line *neighb_frontier = orwl_read_map(&there_buffer_Read);
      fline *frontier = orwl_write_map(&here_buffer_Write);
      update_edge_in_edge(task1, n, frontier, neighb_frontier);
    }
  }
  /*****                Computation Iterations                  *****/
  for (size_t iter = 0 ; iter < iterations; iter++){
    ORWL_SECTION(&here_buffer_Write, 1, seed){
      ORWL_SECTION(&there_buffer_Read, 1, seed){
        line *neighb_frontier = orwl_read_map(&there_buffer_Read);
        fline *frontier = orwl_write_map(&here_buffer_Write);
        update_edge_in_edge(task1, n, frontier, neighb_frontier);
      }
    }
  }
  orwl_disconnect(&there_buffer_Read, 1, seed);
  orwl_disconnect(&here_buffer_Write, 1, seed);
}
```

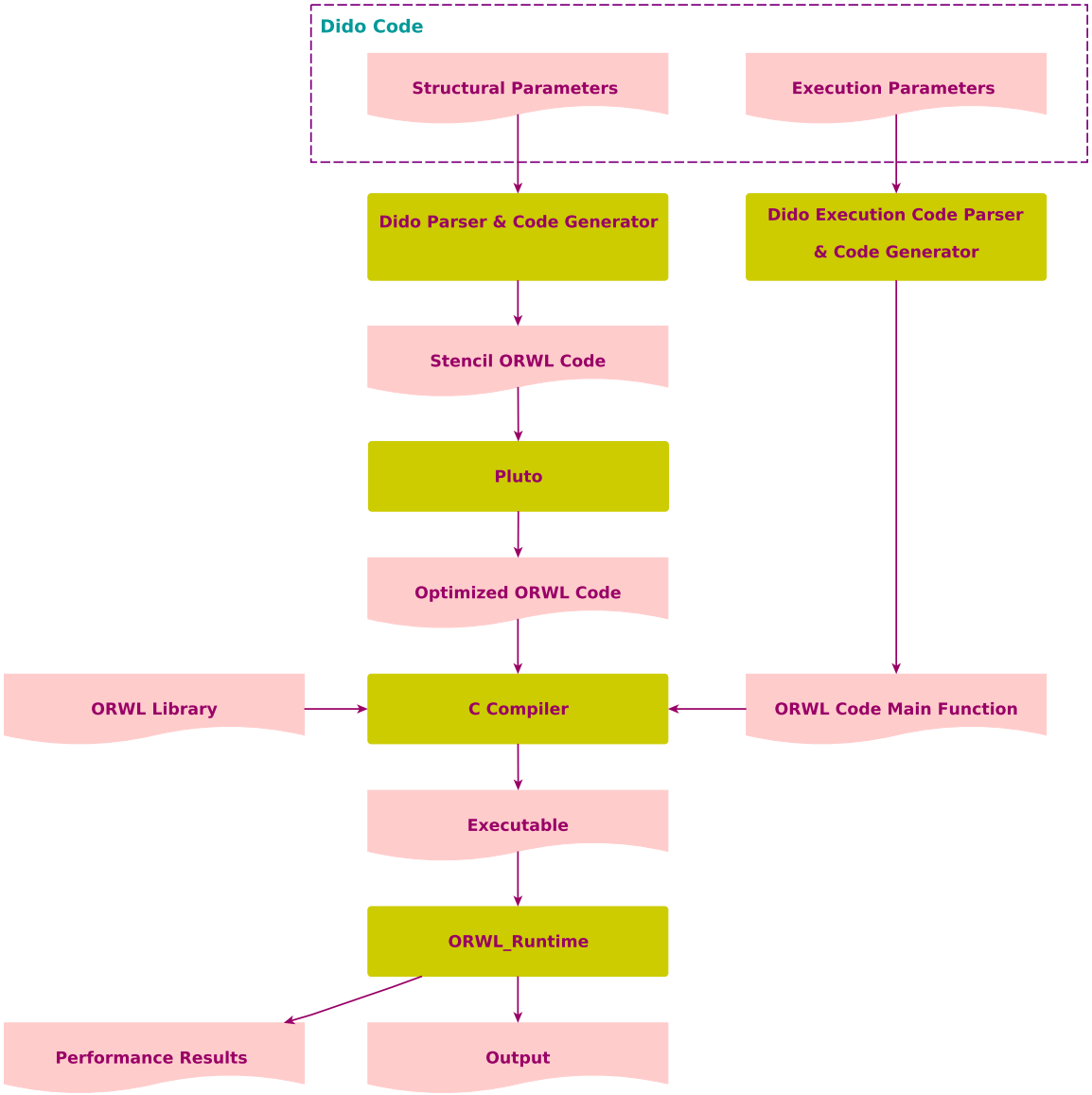


Figure 9.3 — Dido's Compilation Flow

order as specified in the original program. There are specific flags, though, that allow to suppose commutativity or associativity of certain operations that then slightly change the program semantic. In all experiments that are reported in Chapter 11, we did not use such intrusive flags. Figure 9.3 depicts the full compilation flow of Dido, including Pluto as a data locality optimizer.

CHAPTER 10

Dido Features

Dido is an implicitly parallel domain-specific language that has a wide variability and supports a large range of multi-dimensional stencil computations. It meets the specific needs of the application at a high level of abstraction and achieves both productivity and performance benefits. Additionally, Dido offers interesting features for the supported multi-dimensional stencil-based applications. First, it leverages all of the ORWL features evoked in Section 2.6. It also adds on new capabilities such as guaranteeing the correctness, readability and discoverability of the generated code. The combination of those features makes of Dido a powerful code generation framework for stencil computations in general, and for distributed-memory architectures in particular.

This Chapter provides an overview of the different features of the suggested framework. Section 10.1 highlights the different features that Dido guarantees for the generated code. Section 10.2 outlines the variability of Dido and its capacity to handle a wide range of stencil computations. In Section 10.3, we exhibit the programmer productivity benefits achieved by Dido. Section 10.4 underlines the efficiency of the generated code .

10.1 Generated Code Features

Given a stencil specification captured in a concise and trivial syntax, Dido generates hundreds of lines of C code. Details about the generated code are provided in Chapter 9. In this section, we outline the different features of Dido generated code.

10.1.1 Correctness

Dido ensures the correctness of the generated code. The latter is guaranteed to be both error-free and deadlock-free. Prior to code generation, the framework verifies that all the parameters specified within the DSL are coherent. If not, an error message is displayed. The error message indicates the error line. For example, in Listing 8.3 the problem is divided into $10 \times 10 = 100$ blocks, which corresponds to the total number of tasks, $25 \times 4 = 100$. In this case, no error message is displayed and the code is automatically generated. If the error is syntactic, only the error line number is displayed. If the error is related to the computation repartition, the following error message is displayed:

```
./dido seidel-2D.dido
Error: Please verify the computation repartition among nodes.
```

Additionally, Dido generated code is guaranteed to be equivalent to the sequential code. We speak here of code accuracy. In fact, accuracy is one of the most interesting features that Dido inherits from ORWL. As evoked in Section 2.6, the access order of ORWL tasks to the data is completely determined by the static order given in the initialization phase of the FIFOs. Thereby, we can guarantee that each individual data element is computed in exactly the same order with exactly the same low-level operations as it would be in a conventional sequential setting. As a result, the generated parallel code preserves the semantics of the original sequential kernel and gives provably correct results. If the C pragma `FP_CONTRACT` is set to `OFF` such that the compiler is not allowed to reorder instructions, not even a numeric derivation of the results is to be expected.

Pluto does not alter in any way the accuracy of Dido generated code. Per default, its data dependency detection guarantees that operations are performed in exactly the same order as specified in the original program. There are specific flags, though, that allow to suppose commutativity or associativity of certain operations that then slightly change the program semantic. In all experiments that are reported in Chapter 11, we did not use such intrusive flags.

10.1.2 Readability & Discoverability

Despite its complexity, the generated code is relatively clear and readable. This is first due to the defined pattern described in Chapter 7. In fact, the overlapped data partitioning simplifies considerably the code and more precisely the computation of the frontier elements of each block. It improves code locality and spares the analysis and specific treatment details that would have been, otherwise, necessary for their computation. It also helps avoiding the complex indexing relative to the *halo* region updates. The computation part in the *compute* operation is thus reduced to one simple case that is applied on all the block elements. This considerably simplifies the code, making it clearer and easier to understand.

Additionally, naming conventions are carefully chosen not only for tasks and locations, but also functions, iterators and data buffers (*See* Chapter 9). Besides, the generated code is well-indented. For all of these reasons, the generated code is relatively clear and readable.

The readability of the generated code enables the user to easily take it in hand. They can therefore easily modify it in order to optimize it or tune it for specific usage or architecture, or even include it as a part of a larger application that comprises a stencil computation. Larger applications that include stencils in addition to other parts can therefore benefit from Dido to parallelize the stencil part.

10.1.3 Modularity

Dido is oriented towards real-world applications. In real use, users would not be able or even willing to recompile their code for each problem instance as this is moderately time-consuming and involves many technical details. On the other hand, it would be impossible for developers to provide binaries for every possible configuration.

As evoked in Section 8.3, Dido is composed of two parts, *i.e.*, *structural parameters* and *execution parameters*. The *structural parameters* part results in the generation of all the computation and communication code. The resulting code is parametric in the number of processes, tasks and problem sizes. Once compiled, it can be used for different problem instances specified in the second part involving the *execution parameters*. These can be given either within the DSL, if they are fixed, or as parameters at runtime. This solution saves compile time and efforts and significantly enhances the portability of the generated code.

10.1.4 Optimal Communications

Dido enables the user to coarsely choose one of the standard neighborhood types, *i.e.*, *Moore* or *von Neumann*, or alternatively to specify the exact access pattern if it does not correspond exactly to one of these. This option helps eliminating redundant or extra communications. Consequently, it alleviates the contention over the communication bus and provides better performance.

10.2 Variability

Dido meets the needs of real applications at a high level of abstraction by supporting a wide range of stencil computations. It gives the user a free hand to specify, *e.g.*, data aggregate types and customize boundary conditions, among other features. Table 10.1 depicts the variability of Dido and shows its great capacity to support stencil computations with various properties.

10.3 Programmer Productivity

Dido was designed to improve the programmer productivity by sparing them from writing the complex parts of ORWL parallel code. Given a stencil specification that does not exceed a few lines of straightforward code, bare of any details needed for parallelization, hundreds of lines of code are generated. The generated code is guaranteed to be correct and error-free. Table 10.2 depicts the number of hand-written lines compared to the number of generated code lines for the considered benchmarks. The number of lines to be written by the user, in the case of a 3D 27-pt Jacobi for example, is reduced by 96%. We estimate this to be a considerable improvement in terms of programmer productivity.

Table 10.1 — Variability of Dido

Variability	Level	Options
Boundary Conditions	structural	Periodic, Non-periodic (Dirichlet, Neumann ...)
Computational Domain	structural	multi-dimensional (2D, 3D, 4D, ...)
Kernel	structural	arbitrary C code
Halo	structural	various
Iteration_Halo	structural	various
Shape/Neighborhood	structural	*Moore, von Neumann* or any other ...
Data types	structural	basic types or aggregate types
Problem Size	instantiation	various
Block Size	instantiation	various
Number of Nodes	instantiation	various
Temporal Blocking	optimization	with or without
Pluto Optimization	optimization	with or without
Parallelization	optimization	ORWL and OpenMP or ORWL only

Table 10.2 — Number of hand-written lines of code vs. number of generated lines of code

metric	Livermore	3D Wave	3D 27-pt Jacobi
hand-written lines	68	63	77
generated lines	711	988	2081

This measurement of a gain in productivity uses a comparison to explicit hand coding with ORWL. We think that the result would not be much different if we would compare to coding with other paradigms for parallel computation, as long as these involve explicit creation of buffer space, data communications or access locks.

Not only does the DSL reduce the number of hand-coded lines, but it also avoids to manually write complex and error-prone parts such as the halo region update operations. A lot of time and effort is saved by automatically generating the indexing expressions and communication statements. Table 10.3 lists the number of complex programming details such as communication statements and indexing operations that are automatically generated by the DSL. For example, if the user had to write the ORWL parallel code of a 3D 27-point stencil without the aid of the DSL, they would have to handle 53 data loca-

Table 10.3 — Parallel programming details automatically generated and spared for the user

metric	Livermore	3D Wave	3D 27-pt
neighboring tasks	4	6	26
locations	9	13	53
handle initializations	9	13	31
communication statements	29	48	110
indexing expressions	124	167	674

tions. To do so, they would have to initialize 31 handles and write 110 communication statements and 674 indexing operations.

Seeing all these properties, the total development time of a parallel stencil computation within our framework can be just a few minutes. In addition, it took an PhD candidate one year to write the ORWL code of the Cell Nuclei Recognition application. With the help of Dido, we implemented an equivalent code in only two weeks.

For all those reasons, it seems clear that Dido achieves a considerable improvement in terms of programmer productivity.

10.4 Efficiency

Dido provides a big leap in productivity without sacrificing the performance. Thanks to the incorporated domain-specific knowledge that went into the defined patterns, it leverages all the features of the ORWL model including liveness, efficiency and equity among tasks. Additionally, Dido generated code is scalable and achieves competitive performance compared to hand-crafted ORWL code. The achieved gain in performance is even enhanced through the use of the temporal blocking optimization and the loop optimizer Pluto. More details are given in Chapter 11 where a complete performance evaluation of Dido generated code is presented.

CHAPTER 11

Dido Performance Evaluation

Dido is an implicitly parallel domain-specific language that automatically generates ORWL code for multi-dimensional stencil computations following the pattern presented in Chapter 7. Details about the generated code are given in Chapter 9. In this Chapter, we measure the efficiency of our approach experimentally. In order to validate our modeling, we investigate the overhead, if any, that it might add to the computation. We also explore the efficiency of the applied optimizations, *i.e.*, temporal blocking and Pluto for intra-node data locality. Additionally, we compare the performance of the generated code to both handwritten ORWL code and MPI implementations. We also explore the performance of Dido generated code of the molecular dynamics real-world application. In section 11.1, we describe the testbed architectures. The experimental setups and the performance results are presented in section 11.2. We conclude in section 11.3.

11.1 Testbed Architectures

The experiments as presented here have been conducted on the *Grisou* and *Graphene* clusters of the *Grid'5000* experimental testbed[1], using a total of over 10^6 CPU hours counting physical cores.

- ***Grisou:*** It consists of 51 nodes with two Intel Xeon E5-2630 processors, each, of which 48 were available to us. Each of the processors has 8 physical cores at 2.4 Ghz, so the total number of physical cores available for us was 768. Each physical core is doubled by two hyperthreaded cores, a total of 1536. Each node is equipped with 126 GiB of memory, 6.3 TiB in total. The nodes are connected through a 10 Gb/s network.
- ***Graphene:*** It consists of 144 nodes with one Intel Xeon X3440 per node, of which 100 were available to us. Each of the processors has 4 physical single-threaded cores at 2.53 GHz, so the total number of physical cores available for us was 400. Each node is equipped with 16 GiB of memory. The nodes are connected through an Infiniband-20G network.

[1] `https://www.grid5000.fr/`

The relevant software on this platform is a GNU/Linux system (kernel 3.16), a `gcc` compiler (version 4.8.2) and the *pet* branch of Pluto. All the following results are obtained after running 100 iterations. For greater accuracy, we take an average over four runs.

11.2 Experimental Results

11.2.1 Setup 1

In [GJ11], a set of experiments is conducted to evaluate the performance of the ORWL parallel model, by studying the computation efficiency depending on the blocks' number and size. For that purpose, a handwritten implementation of the Livermore Kernel 23 was considered. As a first step in our performance study, we repeat exactly the same setup used in [GJ11], but with Dido Livermore Kernel 23 generated code instead of the handwritten version. Our aim here is to verify that our approach did not add any overhead to the computation and conserved the ORWL efficiency properties. Similarly to [GJ11], we consider problems of 4, 16, 36, 64 and 100 blocks, where one compute node is reserved for each block. For each configuration, we increase the global problem size, until we reach the maximum size per block that fits into the RAM of the target machines. This set of experiments is conducted on the *Graphene* cluster. Results are shown in Figure 11.1 and Figure 11.2. Figure 11.1 shows the variation of the average execution time per data element depending on the blocks' number and size. Figure 11.2 relates the time spent on computation to the time spent on waiting for some frontier data.

- **Comparison with Handcrafted Code:**

As shown in Figure 11.1, the computation time per data element decreases when the problem size increases. It tends to a lower limit. The times for different block partitions are almost indistinguishable, especially for large block sizes. This proves that the overhead for subdividing into more blocks is negligible. Figure 11.2 shows that for small problems, almost all the time is spent waiting. However, for larger problems, the time spent on computation increases and finally reaches about 99% of the total execution time. These results are similar to the results reported in [GJ11], which proves that Dido does not in nay way degrade the performance of the ORWL model. On the contrary, it conserves the ORWL efficiency properties.

11.2.2 Setup 2

In this set of experiments, we study the scalability of Dido generated code and the impact of the applied optimizations on performance. For each of the considered benchmarks, we have launched tests for the five following configurations:

1. Dido generated ORWL code with standard compiler optimizations, only.

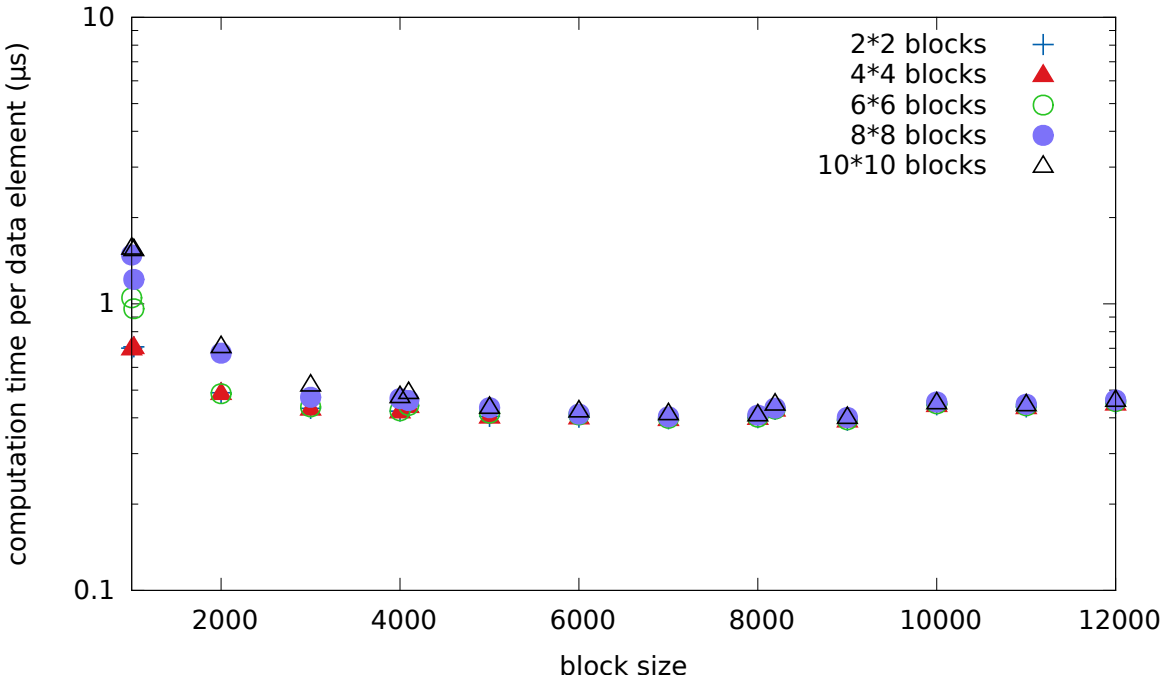


Figure 11.1 — Average computation time for a data element

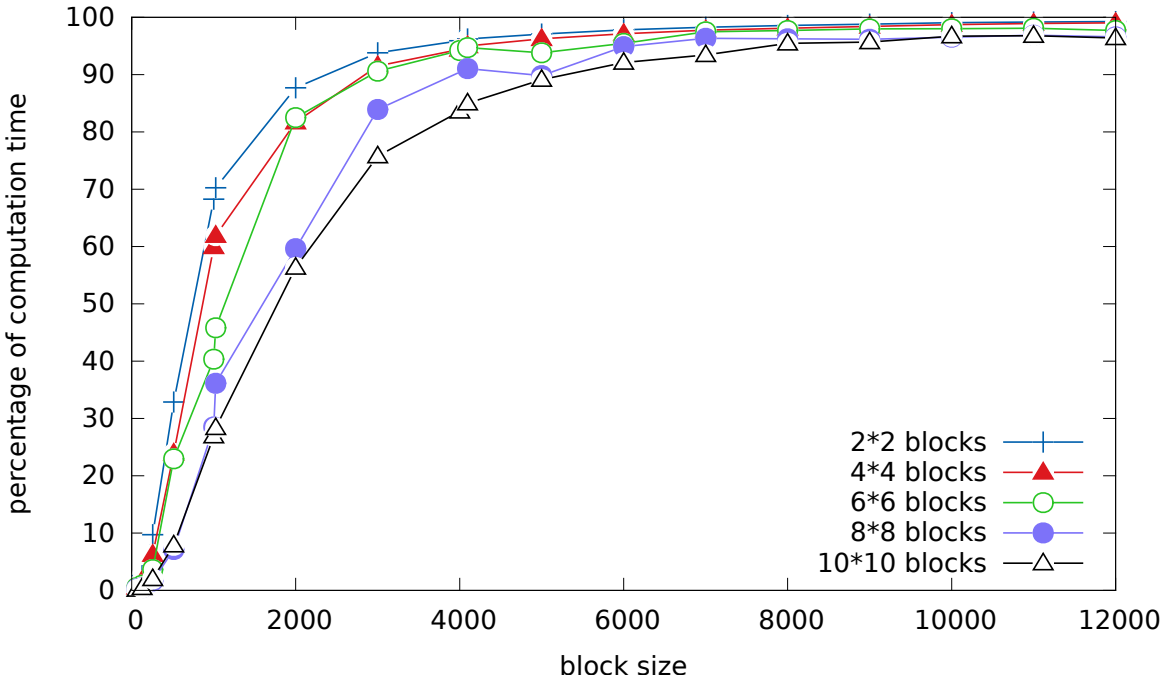


Figure 11.2 — Percentage of computation time

2. Dido generated ORWL code with temporal blocking.

3. Dido generated ORWL code with temporal blocking and Pluto's data locality optimization. This solution doesn't use OpenMP.

4. Dido generated ORWL code with temporal blocking and Pluto's data locality optimization and parallelization. ORWL and OpenMP are both used to guarantee parallelism on different levels.

5. An MPI Stencil code.

For the three first configurations, we place one task/block per hyperthreaded core, that is 32 tasks/blocks per node on $1,4,8,12,\ldots,48$ nodes of the *Grisou* cluster. So, the largest configuration consists of 1536 tasks and blocks. The considered block repartitions and block sizes in single precision floats are given in Table 11.1 and Table 11.2, respectively. For the largest configurations, there is about 14 and 420 billion elements for the 2D and 3D case, respectively. For the fourth configuration, one compute node is reserved for each block having a total equivalent problem size per node as the three first configurations. The OpenMP runtime then uses as many threads as there are hypercores, so in total for the largest configuration this amounts again to 1536 OpenMP threads over the whole system.

Given the complexity of MPI stencil codes, reference implementations to which we can compare the performance of our code were difficult to find. Therefore, we have considered in-house hand-written MPI implementations.

For the Livermore Kernel 23 benchmark, an additional configuration is launched. It consists of a handwritten code that has been implemented by an expert in parallel programming, and in the ORWL model in particular. This code follows the modeling presented in Subsection 4.4 and has been used in [GJ11] to evaluate the efficiency of the ORWL model. Similarly to the three first configurations, we place 32 tasks/blocks per node on $1,4,8,12,\ldots,48$ nodes of the *Grisou* cluster.

Results are shown in Figures 11.3-11.7. Figure 11.3, Figure 11.4 and Figure 11.5 relate the achieved 10^9 stencil operations per second (GStencils/s) to the number of nodes for the considered benchmarks with the different configurations. To better perceive the speedups, summarized weak scaling results for 2D and 3D stencils on 4, 16 and 48 nodes are given in Figure 11.6 and Figure 11.7, respectively.

- **Weak Scaling:**

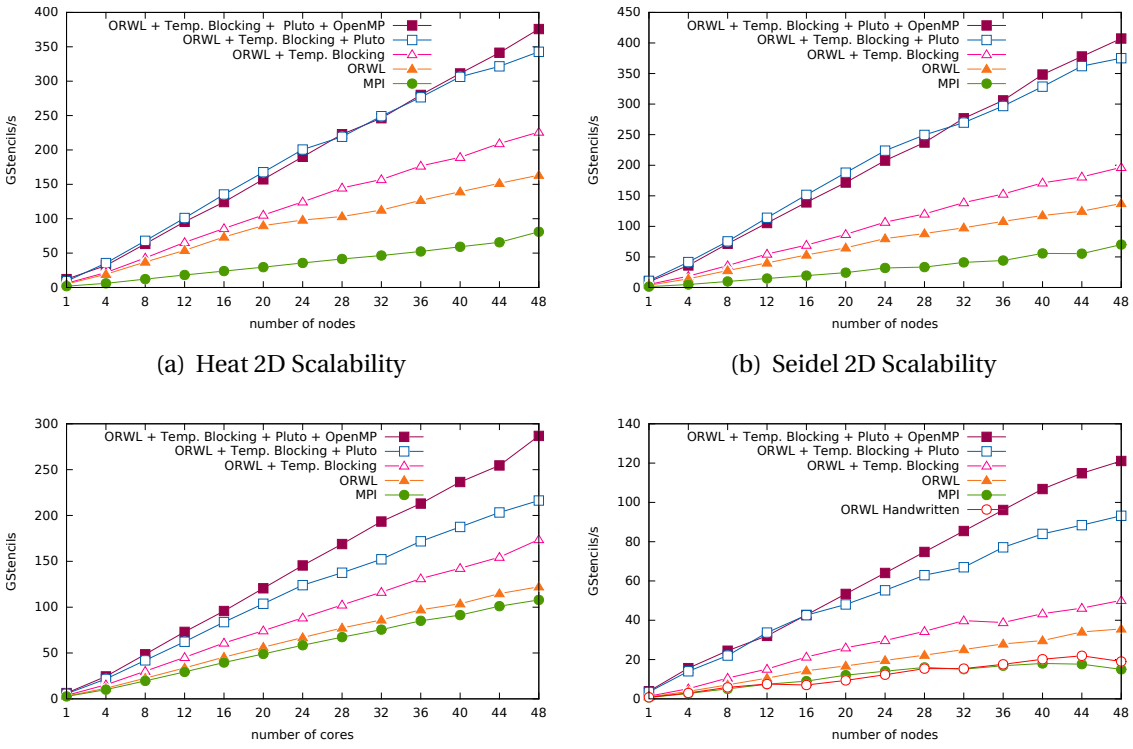
As shown in Figure 11.3, Figure 11.4 and Figure 11.5, all of the 5 implementations, including MPI, show linear weak scaling, *i.e.*, the number of GStencils/s grows linearly with the number of nodes that is at the disposal of the application. This remains valid for different stencil characteristics, *i.e.*, dimension, neighborhood, grid traversal and boundary conditions.

Table 11.1 — Block Repartitions

Dimension	**3D**		**2D**	**4D**	**5D**
Number of Nodes	**32 Tasks per Node**	**16 Tasks per Node**	**32 Tasks per Node**	**32 Tasks per Node**	**32 Tasks per Node**
1	$4 \times 4 \times 2$	$4 \times 2 \times 2$	8×4	$4 \times 2 \times 2 \times 2$	$2 \times 2 \times 2 \times 2 \times 2$
4	$8 \times 4 \times 4$	$4 \times 4 \times 4$	16×8	$4 \times 4 \times 4 \times 2$	$4 \times 4 \times 2 \times 2 \times 2$
8	$8 \times 8 \times 4$	$8 \times 4 \times 4$	16×16	$4 \times 4 \times 4 \times 4$	$4 \times 4 \times 4 \times 2 \times 2$
12	$8 \times 8 \times 6$	$8 \times 6 \times 4$	24×16	$8 \times 4 \times 4 \times 3$	$4 \times 4 \times 4 \times 3 \times 2$
16	$8 \times 8 \times 8$	$8 \times 8 \times 4$	32×16	$8 \times 4 \times 4 \times 4$	$8 \times 4 \times 4 \times 2 \times 2$
20	$10 \times 8 \times 8$	$8 \times 8 \times 5$	32×20	$8 \times 5 \times 4 \times 4$	$8 \times 5 \times 4 \times 2 \times 2$
24	$12 \times 8 \times 8$	$8 \times 8 \times 6$	32×24	$8 \times 6 \times 4 \times 4$	$8 \times 6 \times 4 \times 2 \times 2$
28	$14 \times 8 \times 8$	$8 \times 8 \times 7$	32×28	$8 \times 7 \times 4 \times 4$	$8 \times 7 \times 4 \times 2 \times 2$
32	$16 \times 8 \times 8$	$8 \times 8 \times 8$	32×32	$8 \times 8 \times 4 \times 4$	$8 \times 8 \times 4 \times 2 \times 2$
36	$12 \times 12 \times 8$	$12 \times 8 \times 6$	36×32	$9 \times 8 \times 4 \times 4$	$9 \times 8 \times 4 \times 2 \times 2$
40	$16 \times 10 \times 8$	$10 \times 8 \times 8$	40×32	$10 \times 8 \times 4 \times 4$	$10 \times 8 \times 4 \times 2 \times 2$
44	$22 \times 8 \times 8$	$11 \times 8 \times 8$	44×32	$11 \times 8 \times 4 \times 4$	$11 \times 8 \times 4 \times 2 \times 2$
48	$16 \times 12 \times 8$	$12 \times 8 \times 8$	48×32	$8 \times 8 \times 6 \times 4$	$8 \times 8 \times 6 \times 2 \times 2$

Table 11.2 — Block sizes

Benchmark	**Block Size**
2D Heat ADI	3000^2
2D Seidel	3000^2
2D Wave	3000^2
Livermore	3000^2
3D Heat	650^3
3D Seidel	650^3
3D order-k Wave	450^3
3D 27-pt Jacobi	650^3
3D GSRB	650^3
4D 9-pt Jacobi	140^4
5D 11-pt Jacobi	50^5



(a) Heat 2D Scalability

(b) Seidel 2D Scalability

(c) Wave 2D Scalability

(d) Livermore 2D Scalability

Figure 11.3 — Dido Generated Code Scalability for 2D Benchmarks

- **Comparison with Handcrafted Code:**

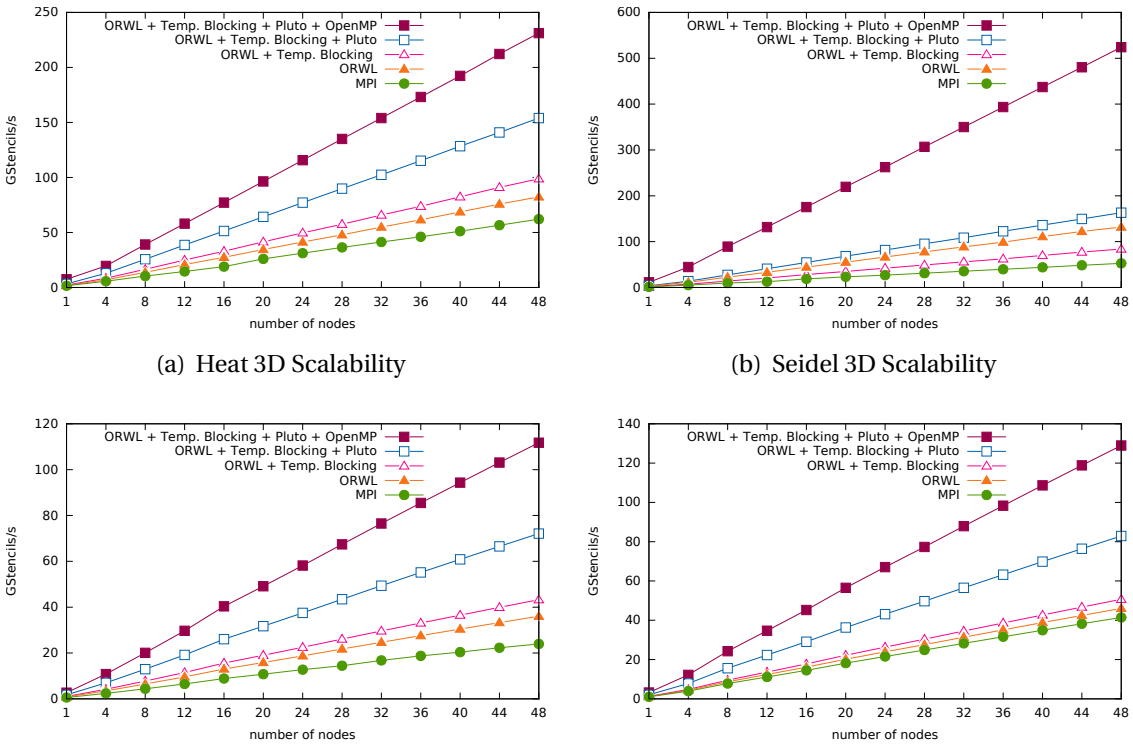
Figure 11.3(d) shows that Dido generated code outperforms the handwritten Livermore Kernel 23 with an average speedup of 1.33×.

- **Comparison with MPI:**

As shown in Figures 11.3-11.7, all of the four ORWL stencil code configurations, including ORWL code using only standard compiler optimizations, outperform the MPI stencil implementation. ORWL optimization-free code outperforms MPI stencil code with an average speedup over all benchmarks of 1.28×. ORWL code optimized for data locality using Pluto following the (3) and (4) configurations outperform MPI stencil code with average speedups of 2.31× and 3.60×, respectively.

- **Temporal Blocking:**

As shown in Figures 11.3-11.7, in most of the cases, temporal blocking is beneficial to the performance. The only exceptions are the 3D Seidel and 3D wave (k=12) benchmarks for



(a) Heat 3D Scalability

(b) Seidel 3D Scalability

(c) GSRB Scalability

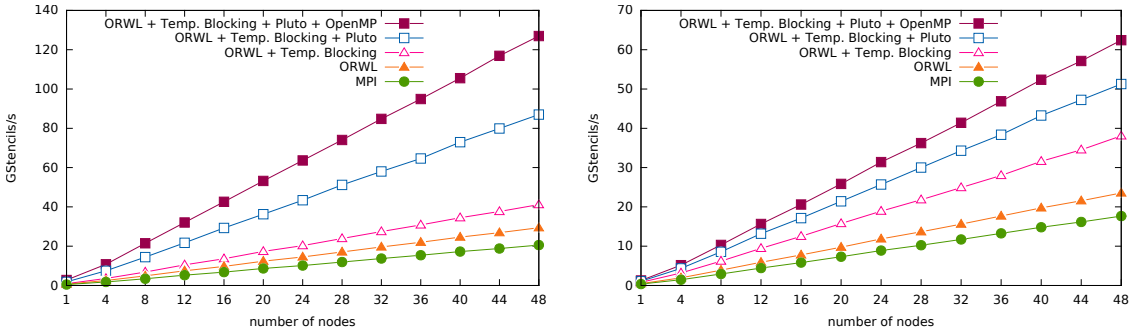
(d) Jacobi 27-pt Scalability

Figure 11.4 — Dido Generated Code Scalability for 3D Heat, Seidel, GSRB and 27-pt Jacobi

which the temporal blocking leads to a slight slow down. For the 3D wave (k=12) benchmark, given the large halo regions, the redundant computations and extra loading that are introduced by temporal blocking counter any gains in temporal locality and any reduction in communication and synchronization overhead. However, it is this same temporal blocking that, when combined to Pluto loop transformations, results in high speedups of 3.97× and 1.43×, respectively.

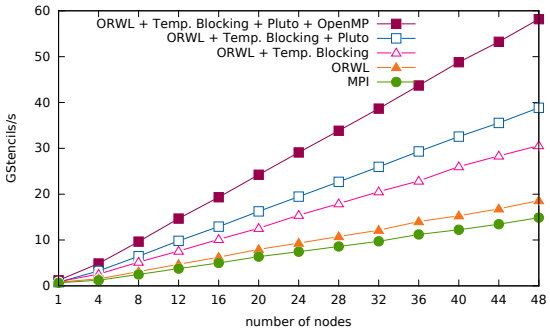
- **Data Locality Optimizations:**

As shown in Figures 11.3-11.7, both configurations (3) and (4) lead to substantial gains in performance. The hybrid solution (4) gave better performance results on most of the benchmarks, especially the 3Ds. Its average speedup over all benchmarks is 2.82× compared to 1.88× for the solution (3). This can be explained by the fact that while solution (3) requires intra-node ORWL communication and synchronization between the different tasks inside one node, the tiling capabilities of Pluto in solution (4) result in dependency-free tiles that can be parallelized using OpenMP without inter-tile communications.

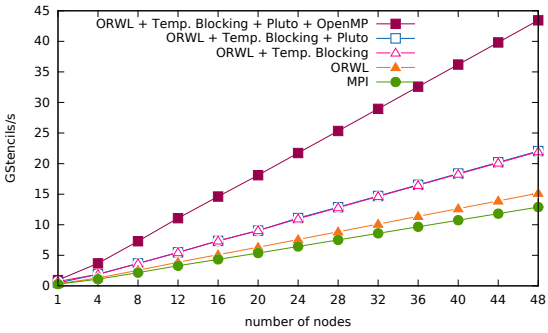


(a) Wave 3D Scalability

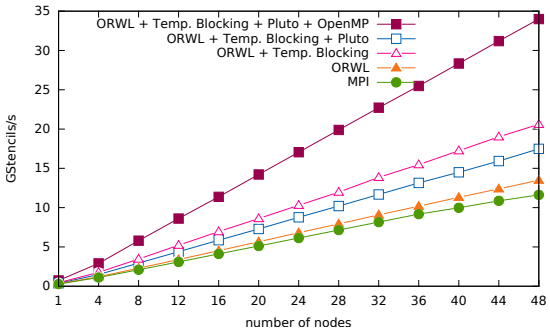
(b) Wave 3D k=4 Scalability



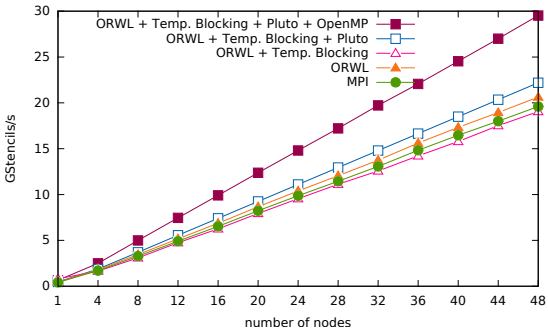
(c) Wave 3D k=6 Scalability



(d) Wave 3D k=8 Scalability

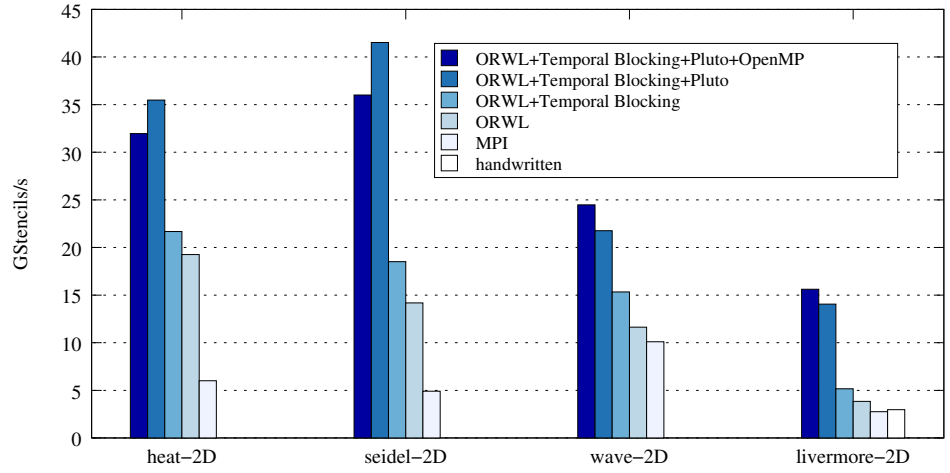


(e) Wave 3D k=10 Scalability

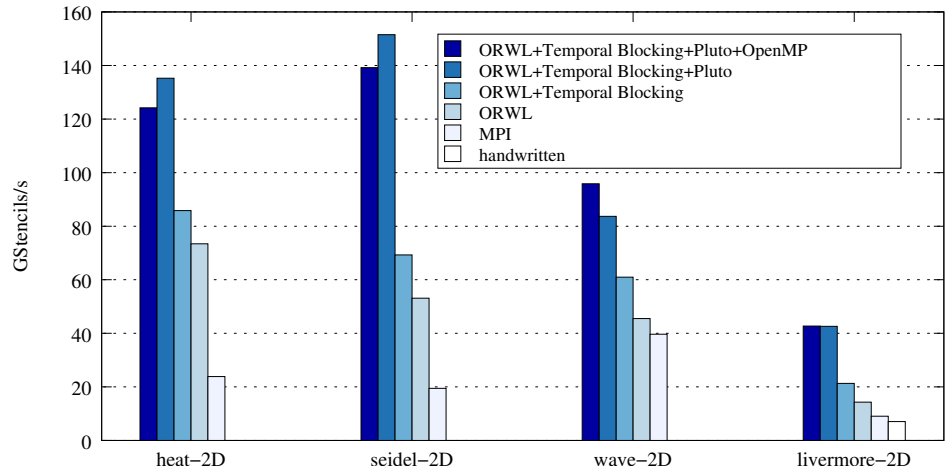


(f) Wave 3D k=12 Scalability

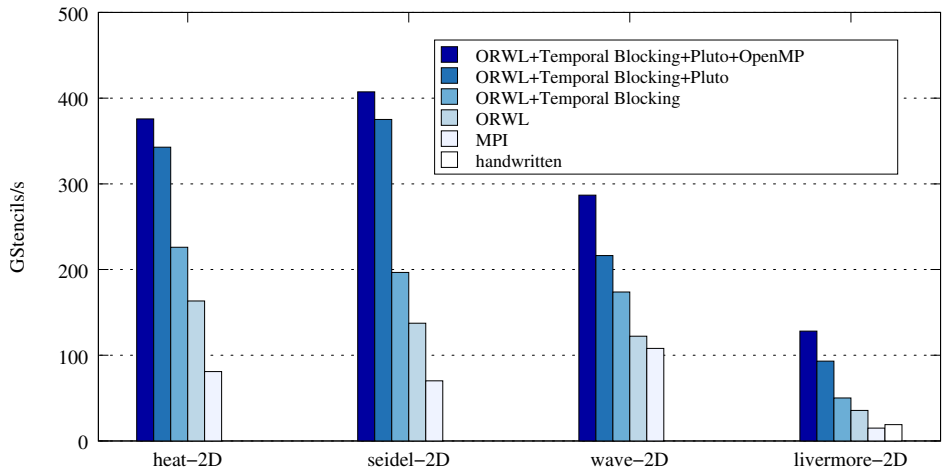
Figure 11.5 — Dido Generated Code Scalability for 3D Wave Equations



(a) 4 nodes

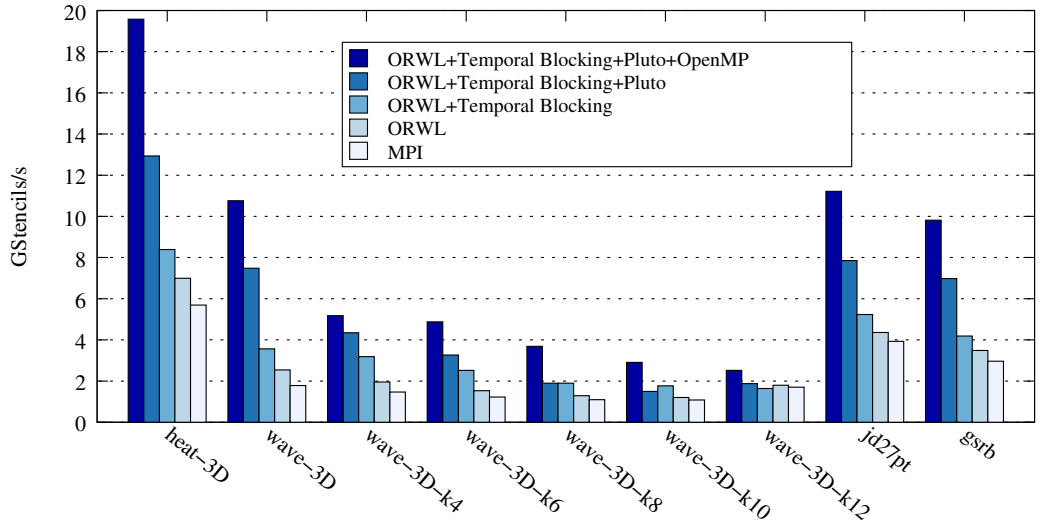


(b) 16 nodes

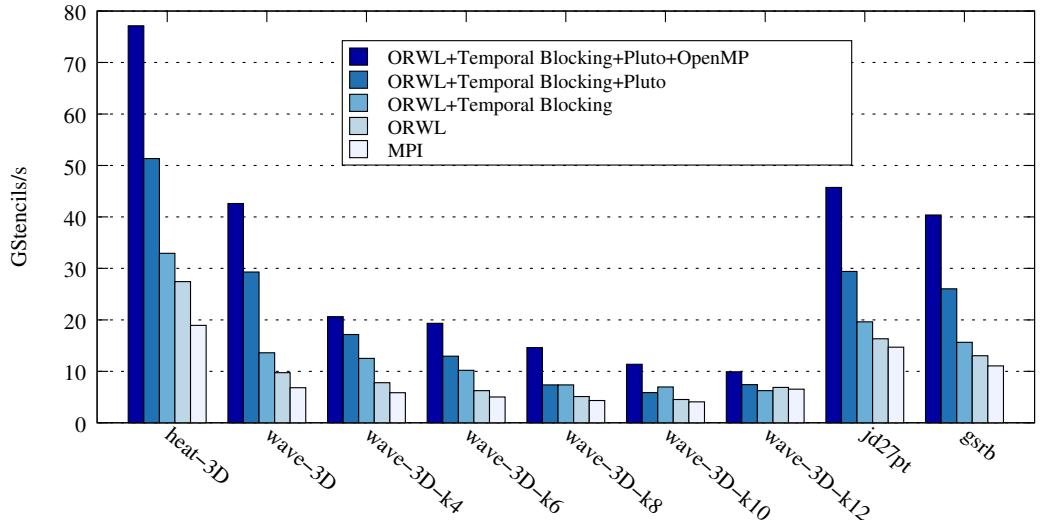


(c) 48 nodes

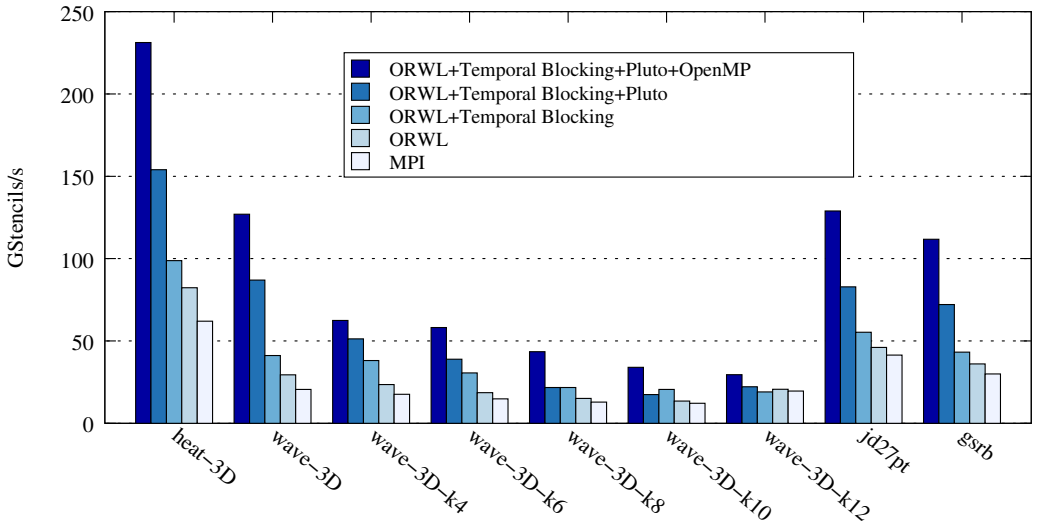
Figure 11.6 **—** Summary of Dido generated code performance for 2D benchmarks: GStencils/s achieved by different configurations on 4, 16 and 48 nodes



(a) 4 nodes

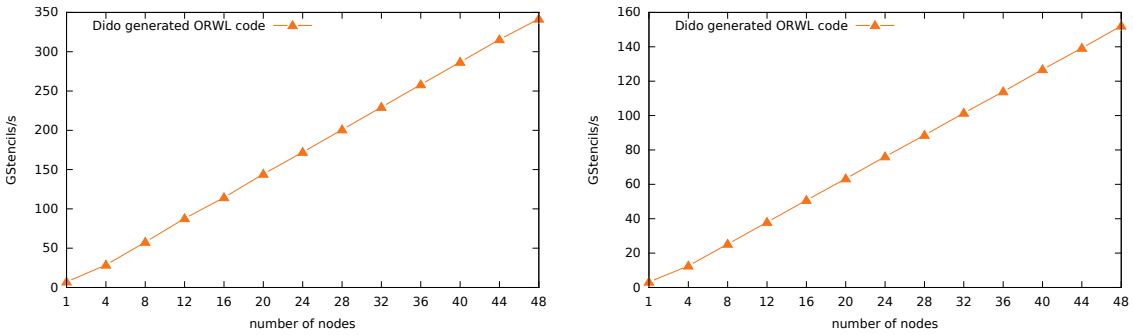


(b) 16 nodes



(c) 48 nodes

Figure 11.7 — Summary of Dido generated code performance for 3D benchmarks: GStencils/s achieved by different configurations on 4, 16 and 48 nodes



(a) 4D Jacobi Scalability

(b) 5D Jacobi Scalability

Figure 11.8 — Dido Generated Code Scalability for 4D and 5D Jacobis

11.2.3 Setup 3

In this setup of experiments, we study the scalability of the generated code for stencils with higher dimensions, *i.e.*, 4D and 5D stencils, by increasing the global problem size. We place one task/block per hyperthreaded core, that is 32 tasks/blocks per node on $1,4,8,12,\ldots,48$ nodes of the *Grisou* cluster. The largest configuration consists thus of 1536 tasks and blocks. The considered block repartitions and block sizes in single precision floats are given in Table 11.1 and Table 11.2, respectively. Results are shown in Figure 11.8. They relate the achieved 10^9 stencil operations per second (GStencils/s) to the number of nodes.

- **Weak Scaling:**

Figure 11.8 shows that even for stencils with higher dimensions, adding compute nodes/processes with equal problem size does not alter the execution time.

11.2.4 Setup 4

In this setup of experiments, we study the scalability of the generated code of the molecular dynamics real-world application, by increasing the global problem size. We place one task/block per physical core, that is 16 tasks/blocks per node on $1,4,8,12,\ldots,48$ nodes of the *Grisou* cluster. Each task computes a block of size 50^3 boxes. We launch three configurations with different numbers of particles per box: 100, 150 and 200 particles per box. For the largest configurations, these are 9.6×10^9, 14.4×10^9, 19.2×10^9 particles in total per global problem. Results are depicted in Figure 11.9. It relates the number of nodes to achieved 10^6 box computations per second (MBoxes/s).

- **Weak Scaling:**

As depicted in Figure 11.9, the generated code for the molecular dynamics application shows also linear weak scaling for different particle numbers per box.

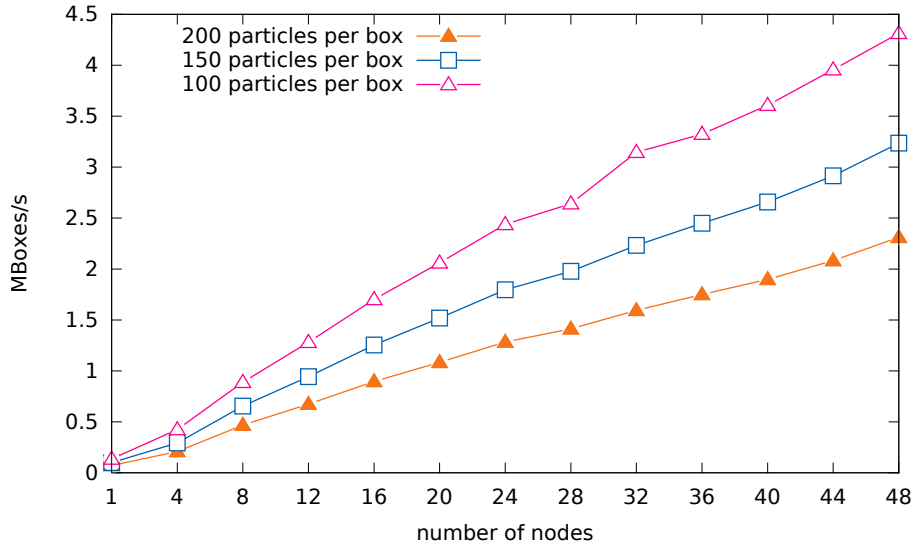


Figure 11.9 — Scalability of Three-Body Forces Molecular Dynamics Application

11.3 Conclusion

Dido generated code is scalable and leverages the efficiency and scalability properties of the ORWL model. This is valid for benchmarks with different stencil characteristics as well as compute-intensive real-world applications, It also outperforms both ORWL and MPI handwritten implementations. This is due to the expert knowledge that went into the pattern defined in Chapter 7 and the quality of the generated code. The *CompUp* form, in particular, ensures for tasks a certain independence from their neighbors. In fact, as soon as the shadow regions are updated in the *main data* location, the *shadow location* buffers are released and ready to receive next data updates from neighbors without interfering with the computation. As soon as the *Compute* operation needs updated values, those are already available and ready to be read. As a result, *Compute* operations of neighboring tasks can be executed simultaneously, which considerably improves the execution times.

As for the applied optimizations, the temporal blocking improves in most cases the computation time. Additionally, the combination of the temporal blocking with Pluto loop transformations provides high speedups. Both configurations using Pluto for data locality lead to substantial gains in performance. However, the hybrid configuration (4), using OpenMP to guarantee intra-node parallelism, gave better performance results on most of the benchmarks. So, this hybrid solution is a clear candidate for an automated default choice for Dido users that don't want to perform runtime calibrations.

CHAPTER 12 Conclusion

We have presented Dido, an implicitly parallel domain-specific language for ORWL stencil code generation. It allows a large programmer community to easily and safely implement parallel stencil codes and leverage, at a low cost, the ORWL model properties. Dido supports general multidimensional stencil computations. It alleviates the burden of writing complex code for distributed-memory architectures and spares the user all the parallelism details necessary for writing highly efficient stencil code. Dido achieves both productivity and performance benefits. It meets the needs of real-world applications by supporting multiple data types and boundary conditions.

The DSL captures high-level stencil abstractions written by the user in a simple syntax and generates all the complex parts such as the shadow region updates and the lock handle positions in the FIFOs. This presents a considerable improvement in terms of programmer productivity.

Additionally, experiments have shown that productivity and performance, often considered as antagonistic, can be reconciled when using Dido. The generated code is scalable and achieves competitive performance that outperforms hand-crafted code. This is due to the domain-specific knowledge about ORWL the stencil computational pattern that went into the defined patterns. It has to be noted that the *CompUp* form can be used for different ORWL applications not only for stencil computations. It ensures expressiveness, deadlock-freeness and better performance for ORWL programs.

Additionally, we show that the ORWL code following the suggested pattern is well-structured and lends itself to different optimizations among which temporal blocking that we use in some cases in order to reduce the communication overhead and minimize data transfers. We combine the temporal blocking optimization with the intra-node data reuse capabilities of the polyhedral loop optimizer *Pluto*. This has considerably improved the performance of the generated code.

We also show that Dido has the expression power to model real-world stencil-based applications and generated code for two different real-world applications. The readability of the generated code enables the user to easily take it in hand. They can therefore easily modify it. Dido can thus become an interactive tool that assists users to implement large applications that comprise stencil computations and generate code for distributed-memory architectures.

CHAPTER 13 Future Work

Dido provides both performance and programmer productivity benefits for multi-dimensional stencil computations on distributed memory architectures. However, there are still ideas to revisit in order to further enhance the framework on different levels.

Multi-Stencils

Stencils involve a large panel of computations. In this work, we focus on iterative one-stencil computations applied on one input grid, where the input and output grids have the same structure. There is hence one *main data* grid. These involve the most commonly used stencil algorithms. However, some applications may contain multiple stencil computations, *i.e.*, computations that perform stencils on different *main data* grids. Among the short-term goals, the priority would be to extend Dido to handle multi-stencil computations. From a development point of view, this does not require much of effort, as all of the suggested patterns and techniques remain valid. However, synchronization has to be added.

Autotuning Tile Sizes

For the moments, tiles sizes taken by Pluto are default. Pluto offers the possibility to specify tile sizes in a separate file. One way of enhancing Dido is to autotune tile sizes for optimal performance.

Adaptive Mesh Refinement

In real-world applications, some regions of the *main data* grid may contain details that are more difficult to approximate and require finer blocks or meshes, compared to the rest of the grid. For example, solutions for systems of hyperbolic partial differential equations (PDE) are often smooth and easily approximated over large portions of their domains, but may contain boundary layers or locally isolated internal regions with steep gradients, shocks, or discontinuities. Adaptive mesh refinement [BO84] is a finite-difference method that consists in adaptively placing finer grids in these regions, or removing existing ones

to attain a given accuracy with a minimum cost. The approach is recursive and values on the boundaries of the finer grids are defined using interpolation.

Dido could be enhanced by supporting adaptive mesh refinement. An intermediary step would be to allow the definition of different stencils over different domains of the *main data* grid. The main difficulty in implementing adaptive mesh refinement in Dido lies in disconnecting the ORWL lock handles and linking them with new locations which is challenging because of the static nature of ORWL and its initialization step.

Adaptive ORWL

Computational science and engineering applications may sometimes exhibit irregular structure. It is thus difficult to subdivide the problem such that every partition has equal computational load. In addition, computational load requirements of each partition may vary as computation progresses as for Adaptive Mesh Refinement. Among the long-term goals, ORWL and thus Dido could be enhanced by adding dynamic load balancing features *i.e.*, re-mapping partitions to physical processors in response to variation in load conditions. Given the static nature of ORWL and its initialization step, this is particularly challenging as it implies disconnecting the ORWL lock handles and linking them with new locations.

Heterogeneous Architectures

For the moment, Dido targets distributed-memory clusters of CPUs. One way to enhance Dido is to support heterogeneous architectures with GPU accelerators and Xeon PHI manycore processors. ORWL can be used to ensure both inter-node communication as well as CPU-GPU communication.

Affinity

In [GJM16b], Gustedt *et al.* were able to automatically map the task graph of ORWL applications to the hierarchy of the execution platform. For the shared memory part, the ORWL runtime thereby ensures a better placement of tasks. In particular, it can then guarantee that exactly one compute thread is executed per physical core, and that threads that exchange data are placed close in the hierarchy. This strategy minimizes migration of threads and drastically reduces the number of cache misses and pipeline stalls. First benchmark results are very promising and outperform affinity placement by means of OpenMP by a substantial amount. We are confident that a combination of this strategy with Dido will allow to improve the performance of the generated programs even more.

Sparse Computations

To cover a wider range of applications, one promising direction are sparse matrices and applications. This type of algorithms is in widespread use by scientific communities and

lacks user-friendly specific tools for distributed-memory architectures. The ORWL library provides tools that can be particularly useful for sparse computations.

Bibliography

[AFB13] Christophe AVENEL, Pierre FORTIN et Dominique BÉRÉZIAT : Parallel birth and death process for cell nuclei extraction in histopathology images. *In Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 429–438. IEEE, 2013.

[ATR98] Frank J AHERNE, Neil A THACKER et Peter I ROCKETT : The bhattacharyya metric as an absolute similarity measure for frequency coded data. *Kybernetika*, 34(4):363–368, 1998.

[Bas02] Cédric BASTOUL : Generating loops for scanning polyhedra. *PRiSM, Versailles University*, 23, 2002.

[Bas04] Cédric BASTOUL : *Improving Data Locality in Static Control Programs.* Thèse de doctorat, University Paris 6, Pierre et Marie Curie, France, décembre 2004.

[Bas08] Cédric BASTOUL : Extracting polyhedral representation from high level languages. *Tech. rep. Related to the Clan tool. LRI, Paris-Sud University*, 2008.

[Bas11] Cédric BASTOUL : Openscop: A specification and a library for data exchange in polyhedral compilation tools. Rapport technique, tech. rep., Paris-Sud University, France, 2011.

[Bas12] Cédric BASTOUL : *Contributions to high-level program optimization.* Thèse de doctorat, Habilitation Thesis. Paris-Sud University, France, 2012.

[Bas13] Cédric BASTOUL : Cloog: The chunky loop generator.(2013), 2013.

[BBB+91] David H BAILEY, Eric BARSZCZ, John T BARTON, David S BROWNING, Robert L CARTER, Leonardo DAGUM, Rod A FATOOHI, Paul O FREDERICKSON, Thomas A LASINSKI, Rob S SCHREIBER *et al.* : The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[BBG+98] Hesheng BAO, Jacobo BIELAK, Omar GHATTAS, Loukas F KALLIVOKAS, David R O'HALLARON, Jonathan R SHEWCHUK et Jifeng XU : Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer methods in applied mechanics and engineering*, 152(1-2):85–102, 1998.

[BBP17] Uday BONDHUGULA, Vinayaka BANDISHTI et Irshad PANANILATH : Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, 2017.

[Ber96] Herman JC BERENDSEN : Bio-molecular dynamics comes of age. *Science*, 271(5251):954–954, 1996.

[BHRS08] Uday BONDHUGULA, Albert HARTONO, Jagannathan RAMANUJAM et Ponnuswamy SADAYAPPAN : A practical automatic polyhedral parallelizer and locality optimizer. *In ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[BO84] Marsha J BERGER et Joseph OLIGER : Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512, 1984.

[Bon09] Uday BONDHUGULA : Pluto-an automatic parallelizer and locality optimizer for multicores, 2009.

[Bon13] Uday BONDHUGULA : Compiling affine loop nests for distributed-memory parallel architectures. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 33. ACM, 2013.

[BRHS92] Rainer BLECK, Claes ROOTH, Dingming HU et Linda T SMITH : Salinity-driven thermocline transients in a wind-and thermohaline-forced isopycnic coordinate model of the north atlantic. *Journal of Physical Oceanography*, 22(12):1486–1505, 1992.

[CB95] Thomas A CORTESE et S BALACHANDAR : High performance spectral simulation of turbulent flows in massively parallel machines with distributed memory. *The International journal of supercomputer applications and high performance computing*, 9(3):187–204, 1995.

[CCH08] Chun CHEN, Jacqueline CHAME et Mary HALL : Chill: A framework for composing high-level loop transformations. Rapport technique, Citeseer, 2008.

[CG10a] Pierre-Nicolas CLAUSS et Jens GUSTEDT : Experimenting iterative computations with ordered read-write locks. *In* Marco DANELUTTO, Tom GROSS et Julien BOURGEOIS, éditeurs : *18th Euromicro International Conference on Parallel, Distributed and network-based Processing*, pages 155–162, Pisa, Italy, 2010. IEEE.

[CG10b] Pierre-Nicolas CLAUSS et Jens GUSTEDT : Iterative computations with ordered read-write locks. *Journal of Parallel and Distributed Computing*, 70(5): 496–504, 2010.

[CHZ11] Jason CONG, Muhuan HUANG et Yi ZOU : Accelerating fluid registration algorithm on multi-FPGA platforms. *In Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 50–57. IEEE, 2011.

[CM09] J COHEN et M Jeroen MOLEMAKER : A fast double precision cfd code using cuda. *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 414–429, 2009.

[CSB11] Matthias CHRISTEN, Olaf SCHENK et Helmar BURKHART : Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. *In Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[CSN+10] Matthias CHRISTEN, Olaf SCHENK, Esra NEUFELD, Maarten PAULIDES et Helmar BURKHART : Manycore stencil computations in hyperthermia applications. *Scientific Computing with Multicore and Accelerators*, pages 255–277, 2010.

[CZ08] Jason CONG et Yi ZOU : Lithographic aerial image simulation with FPGA-based hardware acceleration. *In Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 67–76. ACM, 2008.

[DHK+00] Craig C DOUGLAS, Jonathan HU, Markus KOWARSCHIK, Ulrich RÜDE et Christian WEISS : Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.

[DJ05] A DAVID et Hennessy JOHN : Computer organization and design: the hardware/software interface. *San mateo, CA: M organ Kaufmann Publishers*, 1:998, 2005.

[DKW+09] Kaushik DATT, Shoaib KAMIL, Samuel WILLIAMS, Leonid OLIKER, John SHALF et Katherine YELICK : Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 51(1):129–159, 2009.

[DMV+08] Kaushik DATTA, Mark MURPHY, Vasily VOLKOV, Samuel WILLIAMS, Jonathan CARTER, Leonid OLIKER, David PATTERSON, John SHALF et Katherine YELICK : Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *In Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[DOR+99] Fiona S DOUGLAS, Lindsay C O'DAIR, Marion ROBINSON, D Gareth R EVANS et Sally A LYNCH : The accuracy of diagnoses as reported in families with cancer: a retrospective study. *Journal of medical genetics*, 36(4):309–312, 1999.

[DRRB13] Roshan DATHATHRI, Chandan REDDY, Thejas RAMASHEKAR et Uday BONDHUGULA : Generating efficient data movement code for heterogeneous architectures with distributed-memory. *In Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 375–386. IEEE, 2013.

[EMBS17] Christopher EARL, Matthew MIGHT, Abhishek BAGUSETTY et James C SUTHERLAND : Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations. *Journal of Systems and Software*, 125:389–400, 2017.

[FJ05] Matteo FRIGO et Steven G JOHNSON : The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[FLPR99] Matteo FRIGO, Charles E LEISERSON, Harald PROKOP et Sridhar RAMACHANDRAN : Cache-oblivious algorithms. *In Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[FRR+07] Matthew FLUET, Mike RAINEY, John REPPY, Adam SHAW et Yingqi XIAO : Manticore: A heterogeneous parallel language. *In Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 37–44. ACM, 2007.

[FS05] M FRIGO et V STRUMPEN : Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. *In Proc. of the 19th ACM International Conference on Supercomputing (ICS05), Boston, MA*, 2005.

[GAK03] Georgios GOUMAS, Maria ATHANASAKI et Nectarios KOZIRIS : An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, 2003.

[GCK+13] Tobias GROSSER, Albert COHEN, Paul HJ KELLY, J RAMANUJAM, P SADAYAPPAN et Sven VERDOOLAEGE : Split tiling for gpus: automatic parallelization using trapezoidal tiles. *In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 24–31. ACM, 2013.

[GJ11] Jens GUSTEDT et Emmanuel JEANVOINE : Relaxed synchronization with ordered read-write locks. *In* Michael ALEXANDER *et al.*, éditeurs : *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 de *LNCS*, pages 387–397, Bordeaux, France, août 2011. Springer.

[GJM16a] Jens GUSTEDT, Emmanuel JEANNOT et Farouk MANSOURI : Fully-abstracted affinity optimization for task-based models. Research Report RR-8993, INRIA Nancy, décembre 2016.

[GJM16b] Jens GUSTEDT, Emmanuel JEANNOT et Farouk MANSOURI : Optimizing locality by topology-aware placement for a task based programming model. *In*

Cluster Computing (CLUSTER), 2016 IEEE International Conference on, pages 164–165. IEEE, 2016.

[GKV12] Pieter GHYSELS, P KŁOSIEWICZ et Wim VANROOSE : Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Linear Algebra with Applications*, 19(2):253–267, 2012.

[GVM14] Jens GUSTEDT, Stéphane VIALLE et Patrick MERCIER : Resource Centered Computing delivering high parallel performance. *In Heterogeneity in Computing Workshop (HCW 2014)*, Heterogeneity in Computing Workshop (HCW 2014), workshop of 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014), Phenix, AZ, United States, mai 2014. IEEE.

[HCC+09] Mary HALL, Jacqueline CHAME, Chun CHEN, Jaewook SHIN, Gabe RUDY et Malik Murtaza KHAN : Loop transformation recipes for code generation and auto-tuning. *In International Workshop on Languages and Compilers for Parallel Computing*, pages 50–64. Springer, 2009.

[HDS96] William HUMPHREY, Andrew DALKE et Klaus SCHULTEN : Vmd: visual molecular dynamics. *Journal of molecular graphics*, 14(1):33–38, 1996.

[HHV+] Tom HENRETTY, Justin HOLEWINSKI, Richard VERAS, Franz FRANCHETTI, Louis-Noël POUCHET, J RAMANUJAM, Atanas ROUNTEV et P SADAYAPPAN : A domain-specific language and compiler for stencil computations on short-vector simd and gpu architectures.

[HPS12] Justin HOLEWINSKI, Louis-Noël POUCHET et Ponnuswamy SADAYAPPAN : High-performance code generation for stencil computations on gpu architectures. *In Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[HSP+11] Tom HENRETTY, Kevin STOCK, Louis-Noël POUCHET, Franz FRANCHETTI, J RAMANUJAM et P SADAYAPPAN : Data layout transformation for stencil computations on short-vector simd architectures. *In Compiler Construction*, pages 225–245. Springer, 2011.

[HVF+13] Tom HENRETTY, Richard VERAS, Franz FRANCHETTI, Louis-Noël POUCHET, Jagannathan RAMANUJAM et Ponnuswamy SADAYAPPAN : A stencil compiler for short-vector simd architectures. *In Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[KBB+07] Sriram KRISHNAMOORTHY, Muthu BASKARAN, Uday BONDHUGULA, Jagannathan RAMANUJAM, Atanas ROUNTEV et Ponnuswamy SADAYAPPAN : Effective automatic parallelization of stencil computations. *In ACM sigplan notices*, volume 42, pages 235–244. ACM, 2007.

[KCO+10] Shoaib KAMIL, Cy CHAN, Leonid OLIKER, John SHALF et Samuel WILLIAMS : An auto-tuning framework for parallel multicore stencil computations. *In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[KLH+04] K KADAU, PS LOMDAHL, BL HOLIAN, TC GERMANN, D KADAU, P ENTEL, DE WOLF, M KRETH et F WESTERHOFF : Molecular-dynamics study of mechanical deformation in nano-crystalline aluminum. *Metallurgical and materials transactions A*, 35(9):2719–2723, 2004.

[KW01] Markus KOWARSCHIK et Christian WEISS : Dimepack–a cache-optimized multigrid library. *In PROC. OF THE INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS (PDPTA 2001), VOLUME I*. Citeseer, 2001.

[LAB+14] Christian LENGAUER, Sven APEL, Matthias BOLTEN, Armin GRÖSSLINGER, Frank HANNIG, Harald KÖSTLER, Ulrich RÜDE, Jürgen TEICH, Alexander GREBHAHN, Stefan KRONAWITTER *et al.* : Exastencils: advanced stencil-code engineering. *In European Conference on Parallel Processing*, pages 553–564. Springer, 2014.

[LDG+08] Xavier LEROY, Damien DOLIGEZ, Jacques GARRIGUE, Didier RÉMY et Jérôme VOUILLON : The objective caml system release 3.11. *Documentation and user's manual. INRIA*, 2008.

[Loe99] Vincent LOECHNER : Polylib: A library for manipulating parameterized polyhedra, 1999.

[LS04] Zhiyuan LI et Yonghong SONG : Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6): 975–1028, 2004.

[MF46] Philip McCord MORSE et Herman FESHBACH : *Methods of theoretical physics*. Technology Press, 1946.

[Mic09] Paulius MICIKEVICIUS : 3d finite difference computation on gpus using cuda. *In Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84. ACM, 2009.

[MKCK00] Simanta MITRA, Suresh C KOTHARI, Jaekyu CHO et A KRISHNASWAMY : Paragent: A domain-specific semi-automatic parallelization tool. *In International Conference on High-Performance Computing*, pages 141–148. Springer, 2000.

[MNSM11] Naoya MARUYAMA, Tatsuo NOMURA, Kento SATO et Satoshi MATSUOKA : Physis: an implicitly parallel programming model for stencil computations on

large-scale gpu-accelerated supercomputers. *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 11. ACM, 2011.

[MS11] Jiayuan MENG et Kevin SKADRON : A performance study for iterative stencil loops on gpus with ghost zone optimizations. *International Journal of Parallel Programming*, 39(1):115–142, 2011.

[MVB15] Ravi Teja MULLAPUDI, Vinay VASISTA et Uday BONDHUGULA : Polymage: Automatic optimization for image processing pipelines. *In ACM SIGARCH Computer Architecture News*, volume 43, pages 429–443. ACM, 2015.

[MW04] Hua MENG et Chao-Yang WANG : Large-scale simulation of polymer electrolyte fuel cells by parallel computing. *Chemical Engineering Science*, 59(16): 3331–3343, 2004.

[NKV94] Aiichiro NAKANO, Rajiv K KALIA et Priya VASHISHTA : Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Computer Physics Communications*, 83(2-3):197–214, 1994.

[NSC+10] Anthony NGUYEN, Nadathur SATISH, Jatin CHHUGANI, Changkyu KIM et Pradeep DUBEY : 3.5-d blocking optimization for stencil computations on modern cpus and gpus. *In High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–13. IEEE, 2010.

[Nvi10] CUDA NVIDIA : Programming guide, 2010.

[PE06] Zhelong PAN et Rudolf EIGENMANN : Fast and effective orchestration of compiler optimizations for automatic performance tuning. *In Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332. IEEE Computer Society, 2006.

[PF10] Everett H PHILLIPS et Massimiliano FATICA : Implementing the himeno benchmark with cuda on gpu clusters. *In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[PPS+13] Sander PRONK, Szilárd PÁLL, Roland SCHULZ, Per LARSSON, Pär BJELKMAR, Rossen APOSTOLOV, Michael R SHIRTS, Jeremy C SMITH, Peter M KASSON, David van der SPOEL *et al.* : Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[PRG15] Alyson D PEREIRA, Luiz RAMOS et Luís FW GÓES : Pskel: A stencil programming framework for cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, 27(17):4938–4953, 2015.

[Rap04] Dennis C RAPAPORT : *The art of molecular dynamics simulation*. Cambridge university press, 2004.

[RKBA+13] Jonathan RAGAN-KELLEY, Connelly BARNES, Andrew ADAMS, Sylvain PARIS, Frédo DURAND et Saman AMARASINGHE : Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[RKRS07] Lakshminarayanan RENGANARAYANAN, DaeGon KIM, Sanjay RAJOPADHYE et Michelle Mills STROUT : Parameterized tiled loops for free. *In ACM SIGPLAN Notices*, volume 42, pages 405–414. ACM, 2007.

[RT00] Gabriel RIVERA et Chau-Wen TSENG : Tiling optimizations for 3d scientific computations. *In Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 32. IEEE Computer Society, 2000.

[SC04] Sriram SELLAPPA et Siddhartha CHATTERJEE : Cache-efficient multigrid algorithms. *The International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[SGM16] Mariem SAIED, Jens GUSTEDT et Gilles MULLER : Automatic Code Generation for Iterative Multi-dimensional Stencil Computations. *In* Anne BENOÎT, éditeur : *High Performance Computing, Data, and Analitics*, Hydarabat, India, décembre 2016. IEEE.

[SGRP16] Daniel SALAS, Jens GUSTEDT, Daniel RACOCEANU et Isabelle PERSEIL : Resource-Centered Distributed Processing of Large Histopathology Images. *In 19th IEEE International Conference on Computational Science and Engineering*, Paris, France, août 2016.

[SGS10] John E STONE, David GOHARA et Guochun SHI : Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[SL99] Yonghong SONG et Zhiyuan LI : New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices*, 34(5):215–228, 1999.

[TCC+09] Ananta TIWARI, Chun CHEN, Jacqueline CHAME, Mary HALL et Jeffrey K HOLLINGSWORTH : A scalable auto-tuning framework for compiler optimization. *In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[ŢCH+02] Cristian ŢĂPUŞ, I-Hsin CHUNG, Jeffrey K HOLLINGSWORTH *et al.* : Active harmony: Towards automated performance tuning. *In Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[TCK+11] Yuan TANG, Rezaul Alam CHOWDHURY, Bradley C KUSZMAUL, Chi-Keung LUK et Charles E LEISERSON : The pochoir stencil compiler. *In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[TCLL11] Yuan TANG, Rezaul CHOWDHURY, Chi-Keung LUK et Charles E LEISERSON : Coding stencil computations using the pochoir stencil-specification language. *In Poster session presented at the 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.

[TLCS11] Ananta TIWARI, Michael A LAURENZANO, Laura CARRINGTON et Allan SNAVELY : Auto-tuning for energy usage in scientific applications. *In European Conference on Parallel Processing*, pages 178–187. Springer, 2011.

[Tsc04] D TSCHUMPERLÉ : The cimg library: http://cimg. sourceforge. net. *The C++ Template Image Processing Library*, 2004.

[TU90] Allen TAFLOVE et Korada R UMASHANKAR : The finite-difference time-domain method for numerical modeling of electromagnetic wave interactions. *Electromagnetics*, 10(1-2):105–126, 1990.

[UCB11] Didem UNAT, Xing CAI et Scott B BADEN : Mint: realizing cuda performance in 3d stencil methods with annotated c. *In Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[VCJC+13] Sven VERDOOLAEGE, Juan CARLOS JUEGA, Albert COHEN, Jose IGNACIO GOMEZ, Christian TENLLADO et Francky CATTHOOR : Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[VDY05] Richard VUDUC, James W DEMMEL et Katherine A YELICK : Oski: A library of automatically tuned sparse matrix kernels. *In Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[Ver10] Sven VERDOOLAEGE : isl: An integer set library for the polyhedral model. *In International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[WCO+08] Samuel WILLIAMS, Jonathan CARTER, Leonid OLIKER, John SHALF et Katherine YELICK : Lattice boltzmann simulation optimization on leading multicore platforms. *In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–14. IEEE, 2008.

[WD98] R Clinton WHALEY et Jack J DONGARRA : Automatically tuned linear algebra software. *In Supercomputing, 1998. SC98. IEEE/ACM Conference on*, pages 38–38. IEEE, 1998.

[WHZ+09] Gerhard WELLEIN, Georg HAGER, Thomas ZEISER, Markus WITTMANN et Holger FEHSKE : Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. *In Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 1, pages 579–586. IEEE, 2009.

[Wir71] Niklaus WIRTH : Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[WLSG+06] Brian G WILLIAMS, James O LLOYD-SMITH, Eleanor GOUWS, Catherine HANKINS, Wayne M GETZ, John HARGROVE, Isabelle DE ZOYSA, Christopher DYE et Bertran AUVERT : The potential impact of male circumcision on hiv in sub-saharan africa. *PLoS medicine*, 3(7):e262, 2006.

[Wol89] Michael WOLFE : More iteration space tiling. *In Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.

[Won00] David WONNACOTT : Using time skewing to eliminate idle time due to memory bandwidth and network limitations. *In Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 171–180. IEEE, 2000.

[WYTX13] Qiang WU, Canqun YANG, Tao TANG et Liquan XIAO : Exploiting hierarchy parallelism for molecular dynamics on a petascale heterogeneous system. *Journal of Parallel and Distributed Computing*, 73(12):1592–1604, 2013.

[Xue12] Jingling XUE : *Loop tiling for parallelism*, volume 575. Springer Science & Business Media, 2012.

[ZGG+12] Xing ZHOU, Jean-Pierre GIACALONE, María Jesús GARZARÁN, Robert H KUHN, Yang NI et David PADUA : Hierarchical overlapped tiling. *In Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 207–218. ACM, 2012.

[ZM12] Yongpeng ZHANG et Frank MUELLER : Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. *In Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 155–164. ACM, 2012.

[ZWN+08] Thomas ZEISER, Gerhard WELLEIN, Aditya NITSURE, Klaus IGLBERGER, U RUDE et Georg HAGER : Introducing a parallel cache oblivious blocking approach for the lattice boltzmann method. *Progress in Computational Fluid Dynamics, an International Journal*, 8(1-4):179–188, 2008.

Université de Strasbourg

Mariem SAIED

Automatic Code Generation and Optimization of Multi-dimensional Stencil Computations on Distributed-Memory Architectures

Résumé

Nous proposons Dido, un langage dédié (DSL) implicitement parallèle qui capture les spécifications de haut niveau des stencils et génère automatiquement du code parallèle de haute performance pour les architectures à mémoire distribuée. Le code généré utilise ORWL en tant que interface de communication et runtime. Nous montrons que Dido réalise un grand progrès en termes de productivité sans sacrifier les performances. Dido prend en charge une large gamme de calculs stencils ainsi que des applications réelles à base de stencils. Nous montrons que le code généré par Dido est bien structuré et se prête à de différentes optimisations possibles. Nous combinons également la technique de génération de code de Dido avec Pluto l'optimiseur polyédrique de boucles pour améliorer la localité des données. Nous présentons des expériences qui prouvent l'efficacité et la scalabilité du code généré qui atteint de meilleures performances que les implémentations ORWL et MPI écrites à la main.

Mots-clés: langage dédié, verroux ordonnés de lecture/écriture, calculs stencils, modèle polyédrique, mémoire distribuée

Summary

In this work, we present Dido, an implicitly parallel domain-specific language (DSL) that captures high-level stencil abstractions and automatically generates high-performance parallel stencil code for distributed-memory architectures. The generated code uses ORWL as a communication and synchronization backend. We show that Dido achieves a huge progress in terms of programmer productivity without sacrificing the performance. Dido supports a wide range of stencil computations and real-world stencil-based applications. We show that the well-structured code generated by Dido lends itself to different possible optimizations and study the performance of two of them. We also combine Dido's code generation technique with the polyhedral loop optimizer Pluto to increase data locality and improve intra-node data reuse. We present experiments that prove the efficiency and scalability of the generated code that outperforms both ORWL and MPI hand-crafted implementations.

Key Words: domain-specific language, ordered read write locks, stencil computations, polyhedral model, distributed memory