Compilation pour machines à mémoire répartie : une approche multipasse

Nelson Lossing

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à MINES ParisTech

Compilation pour machines à mémoire répartie : une approche multipasse

École doctorale n°521

SCIENCES ET MÉTIERS DE L'INGÉNIEUR

Spécialité MATHÉMATIQUES, INFORMATIQUE TEMPS-RÉEL, ROBOTIQUE

COMPOSITION DU JURY :

Mme Christine Eisenbeis
Inria, Président

M Henri-Pierre Charles
CEA-LIST, Rapporteur

M Frédéric Desprez
Inria, Rapporteur

Mme Elisabeth Brunet
Telecom SudParis, Examinatrice

M Antoniu Pop
University of Manchester, Examinateur

Mme Corinne Ancourt
MINES ParisTech, Maître de thèse

M François Irigoin
MINES ParisTech, Directeur de thèse

Soutenue par **Nelson Lossing**
le lundi 3 avril 2017

Dirigée par **François Irigoin**
et **Corinne Ancourt**

MINES ParisTech | PSL RESEARCH UNIVERSITY PARIS

Remerciements

Cette thèse a été préparée au Centre de recherche en informatique (CRI), MINES PartisTech, sous la direction de François Irigoin et Corinne Ancourt.

Je souhaite dans un premier temps remercier mon directeur de thèse, François Irigoin, et mon maître de thèse, Corinne Ancourt, qui m'ont proposé de réaliser cette thèse et m'ont encadré, conseillé et encouragé tout au long de mon travail, et avec qui j'ai eu des échanges fructueux, je leur en suis reconnaissant.

Je remercie vivement Christine Eisenbeis d'avoir accepté de présider mon jury, Henri-Pierre Charles et Frédéric Desprez d'avoir accepté d'être mes rapporteurs de thèse, et Elisabeth Brunet et Antoniu Pop de faire partie de mon jury.

Je remercie chaleureusement tous les membres de mon laboratoire pour leur accueil et tout le temps que j'ai pu passer avec eux, aussi bien studieux que récréatif. Particulièrement, Olivier Hermant, ancien professeur de mon école d'ingénieur, qui m'a soutenu et conseillé dans ma volonté de faire un doctorat ; Vivien Maisonneuve, doctorant avec qui j'ai partagé mon bureau durant ma première année et qui m'a aidé et conseillé ; Adilla Susungi, doctorante qui a partagé mon bureau jusqu'à la fin de ma thèse et avec qui on a pu plaisanter ; Pierre Guillou, doctorant avec qui j'ai commencé un stage, continué avec un doctorat, et partagé plusieurs aventures (dont je ne suis pas toujours revenu intact ;)) ; Pierre Jouvelot, responsable des doctorants du centre, qui gré mal gré m'a accompagné et m'a permis de faire de belles balades en avion ; Émilio Jesús Gallego et Arnaud Spiwack, post-docs, qui ont pris de leur temps pour m'aider à faire mes preuves de programmes ; et également tous les autres membres du centre, par ordre alphabétique, Fabien Coelho, Laurent Davario, Catherine Le Caer, Claire Medrala, Benoît Pin et Claude Tadonki.

Je souhaite également remercier les doctorants, post-docs et amis que j'ai pu côtoyer et discuter, ou encore avec qui j'ai pu jouer, me détendre et se soutenir : Pierre W., Haishen, Florian, Dounia, Patryk, Bruno, Emmanuelle, Charlotte, Mélanie, Sébastien, Vaïa, JB, Gianni, Luc, Serge, et bien d'autres qui rendraient cette liste bien trop longue.

Enfin, je souhaite remercier ma famille qui, malgré la distance, m'a soutenu et encouragé et sur qui je pourrais toujours compter. Leur présence, pour ma soutenance, témoigne de l'amour et du soutien qui nous lient. Elle a également pu faire découvrir des saveurs polynésiennes lors du pot.

Merci à tous, Māuruuru roa

Sommaire

Liste des figures

Liste des tableaux

Introduction

Contexte

Les ordinateurs modernes ont désormais plusieurs cœurs de calculs par processeur. Ce que l'on appelle couramment les *dualcore*, *quadcore*, *octocore*, *etc.* sont des processeurs possédant respectivement deux, quatre, huit cœurs ou plus. Il est devenu plus simple pour les fabricants de multiplier le nombre de cœurs ou de processeurs dans une machine que d'améliorer les performances d'un processeur en tant que tel [97]. Les développeurs d'applications doivent donc tenir compte de ce nouveau paradigme de programmation s'ils souhaitent profiter pleinement de toutes les ressources mises à leur disposition. Ils doivent développer des applications parallèles.

Les applications parallèles s'exécutent principalement sur deux types d'architectures. Les architectures à mémoire partagée possèdent plusieurs processus qui s'exécutent en partageant le même espace mémoire. Le résultat d'un processus est ainsi accessible directement par les autres processus et un processus peut modifier la valeur d'une variable de n'importe quel autre processus. Ces processus sont également appelés fils d'exécution ou *threads*. Les architectures à mémoire distribuée imposent aux différents processus d'échanger des messages pour pouvoir connaître les valeurs des variables des autres processus. Ils ont chacun un espace mémoire qui leur est dédié. Il existe également d'autres architectures parallèles permettant, par exemple, de modifier à distance une variable, ou encore des architectures hybrides.

Les machines considérées comme les plus puissantes au monde, faisant partie du top500 [102], sont actuellement des machines à mémoire distribuée. La Figure 1 montre l'évolution des architectures des machines du top500 des années 90 à nos jours. On constate que les machines à cœur unique (*single processor* et *SIMD*) disparaissent du top500 dès la fin des années 90. Elles laissent place à des machines massivement parallèles dédiées (*MPP*) et à des machines parallèles à mémoire partagée (*SMP*). En 1994, la machine Beowulf a été créée pour la NASA [95]. Cette machine est constituée de plusieurs ordinateurs connectés entre eux. Elle a permis de démocratiser les grilles de calculs, *cluster*, qui sont des machines à mémoire distribuée. Dans les années 2000, les grilles de calcul se développent pour devenir les principales architectures présentes dans le top500. Les machines classées comme *constellations* sont des grilles de calculs dont la particularité est que chaque ordinateur, appelé nœud, la composant possède "beaucoup" de cœurs de calcul. La notion de "beaucoup" reste vague et deux interprétations existent. La première [15] consiste à dire que chacun des nœuds doit avoir au minimum 16 cœurs. La seconde [31] considère que le nombre de

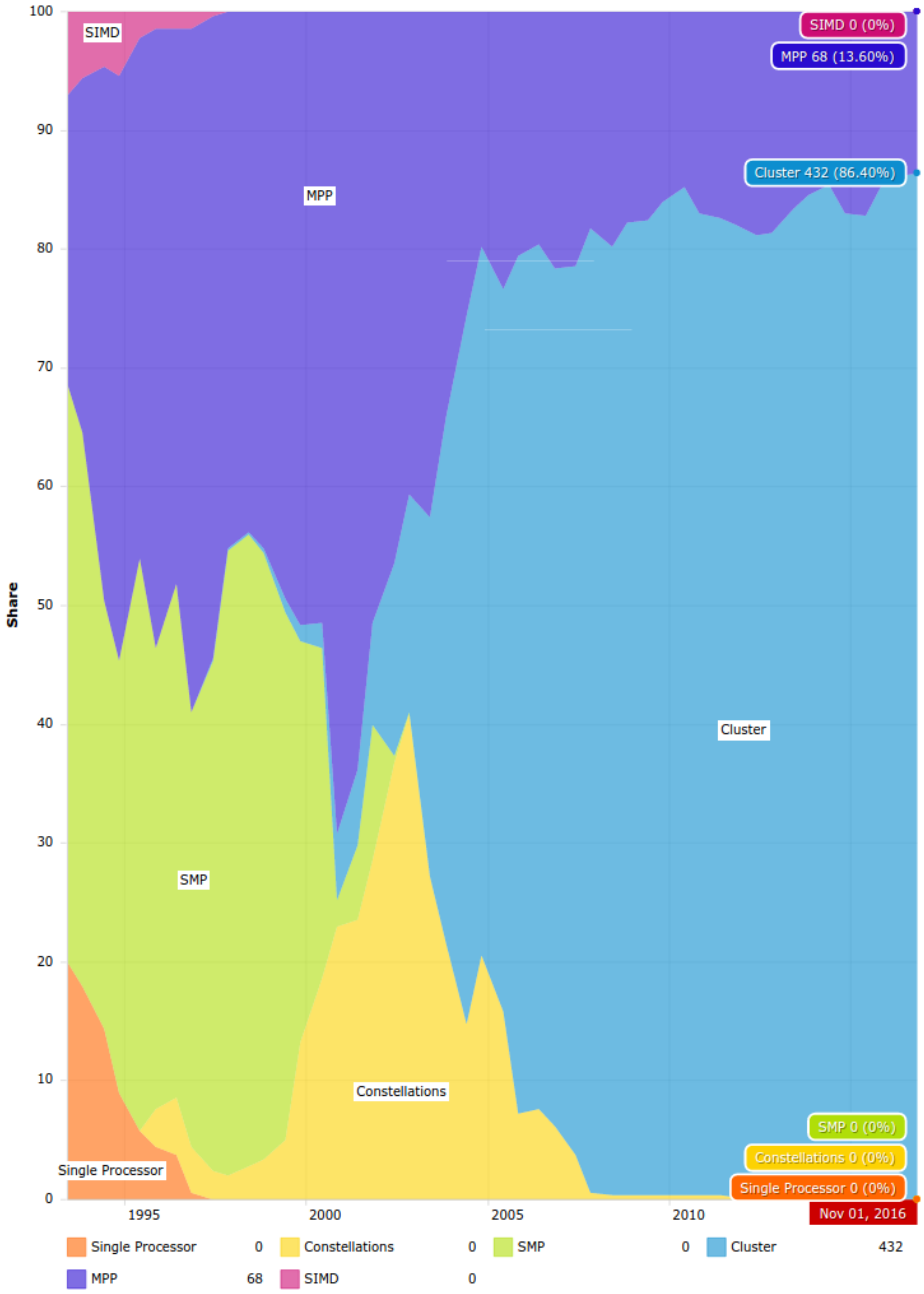


FIGURE 1 – Évolution de la distribution des architectures des machines du top500 au court des années (source top500.org)

cœurs d'un des nœuds doit être supérieur au nombre de nœuds total.

L'évolution des architectures composant le top500 montre l'importance croissante des machines à mémoire distribuée. Il est donc nécessaire de développer des outils permettant de faciliter l'implémentation de programmes sur de telles machines.

Objectif

Contrairement à la programmation sur machine à mémoire partagée, où il existe des compilateurs et outils permettant de générer des codes efficaces appropriés à ces machines, il existe peu de générateurs automatiques de code pour des machines à mémoire distribuée. Je pense notamment à la bibliothèque OpenMP [84] et à son interface de programmation consistant "simplement" à ajouter des directives `pragma` aux sections de code à paralléliser. Du coté des machines à mémoire distribuée, la bibliothèque MPI [71] est le standard. Mais elle oblige le programmeur à gérer la cohérence mémoire et à ajouter des instructions de communication entre les différents processus. Toutes les autres approches par langage ou directives depuis HFP sont restées marginales ou ont échoué.

L'objectif de cette thèse n'est pas de fournir une solution miracle au problème de la programmation des machines parallèles à mémoire distribuée. Plus modestement, nous nous proposons de **faire progresser le processus de compilation** pour de telles machines et **éviter l'approche habituelle consistant à utiliser des *built-ins*** opaques pour gérer la distribution. En effet, ces *built-ins* bloquent les analyses et transformations exécutées par des passes d'optimisations des compilateurs.

Nous avons donc défini **une interface simple** de programmation permettant de distribuer des tâches en utilisant des directives et nous avons conçu **un processus de compilation** constitué d'une succession de petites transformations, originales ou non. Ce processus de compilation permet à partir d'un code séquentiel et d'un ordonnanceur (automatique ou manuel) de tâches de **générer progressivement et automatiquement un code parallèle pour une machine à mémoire distribuée**. Ces transformations sont chacunes suffisamment petites pour pouvoir être **prouvées correctes**.

Organisation du manuscrit

Ce manuscrit est composé de six chapitres.

Le chapitre 1 présente un état de l'art des modèles de programmation parallèle. Les différentes architectures existantes et leur modèle de programmation sont décrits.

Le chapitre 2 discute des différentes problématiques apparaissant lors de la génération automatique de code parallèle distribué pour des tâches. Différentes solutions existantes sont présentées avec leurs avantages et inconvénients. Les solutions envisagées sont argumentées et les hypothèses retenues présentées.

Le chapitre 3 est le cœur de ce manuscrit et présente la solution mise en place. Une description détaillée de notre processus de compilation menant à un

code parallèle distribué pour des tâches est donnée. Elle décrit formellement les différentes étapes de transformations réalisées.

Le chapitre 4 présente les preuves de correction des transformations présentées au chapitre 3. Il permet de montrer que le code généré par le processus de compilation retenu est correct et que le résultat de l'exécution du code généré est identique à celui de l'exécution du code initial.

Le chapitre 5 est une extension du chapitre 3. Il décrit des améliorations et optimisations qui ont été réalisées.

Le chapitre 6 présente les expériences effectuées et les résultats obtenus par la génération automatique de code parallèle distribué pour des tâches.

Enfin, une conclusion résume mes contributions et présente les perspectives possibles.

Chapitre 1

Les modèles de programmation parallèle

Sommaire

La programmation parallèle est devenue une nécessité si l'on souhaite profiter pleinement des capacités qu'offrent les ordinateurs actuels. En effet, aussi bien au niveau d'un cœur de processeur qui va utiliser des instructions parallèles, que d'un processeur qui possède un ensemble de cœurs pouvant faire des calculs simultanément, ou encore des réseaux de communication qui permettent de faire travailler plusieurs processeurs simultanément, la programmation parallèle est présente partout de façon plus ou moins transparente par le programmeur.

Ainsi, la programmation parallèle peut être effectuée de plusieurs façons avec divers modèles et contraintes [14, part 1]. Elle peut s'effectuer au niveau architectural lors de l'exécution d'une instruction, ou au niveau du programme sur un ensemble d'instructions. Ces types de parallélisme sont décrits dans la section 1.1. De plus, plusieurs façons d'accéder à la mémoire sont possibles. Les modèles d'accès à la mémoire partagée, distribuée, ou partitionnée globalement, lorsque des processus s'exécutent sur plusieurs cœurs de processeurs, sont décrits respectivement aux sections 1.2, 1.3 et 1.4. Enfin, un engouement pour la programmation sur accélérateur matériel, notamment les GPGPU, est présent. Ce modèle est présenté en section 1.5.

1.1 Différents paradigmes de parallélisme

La taxonomie de Flynn [36] est une classification des architectures parallèles datant des année 70. Elle porte sur le flux d'instructions (I), exécution du code, et le flux de données (D), flot de données. Ils peuvent être uniques (S, *single*) ou multiples (M). Quatre cas sont possibles :

SISD (*Single Instruction Stream, Single Data Stream*). Une machine "séquentielle" qui exécute une instruction à la fois sur une donnée unique. Bien qu'étant soit disant séquentielle, elle permet dans une moindre mesure de faire du parallélisme. En effet, les instructions pipelinées sont considérées comme faisant partie de cette catégorie. Elle correspond à une machine de type Von Neumann [74]. Ces machines standards des années 70 sont rares de nos jours. La Figure 1.1(a) représente une architecture SISD.

SIMD (*Single Instruction Stream, Multiple Data Streams*). Une machine où chaque instruction peut traiter plusieurs données simultanément. Les machines vectorielles en sont la parfaite illustration. Les applications utilisant beaucoup de tableaux ou de matrices, par exemple le traitement de signal ou d'image, tirent pleinement partie d'une telle architecture. De nos jours, quasiment toutes les machines possèdent un processeur vectoriel avec des jeux d'instructions vectorielles telles que le MMX, SSE ou encore AVX. La section 1.1.1 donne de plus amples explications. La Figure 1.1(b) représente une architecture SIMD.

MISD (*Multiple Instruction Streams, Single Data Stream*). Une machine qui, pour une donnée, exécute simultanément plusieurs instructions. Elles sont souvent spécifiques à un domaine d'application. Les machines systoliques font partie de cette classe. La Figure 1.1(c) représente une architecture MISD.

MIMD (*Multiple Instruction Streams, Multiple Data Streams*). Une machine qui exécute simultanément plusieurs instructions sur plusieurs données différentes. C'est le type de machines parallèles le plus répandu. Par exemple, les architectures *Very Long Instruction Word* VLIW permettent d'exécuter plusieurs instructions différentes sur plusieurs données différentes. VLIW a inspiré l'architecture EPIC, Explicitly Parallel Instruction Computing, qui est décrite à la section 1.1.2. La Figure 1.1(d) représente une architecture MIMD. Le modèle MIMD est souvent décomposé en divers sous-catégories pouvant ou non se superposer. Par exemple, une

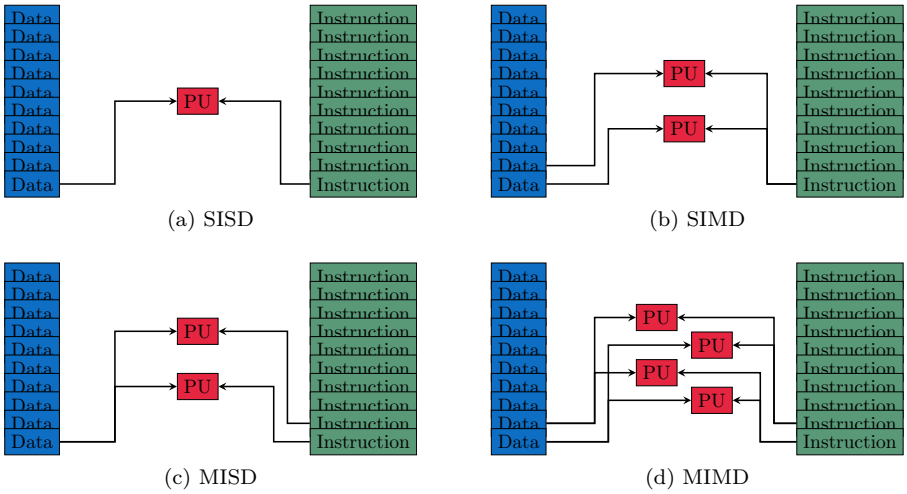


FIGURE 1.1 – Taxonomie de Flynn

sous-catégorie est la séparation entre le SPMD (*Single Program Multiple Data*) et le MPMD (*Multiple Program Multiple Data*), une autre est la séparation entre mémoire partagée, distribuée et hybride dont nous parlerons par la suite. Cette catégorie de machines tend à devenir la machine standard avec des processeurs possédant 2, 4, 8 voire 16 cœurs.

Concernant la catégorisation des programmes parallèles, les modèles SPMD et MPMD, sous-catégories du MIMD, sont souvent utilisés. Le SPMD regroupe principalement les programmes scientifiques alors que le MPMD correspond plutôt à des programmes clients-serveurs que l'on trouve sur internet.

Cette sous-classification ne permet cependant pas de décrire le type de parallélisation que le compilateur ou le programmeur effectue lorsque plusieurs processeurs sont disponibles. On distingue deux types de parallélisation : la parallélisation de boucles, détaillée en section 1.1.3, et la parallélisation de tâches, présentée en section 1.1.4.

1.1.1 Parallélisme de données

```
for (i=0; i<=8; i++) {
  c[i] = a[i] + b[i];
}
```

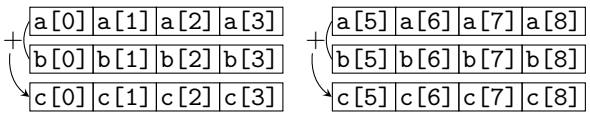


FIGURE 1.2 – Exemple de vectorisation

Le parallélisme de données est principalement exploitée lors de la vectorisation. La Figure 1.2 illustre le principe de fonctionnement d'un processeur pouvant exécuter des instructions vectorielles. Elles permettent de réaliser un calcul sur un ensemble de données, souvent un multiple de 2, 4 ou 8. Dans la Figure 1.2, le calcul des éléments du tableau `c` s'effectue en deux instructions vectorielles d'addition traitant simultanément quatre éléments du tableau. Sans

instruction vectorielle, il aurait fallu exécuter huit instructions pour obtenir le même résultat, soit quatre fois plus de temps d'exécution.

Les premiers processeurs vectoriels sont apparus au début des années 1970. Ils étaient intégrés dans les supercalculateurs de l'époque et coûtaient relativement chers. Ils étaient peu accessibles au grand public. La plupart des machines Cray, Cray-1, Cray X-MP ou Cray-2, possédaient des processeurs vectoriels. Puis, les processeurs vectoriels ont commencé à se démocratiser et différents jeux d'instructions SIMD sont apparus. En 1997, Intel développe le jeu d'instructions MMX [70][50, chapitre 9]. Il sera concurrencé par celui d'AMD avec 3DNow [10]. Plusieurs nouveaux jeux d'instructions SIMD suivront avec les SSE [98] et tous ses descendants SSE2, SSE3, et SSE4 [50, chapitres 10 à 12]. En 2008, la première version d'AVX [50, chapitre 14] sort et sera étendue avec AVX2 et AVX-512 [51][50, chapitre 15].

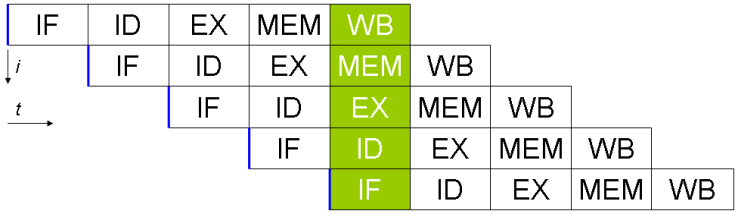
Une variante plus efficace de la vectorisation est la parallélisation SLP, *Superword Level Parallelism* [63]. Comme la vectorisation, le SLP utilise les différents jeux d'instructions SIMD. Mais tandis que la vectorisation classique ne s'applique que si toutes les itérations d'une boucle et chacune de ses instructions sont indépendantes les une des autres, SLP permet aussi de paralléliser les diverses instructions qui composent la boucle. En plus d'être plus efficace, la chaîne de compilation utilisée est souvent plus simple à mettre en place et s'applique sur un plus large panel de programmes. Elle est de nos jours activée par défaut lors d'une compilation en -O3 avec gcc [38].

1.1.2 Parallélisme d'instructions

Le parallélisme d'instructions, *Instruction-Level Parallelism (ILP)*, est un parallélisme de bas niveau dans un processeur.

Je présente sommairement la technique de pipeline et les instructions de type VLIW. Plusieurs autres techniques associées existent comme les exécutions spéculatives ou les prédictions de branchement [45, Chapitre 3].

(a) Sans pipline



(b) Avec pipline

FIGURE 1.3 – Exemple de processeur avec et sans pipeline (source wikipedia)

Le principe du pipeline est simple. Il consiste à considérer les différentes étapes d'exécution d'une instruction. Prenons l'exemple des cinq étapes suivantes :

IF (*Instruction Fetch*) met l'instruction en mémoire,

ID (*Instruction Decode*) décode l'instruction et récupère les registres nécessaires pour l'exécution,

EX (*Execute*) exécute l'instruction sur l'unité arithmétique et logique correspondante,

MEM (*Memory*) effectue un accès mémoire, de la mémoire vers un registre pour une lecture *LOAD*, ou du registre vers la mémoire pour une écriture *STORE*,

WB (*Write Back*) écrit le résultat dans un registre.

Ainsi, lorsque la première étape de l'instruction *IF* est terminée, l'instruction suivante peut commencer à s'exécuter en initiant une nouvelle étape *IF*, en parallèle de l'exécution du *ID* de l'instruction précédente. La Figure 1.3 permet d'illustrer une exécution pipelinée. On représente les instructions avec les différentes étapes d'exécution en ordonnée, et les cycles d'exécution en abscisse. (a) montre que sans pipeline, il faudrait 15 cycles pour pouvoir exécuter 3 instructions, alors qu'avec un pipeline (b), seulement 9 cycles sont nécessaires pour exécuter 5 instructions, et au cinquième cycle les cinq étapes permettant d'exécuter une instruction sont au travail.

En réalité, plusieurs problèmes peuvent se poser. Par exemple, toutes ces étapes ne s'exécutent pas forcément en un même nombre de cycles horloge, ou encore toutes les instructions ne nécessitent pas forcément toutes ces étapes. Ainsi, toutes les étapes du pipeline ne sont pas forcément toutes occupées pour cause de latence ou simplement car elles ont un statut inoccupé, *idle*, le temps que l'étape qui la précède ait fini son travail. [45, Annexe C] présente plus en détails toutes ces problématiques.

Les architectures de type VLIW, *Very Long Instruction Word* permettent d'exécuter plusieurs instructions simultanément. Le VLIW a été introduit par Fisher [35][34] au début des années 1980. Il a été repris par Intel et HP pour créer l'architecture EPIC, *Explicitly Parallel Instruction Computing* [94]. L'exécution parallèle d'instructions est possible en précisant explicitement les instructions assembleur sur une même ligne d'exécution. EPIC est une extension du VLIW car il permet également de définir des instructions conditionnelles, prédictives, mélangeant instructions flottantes et entières ou encore modifiant les accès registres lors d'une même ligne d'exécution. Dans un certain sens, les EPIC permettent de déplacer des optimisations qui sont faites par le matériel, comme la prédiction de branchement, au niveau du compilateur. Les processeurs Itanium d'Intel sont les principaux représentants des architectures EPIC. Pour plus de détails, [45, Annexe H] décrit les optimisations qui peuvent être faites pour générer et utiliser au mieux les VLIW et EPIC.

1.1.3 Parallélisme de boucles

Lorsque l'on souhaite faire de la parallélisation à plus gros grain, pour utiliser plusieurs processus, on réalise une parallélisation au niveau des boucles.

La parallélisation de boucle est possible lorsque les différentes itérations de la boucle sont indépendantes. Des itérations totalement indépendantes permettent de répartir chaque itération sur des processus différents. Elles pourront s'exécuter simultanément. Le parallélisme de boucles permet souvent d'exploiter du parallélisme de données.

Dans les autres cas, plusieurs analyses et transformations peuvent être appliquées [9][109]. On peut chercher si un motif de dépendance ou un stencil apparaît entre les différentes itérations de la boucle. Si tel est le cas, la réalisation d'un pavage, *loop tilling* [52][108], permet en général de mettre en évidence au moins une boucle parallèle. Par exemple, prenons le cas d'un tableau bidimensionnel A, si pour chaque itération (i, j) le calcul de $A[i][j]$ a besoin de $A[i-1][j-1]$, alors il est possible de réorganiser les itérations de boucles pour effectuer un parcours en diagonal, de direction $(1, 1)$, plutôt qu'un traditionnel parcours en ligne/colonne. Cette transformation permet d'exécuter en parallèle toutes les itérations se trouvant sur la diagonale $(-1, -1)$.

Dans tous les cas, on cherche à conserver le maximum de localité spatiale lorsque les différentes itérations s'exécutent sur des processus différents, et ce afin d'utiliser au mieux les caches des processeurs. Les éléments contigus d'un tableau devront être accédés ou modifiés par un même processus lorsque cela est possible. De nombreuses analyses polyédriques [33] ont été développées pour chercher les meilleurs stencils possibles [22] et pour paralléliser et optimiser les nids de boucles [23][16].

1.1.4 Parallélisme de tâches

La parallélisation de tâches vise également l'optimisation de programmes s'exécutant sur plusieurs processus. En théorie, son fonctionnement est très simple. Diverses tâches sont présentes et s'exécutent sur plusieurs processus simultanément. Des dépendances entre tâches peuvent être présentes. Elles imposent des contraintes sur les ordonnancements possibles de ces tâches.

En pratique, les problèmes liés à la parallélisation de tâches ou à la génération de tâches parallèles sont nombreux. L'objectif principal est d'avoir le moins de dépendances possibles entre tâches, tout en conservant des tâches de "tailles" raisonnables. En effet, si toutes les tâches sont dépendantes les unes des autres, alors aucun parallélisme n'est possible et chaque tâche doit attendre que les précédentes aient fini, avant de pouvoir s'exécuter. De plus, un grand nombre de tâches indépendantes permet d'avoir beaucoup de parallélisme. Ainsi, un maximum de tâches indépendantes doit être présent pour avoir le plus de parallélisme possible. Enfin, il faut réussir à trouver la bonne charge de travail pour chaque tâche car il y a un coût de lancement et de récupération des informations nécessaires à l'exécution d'une tâche. Par exemple, il est préférable d'avoir une tâche qui exécute un ensemble d'instructions pour calculer un élément d'un tableau, plutôt qu'un ensemble de tâches n'exécutant qu'une seule instruction. De même, il faut faire attention à l'équilibrage de charge entre les différentes tâches. Si une tâche s'exécute trop rapidement par rapport à une autre tâche, qui s'exécute simultanément, alors le processus qui gère cette tâche doit attendre un long moment avant de pouvoir faire autre chose. C'est pour cette raison que les tâches les plus lentes sont importantes pour connaître les améliorations possibles des performances.

La parallélisation de boucles peut être vue comme un cas particulier de la

parallélisation de tâches. Chaque itération ou groupe d'itérations s'exécutant sur un processus peut être vue comme une tâche.

1.2 Le modèle à mémoire partagée

Les machines à mémoire partagée sont classées comme MIMD dans la taxonomie de Flynn. Comme son nom l'indique, elles partagent une mémoire unique entre différents processus. La Figure 1.4 illustre une architecture à mémoire partagée où plusieurs processus partagent le même espace d'adressage mémoire. Ces processus correspondent souvent aux différents cœurs d'un processeur.

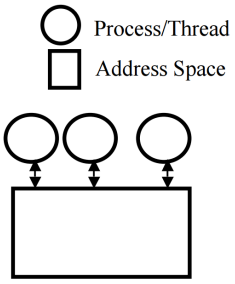


FIGURE 1.4 – Modèle à mémoire partagée

Dans le cadre de programmes parallèles, l'avantage de ce type de machines est que tous les processus ont accès à toutes les valeurs des variables présentes dans le programme. L'inconvénient est qu'il faut faire attention à la cohérence des accès mémoire entre les différents processus. En effet, si plusieurs processus peuvent lire simultanément une variable, sans créer d'incohérence, plusieurs processus ne doivent pas modifier simultanément une même variable, sinon un indéterminisme survient et la valeur de la variable pourra aussi bien valoir la valeur affectée par l'un ou l'autre des processus. De même, si un processus modifie une variable, les autres processus doivent attendre la fin de l'écriture en mémoire avant de récupérer la nouvelle valeur. Ce problème est un problème d'accès concurrent, *data race*, pour les cas de modifications simultanées non protégées, ou un problème de course critique, *race condition*, dans le cas où l'état final devient indéterminé [73][92].

Toujours dans le cadre parallèle, les performances des accès aux variables sont identiques à ceux d'un accès en séquentiel, avec tous les avantages que peuvent offrir la mise en cache des variables pour un accès plus rapide. Par contre, des problèmes de cohérence de caches peuvent apparaître dans un code mal conçu. Dès qu'un processus modifie une variable, ou un élément de tableau, cette modification entraîne une invalidation de la ligne de cache qui contient cette variable, ou élément de tableau, pour tous les autres processus. Ainsi, si deux processus modifient simultanément et respectivement les éléments pairs et impairs d'un même tableau, le code sera parfaitement correct, mais les performances seront médiocres car une succession d'invalidations de leur ligne de cache s'effectuera et obligera à faire des accès mémoire plus coûteux régulièrement.

Plusieurs normes et bibliothèques permettant de faire de la parallélisation à mémoire partagée existent. Parmi elles, les plus répandues sont POSIX, section 1.2.1, et OpenMP, section 1.2.2.

1.2.1 POSIX

POSIX, *Portable Operating System Interface*, est un standard IEEE, *IEEE 1003* qui comporte plusieurs versions. Il est aussi connu sous la référence *ISO/IEC 9945* [53]. Il a entre autres standardisé l'utilisation de `thread`, aussi appelé "fil d'exécution" ou "processus léger". On parle de POSIX threads, ou plus couramment `pthreads`.

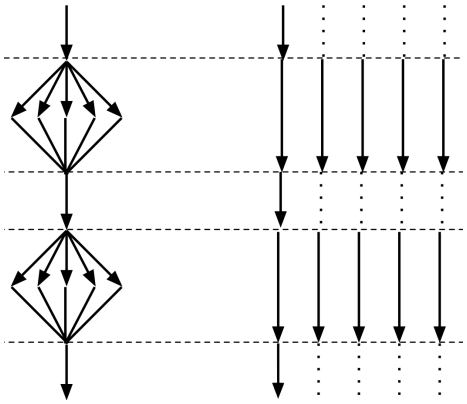


FIGURE 1.5 – Exécution avec le paradigme *fork/join*

POSIX utilise principalement le paradigme *fork/join*, Figure 1.5. Ce paradigme crée plusieurs processus légers, au moyen de la fonction `pthread_create`, qui vont s'exécuter simultanément. Ces processus attendent ensuite que tous les `pthreads` finissent leur exécution pour se rejoindre lors de l'appel à la fonction `pthread_join`. Ces deux fonctions s'appliquent sur des variables de type `pthread_t`.

Pour gérer les problèmes d'accès concurrents, on peut utiliser des `mutex`, pour *mutual exclusion* ou exclusion mutuelle. Le principe consiste à mettre en place un verrou sur l'exécution d'une partie d'un code, pour que celle-ci ne puisse être exécutée que par un seul et unique `thread` en même temps. Cette partie de code doit donc être la plus courte possible pour garder le maximum de parallélisme. Les exclusions mutuelles sont principalement utilisées lors de la mise à jour d'une variable qui pourrait être modifiée par plusieurs `thread`. La bibliothèque POSIX utilise les fonctions `pthread_mutex_lock` et `pthread_mutex_unlock` sur des variables de type `pthread_mutex_t`.

POSIX est une bibliothèque relativement lourde à utiliser car le programmeur doit tout gérer, aussi bien la création que l'affectation ou la fermeture des `thread`. C'est une bibliothèque très bas niveau qui est peu utilisée dans les développements effectifs de programmes car elle demande beaucoup d'effort aussi bien pour adapter le code que pour obtenir des résultats et des performances corrects.

1.2.2 OpenMP

OpenMP [84][85] est une bibliothèque et une interface de programmation pour la parallélisation de programmes écrits en C, C++ et Fortran, principale-

ment orientée pour des architectures à mémoire partagée. Elle est gérée par une organisation à but non lucratif l'*OpenMP Architecture Review Board*, ou plus simplement l'ARB, qui est composée de plusieurs grandes sociétés fabriquant du matériel informatique ou éditant des logiciels. Sa première spécification date de 1997 pour Fortran et 1998 pour C.

OpenMP se base sur des directives *pragma* en C ou C++ et sur des commentaires en Fortran. De ce fait, OpenMP est portable sur différentes machines, car les directives peuvent tout simplement être ignorées lors de la compilation si le compilateur ou la machine cible d'exécution ne les supporte pas. Ainsi, au pire des cas, un code séquentiel pourra toujours être généré. Le grand nombre d'acteurs dans l'ARB fait que ce dernier cas n'arrive que rarement.

Les modifications à faire sur le code se veulent simples de prime abord. Un simple ajout de `#pragma omp parallel for` peut suffire pour paralléliser une boucle. OpenMP offre principalement deux paradigmes de parallélisation.

Le premier est le traditionnel *fork/join*. Il est plus simple à mettre en place que son équivalent `pthread`. Il est utilisé pour paralléliser les boucles.

Ainsi, on utilise `#pragma omp parallel` sur une séquence d'instructions. L'ouverture du *scope*, *ie.* le début de la séquence, initie plusieurs `thread` à exécuter (*fork*) ; et la fermeture du *scope*, *ie.* la fin de la séquence, ferme tous les `thread` lancés (*join*).

Puis dans la séquence d'instructions, la parallélisation de boucle `for` s'effectue avec `#pragma omp for`. Cette directive se charge de répartir les différentes itérations entre les différents `thread` lancés précédemment. Toutes les itérations doivent avoir été exécutées et chaque `thread` attend que tous les autres aient terminé pour continuer l'exécution.

Plusieurs paramètres peuvent être ajoutés à ces directives de base. Par exemple, pour indiquer le nombre de `thread` à exécuter (`num_threads`), pour contrôler l'ordonnancement lors de l'exécution des boucles (`schedule`), ou pour indiquer qu'il n'y a pas besoin d'attendre la fin de l'exécution des autres `thread` pour continuer (`nowait`).

Le deuxième paradigme consiste à faire du parallélisme de tâches. Ce parallélisme est possible depuis la version 3.0 d'OpenMP, avec l'introduction de la directive `#pragma omp task`.

La gestion des tâches par OpenMP se fait au moyen d'un pool de tâches, *task pool*. Ainsi, lorsqu'un `thread` crée une tâche, elle est mise dans le pool de tâches. Si un `thread` ne fait rien, il va chercher, dans ce pool de tâches, une tâche à exécuter. Les tâches ne sont pas à priori attribuées à un `thread` particulier mais récupérées dynamiquement par le premier `thread` disponible.

À ces deux paradigmes s'ajoutent divers directives et paramètres. Certaines directives servent à la création de point de synchronisation tel que `#pragma omp barrier` ou `#pragma omp taskwait`, ou à définir des chemins critiques pour éviter les accès concurrents, `#pragma omp critical`, `#pragma omp atomic`. La définition de la localité des données est également possible grâce aux paramètres `shared`, `private` et ses dérivées, `firstprivate` et `lastprivate`.

OpenMP est sans doute la bibliothèque la plus populaire et la plus utilisée pour réaliser de la parallélisation pour des architectures à mémoire partagé. Elle est à la fois simple d'utilisation, car elle ne requiert que peu de modifications du code source d'origine, et portable sur un grand nombre de plate-formes. Toutefois, cette simplicité apparente peut conduire à des codes inefficaces, voir faux, sans une bonne analyse des dépendances et des accès mémoires du code source.

1.3 Le modèle à mémoire distribuée

Les machines à mémoire distribuée sont également classées comme MIMD dans la taxonomie de Flynn. Contrairement aux machines à mémoire partagée, tous les processeurs et donc tous les processus n'ont pas un accès direct à la mémoire des autres processeurs. Ils doivent communiquer les données entre eux s'ils veulent connaître ou mettre à jour des données distantes. La Figure 1.6 illustre une architecture à mémoire distribuée où chaque processus possède son propre espace d'adressage mémoire. Si un processus a besoin de la valeur d'une variable se trouvant sur un autre espace mémoire, il devra en faire la requête auprès du processus qui le possède.

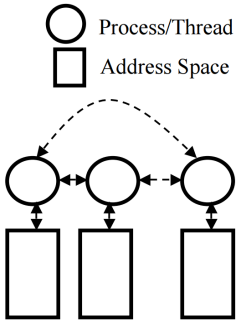


FIGURE 1.6 – Modèle à mémoire distribuée

L'avantage est qu'il n'y a forcément pas d'accès concurrents à la mémoire, puisque chacun possède sa propre mémoire. De plus, une mise à l'échelle sur un très grand nombre de processeurs est possible contrairement au modèle à mémoire partagée qui est limité au nombre de cœurs et de processeurs pouvant être présents sur une puce. L'inconvénient est qu'il faut effectuer des communications entre les différents processus pour mettre à jour les valeurs des différentes variables sur les différents processus. Si un processus n'a pas la valeur d'une variable ou si la valeur n'est pas à jour, un résultat erroné est produit.

Les performances obtenues dépendent fortement du nombre de communications effectuées lors de l'exécution du programme et du ratio entre le temps de communications et les calculs à effectuer. Les communications entraînent toujours un surcoût lors de l'exécution d'un programme. Ainsi, il doit être compensé par un temps de calcul en parallèle plus important. De même, l'initiation d'une communication est coûteuse. Il faut donc les limiter quand cela est possible. Il est donc préférable de communiquer toute une partie d'un tableau plutôt que de communiquer élément par élément. Il est parfois également préférable de réunir

les variables à envoyer dans un tampon, *buffer*, et de faire une communication plutôt que plusieurs communications.

Il y a deux catégories de communication :

point à point , une communication entre deux processus, l'un émetteur, l'autre receveur. Ces communications doivent généralement préciser l'identifiant du processus émetteur et celui du processus récepteur.

collective , une communication dont le message est envoyé à un groupe de processus. Trois sous-catégories de communication collective existent : la synchronisation, comme une barrière ; la communication, comme une diffusion, *broadcast* ; et le calcul, comme une réduction.

Nous présentons la bibliothèque PVM en section 1.3.1, le standard MPI en section 1.3.2, et HPF en section 1.3.3.

Les bibliothèques de programmation et de parallélisation pour les architectures à mémoire distribuée sont souvent couplées à des bibliothèques de parallélisation pour mémoire partagée, afin de pouvoir tirer partie au mieux des architectures variées sur lesquelles tournent les programmes. Ainsi, il n'est pas rare de voir du code utilisant à la fois MPI pour les communications entre machines ou nœuds et OpenMP pour la répartition des calculs à exécuter au sein d'une machine ou d'un nœud.

1.3.1 PVM

PVM, *Parallel Virtual Machine* [91], est une bibliothèque de parallélisation des calculs pour des machines hétérogènes, aussi bien pour des ordinateurs standards que pour des super-calculateurs. Le projet a été initié en 1989 par l'*Oak Ridge National Laboratory* (ORNL) aux USA [39][96]. Sa dernière version est la version 3.4.6 publiée en 2009. PVM possède une implémentation pour C, C++ et Fortran.

PVM fonctionne en mode maître/esclaves. Ainsi, il doit toujours y avoir un des programmes qui est le maître. Son fonctionnement implique que différents programmes puissent tourner simultanément sur différentes machines. PVM se classifie donc comme MPMD.

Pour fonctionner, des *daemon* PVM (`pvmd`) doivent tourner sur toutes les machines exécutant des programmes PVM. Ces *daemon* sont chargés d'effectuer les communications et le routage entre les différentes machines. La Figure 1.7 illustre le fonctionnement d'un tel système.

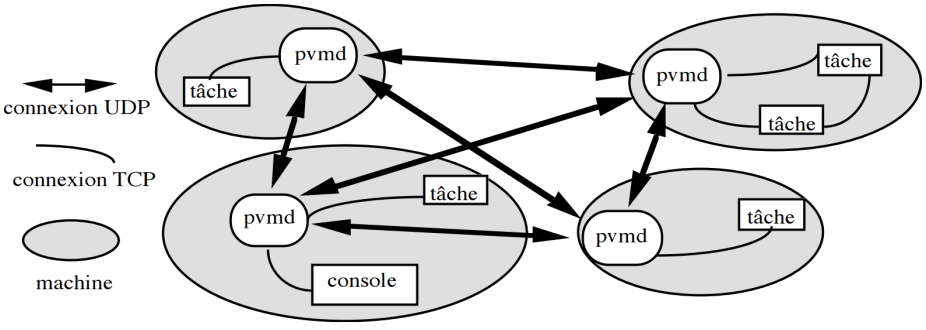


FIGURE 1.7 – Schéma des modes de connexion du système PVM (source [93])

Le programme maître en PVM commence toujours par un appel à `pvm_tid`. Tous les programmes PVM finissent par un appel à `pvm_exit`. Le maître peut lancer plusieurs esclaves exécutant une tâche grâce à la fonction `pvm_spawn`. Ces derniers connaissent l'identité de leur maître avec la fonction `pvm_parent`.

Les messages envoyés sont toujours non bloquants et utilisent un tampon, *buffer*, à initialiser (`pvm_initsend`), puis à remplir (`pvm_pkint`) avant de faire l'envoi (`pvm_send`). La réception peut être bloquante ou non. Elle se fait au moyen de la fonction `pvm_recv` et les données doivent être extraites avec `pvm_upkint` dans le même ordre que le remplissage.

1.3.2 MPI

Le standard MPI, pour *Message Passing Interface* [71], est apparu en 1992 dans un souhait d'homogénéiser la façon de communiquer des messages entre différents processus. Il est développé aussi bien par des académies que des industriels. Plusieurs implémentations existent, parmi elles, les plus connues sont MPICH [72] et OpenMPI [81].

Le mode de fonctionnement de MPI, par exemple maître/esclaves ou pair à pair, *peer-to-peer*, est défini dans le programme lui-même. MPI se classifie parmi les SPMD. Ainsi, pour savoir quelle partie du programme un processus doit exécuter, un test au sein du programme est effectué.

Tous les programmes MPI commencent avec un appel à `MPI_Init`, qui permet d'initier l'environnement MPI, et finissent avec `MPI_Finalize` pour le clôturer. Un appel à `MPI_Comm_rank` permet de savoir sur quel processus on se trouve. Cette information est indispensable pour affecter une portion de code à un processus. L'appel à `MPI_Comm_size` permet de connaître sur combien de machines le programme a été lancé. Il a deux usages principaux : le premier est d'assurer qu'il y a bien le nombre minimum de processus lancés requis pour l'exécution du programme ; le deuxième peut servir, lors de la parallélisation de boucle, à répartir les itérations de boucles sur chaque processus.

Les communications point à point s'effectuent avec les fonctions bloquantes `MPI_Send` et `MPI_Recv` ou non-bloquantes `MPI_Isend` et `MPI_Irecv`. D'autres variantes existent pour envoyer des messages asynchrones, au moyen d'un tampon, ou si la réception est en attente avant même que l'envoi ne soit fait. Les communications collectives se font principalement au moyen des fonctions `MPI_Bcast`, ou `MPI_Ibcast`. Pour les fonctions non-bloquantes, une variable de type `MPI_Request` permet de savoir si l'envoi et/ou la réception sont finis. Une barrière, permettant d'attendre un ensemble de processus, s'implémente avec `MPI_Barrier`.

De nombreux autres fonctions MPI existent, un peu moins de 400 en tout, et permettent une utilisation plus avancée. Par exemple, d'autres fonctions de communications collectives existent pour envoyer ou recevoir seulement des portions de tableaux ; faire une cartographie des différents processus ; définir des sous-ensembles de processus pouvant communiquer entre eux ; ou personnaliser des types de variables à communiquer.

1.3.3 HPF

Le standard HPF, *High Performance Fortran* [48][62], date de 1993. Il est né de la volonté de normaliser les différentes syntaxes de parallélisme de données

utilisées et basées sur Fortran, à savoir Fortran D [47], Vienna Fortran [25] et CM Fortran [99].

Le parallélisme cible principalement le parallélisme de boucle. Contrairement à PVM ou MPI, HPF a des communications totalement implicites, ce qui en fait un langage simple à utiliser. La répartition des calculs sur les différents processus se fait selon le principe du "*owner-computes*", ou plus précisément du "*left-hand-side owner-computes*", c'est-à-dire que le processus qui fait le calcul est celui qui détient la donnée à modifier. Un ensemble de directives compose le standard HPF et permet entre autre de définir qui détient une donnée.

Les directives principales sont `!HPF$ DISTRIBUTE`, qui permet de définir une distribution par bloc ou cycle avec une taille de bloc pour chaque cycle donné, et `!HPF$ ALIGN`, qui permet d'aligner différents tableaux sur une même distribution. Un grand nombre d'intrinsèques est également défini. Ils permettent aussi bien d'avoir des informations sur l'environnement, par exemple le nombre de processus lancés, que d'être fonctionnel, par exemple effectuer des réductions.

Malgré son manque de succès, lié à de mauvaises performances et des performances très hétérogènes entre différentes machines [55], plusieurs idées d'HPF ont été reprises, autant dans la définition des nouvelles normes de Fortran, que pour la définition de certaines directives et intrinsèques de la bibliothèque OpenMP, ou encore dans de nouveaux langages sur le modèle PGAS.

1.4 Le modèle PGAS

Le modèle PGAS, *Partitioned Global Address Space*, permet de simuler une mémoire partagée sur une machine à mémoire distribuée. Il définit un espace d'adressage global pour tout le programme, mais les données peuvent être distantes. Le modèle PGAS permet notamment de programmer des architectures à mémoire partagée distribuée, DSM, *Distributed Shared Memory*. La Figure 1.8 illustre une telle architecture où l'espace d'adressage global peut être distribué sur des mémoires physiques différentes.

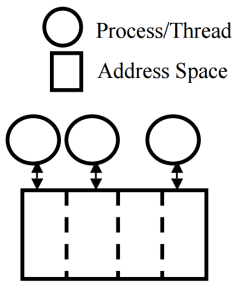


FIGURE 1.8 – Modèle à mémoire partagée distribuée

Le modèle PGAS cherche à tirer partie des avantages du modèle à mémoire partagée et du modèle à mémoire distribuée. Ainsi, il veut pouvoir s'exécuter sur un très grand nombre de processus tout en gardant un paradigme de programmation à mémoire partagée. La cohérence mémoire devra être assurée par le programmeur, alors que les communications des données sont automatiquement gérées par le programme lors de sa compilation ou lors de son exécution.

Parmi les langages PGAS existants nous présentons UPC en section 1.4.1, et CAF en section 1.4.2 [26] qui sont respectivement des extensions des langages C et Fortran.

1.4.1 UPC

Le langage UPC, *Unified Parallel C* [105][106], est une extension du C99 qui permet d'exprimer explicitement le parallélisme. La première spécification de ce langage date de 2001. UPC utilise le paradigme de programmation SPMD. Le nombre de processus à lancer peut être déterminé à la compilation ou à l'exécution.

Étant un langage PGAS, UPC a des communications implicites et les synchronisations doivent être effectuées explicitement lorsque c'est nécessaire, au moyen de fonctions comme `upc_barrier`, `upc_wait/notify` ou `upc_fence`. Des verrous, `upc_lock/unlock`, peuvent également être utilisés pour définir les zones critiques.

Les variables devant être réparties sur les différents processus doivent être déclarées explicitement en tant que variables partagées au moyen du mot clef `shared`. La répartition par défaut est cyclique avec une taille de bloc unitaire. `shared[block-size]` permet de modifier cette taille.

Les boucles parallèles sont précisées au moyen de `upc_forall`. Ses trois premiers arguments sont identiques à ceux d'une boucle `for` en C, et un quatrième argument est une expression, de valeur entière, qui précise les processus exécutant les différentes itérations de boucles.

Comme avec MPI, on peut savoir quel processus est en train de s'exécuter et combien de processus ont été lancés en récupérant respectivement les constantes globales `MYTHREAD` et `THREAD`. Par contre, utiliser ces constantes pour réaliser du parallélisme de tâche peut se révéler très inefficace. En effet, les accès mémoire pour des variables définies `shared` sont très coûteux, même lorsqu'elles sont locales au processus. Un accès mémoire à une variable `shared` locale à un processus peut être deux fois plus lente qu'un accès à une variable privée du processus.

1.4.2 CAF

Le langage CAF, *Co-Array Fortran* [75], est une extension du langage Fortran 95. Il a été spécifié en 1998. CAF 1.0 a été inclus dans la norme Fortran 2008 [76], avec l'ajout de fonctions permettant entre autres de faire des communication globales. Une version 2.0 est développée de son coté par l'université de Rice [67][7]. CAF utilise un paradigme de programmation SPMD. La terminologie utilisée par CAF pour chaque exécution du programme est image.

Les synchronisations s'effectuent avec les fonctions `sync_all` ou `sync_team`. Une liste d'images peut être définie pour donner les processus impliqués dans des fonctions de synchronisation.

Les *co-array* sont des tableaux ou des scalaires qui peuvent être accédés à distance. Ils sont définis par des *co-dimensions* spécifiant leurs déclarations sur les différents processus. Alors que les dimensions des tableaux classiques utilisent des parenthèses `(.)`, les *co-dimensions* utilisent des crochets `[.]`. Ainsi, tous les processus ont les mêmes tailles de tableaux.

Pour savoir quel processus est en cours d'exécution et quel est le nombre de processus exécutant le programme, les fonctions `this_image` et `num_images` peuvent être utilisées. Par défaut, l'indice des images commence à 1.

1.5 Les GPGPU et les accélérateurs

Enfin, un dernier modèle de programmation parallèle consiste à déporter les calculs sur des accélérateurs matériels tels que notamment les GPGPU. Pour ce faire, il est souvent nécessaire (1) d'écrire les noyaux de calculs dans un langage spécifique aux GPGPU, tel que CUDA, décrit en section 1.5.1, ou OpenCL, décrit en section 1.5.2, et (2) de gérer les données à envoyer pour effectuer les calculs et les résultats à recevoir. Certaines bibliothèques permettent de générer automatiquement ces noyaux de calcul et de communications, tel que OpenACC présentée en section 1.5.3, ou OpenMP présentée en section 1.5.4.

De large quantité de données doivent être traitées de façon similaire pour que ce modèle fonctionne et apporte des performances.

1.5.1 CUDA

Le langage CUDA, *Compute Unified Device Architecture* [77], est développé par NVidia. Il est actuellement le leader pour la programmation sur GPGPU. Mais, il est propriétaire et ne peut s'exécuter que sur des GPU de NVidia.

CUDA possède un compilateur spécifique pour ces programmes, `nvcc`. De plus, de nombreuses bibliothèques de programmation très performantes existent dont notamment cuBLAS [79] pour l'algèbre linéaire et cuSPARSE [80] pour les opérations sur les vecteurs et matrices creuses. De même, plusieurs outils de débogage et de profilage sont fournis par NVidia pour aider les programmeurs à optimiser leur code [78]. CUDA utilise la terminologie de grille pour parler d'une exécution sur le GPGPU. Une grille est composée d'une mémoire qui est appelée mémoire globale et d'un ensemble de blocs. Un bloc est lui-même composé d'une mémoire appelée mémoire partagée et d'un ensemble de threads.

Les fonctions CUDA s'écrivent comme des fonctions C classiques. Pour différentier l'architecture sur laquelle s'exécutent les fonctions, des qualificatifs peuvent être ajoutés à leur déclaration. `__global__` indique que la fonction est exécutée sur le GPGPU, le *device*, mais doit être appelée depuis l'hôte, l'*host*. Il correspond au noyau, *kernel*, a exécuté sur le GPGPU. `__device__` est utilisé quand la fonction est exécutée et appelée depuis le *device*. `__host__`, pour une exécution et appel depuis l'*host*, est le qualificatif par défaut. De plus, les appels de noyaux depuis l'hôte doivent être accompagnés d'une configuration définie avec des triples chevrons ⋘`Dg, Db`⋙. `Dg` décrit la grille de calcul sur laquelle le noyaux va s'exécuter en indiquant le nombre de blocs présents sur la grille en 1D, 2D voire 3D. `Db` décrit chaque bloc présent dans la grille en indiquant le nombre de fils d'exécution, *thread*, à exécuter en 1D, 2D voire 3D.

Les variables présentes sur le GPGPU peuvent se trouver sur différents types de mémoire. Les variables en paramètre peuvent être sur la mémoire globale ou constante du GPGPU avec les qualificatifs respectifs `__device__` et `__constant__`. Les variables déclarées au sein d'une fonction sont par défaut mises en registre si possible, sinon elles se retrouvent dans la mémoire globale du GPGPU, le temps de son exécution. Le qualificatif `__shared__` peut être utilisé

pour utiliser la mémoire au niveau des blocs qui est beaucoup plus rapide que la mémoire globale du GPGPU. Dépendantes du placement en mémoire choisi, les performances ne sont pas les mêmes. Un accès à une variable en `__shared__` est plus rapide qu'un accès pour une variable en `__device__`. La gestion de la mémoire du GPGPU s'effectue à l'aide des fonctions `cudaMalloc`, `cudaFree`. Les transferts de données entre CPU et GPGPU se font avec la fonction `cudaMemcpy`.

1.5.2 OpenCL

Le langage OpenCL, *Open Compute Language* [59][60], est un langage permettant l'exécution d'un programme sur tout type de matériel, aussi bien des processeurs que des cartes graphiques ou divers accélérateurs. Il a l'avantage d'être libre et est porté par le consortium Khronos [58] regroupant un grand nombre d'acteurs aussi bien académiques qu'industriels.

OpenCL décrit l'architecture sur laquelle il intervient avec les termes suivants :

- un *NDRange* fait référence à une exécution sur le matériel désiré, il est composé d'une mémoire appelée mémoire globale et d'un ensemble de *work group*, il correspond à une grille sur CUDA ;
- un *work group* est composé d'une mémoire appelée mémoire locale et d'un ensemble de *work item*, il correspond à un bloc CUDA, sa mémoire locale correspond à la mémoire partagée en CUDA ;
- un *work item* possède une mémoire appelée mémoire privée [1] correspond à un thread CUDA.

Du fait de sa portabilité sur tous types de plateforme, un plus grand nombre de fonctions de configuration sont nécessaires. Ainsi, la création d'un contexte définissant le matériel sur lequel va s'exécuter le noyau est donné avec la fonction `clCreateContext` ou `clCreateContextFromType`. De plus, une queue de commandes, permettant d'envoyer et recevoir les données de même que de lancer le noyau, est créée avec la fonction `clCreateCommandQueue` et nécessite le contexte précédemment défini. De même, la compilation du noyau se fait dynamiquement au moyen des fonctions successives suivantes `clCreateProgramWithSource` qui a besoin du contexte précédent et du code source du programme à exécuter sur le matériel distant ; `clBuildProgram` qui compile celui-ci ; et `clCreateKernel` qui définit le noyau. Le code écrit en OpenCL contenant la fonction qui correspond au noyau devra avoir le qualificatif `__kernel`.

Pour définir sur quelle mémoire les données doivent être allouées, les qualificatifs `__global`, `__constant`, `__local` et `__private` correspondant aux différents niveaux de mémoire sont disponibles. Les variables présentes dans la mémoire globale et constante de l'accélérateur matériel doivent être déclarées et allouées depuis l'hôte au moyen des fonctions `clCreateBuffer` en utilisant le contexte et `clEnqueueWriteBuffer` en utilisant une queue de commande. Tous les arguments à envoyer à l'accélérateur et servant à faire les calculs du noyau sont définis à l'aide de `clSetKernelArg` qui nécessite le noyau, le rang de l'argument, sa taille et sa valeur. La demande d'exécution du noyau s'effectue en ajoutant le noyau à la queue de commande au moyen de la fonction `clEnqueueNDRangeKernel` où la dimension et le découpage des *work group* et

1. lorsque l'architecture réelle ne possède pas de mémoire privée au niveau du *work item* celle-ci peut prendre place dans la mémoire globale, dépendant de l'implémentation d'openCL sous-jacente

work item sont faits. `clFinish` permet d'attendre la fin de l'exécution du noyau et `clEnqueueReadBuffer` de récupérer les résultats voulus.

1.5.3 OpenACC

OpenACC, *Open Accelerator* [82][83], est une bibliothèque de parallélisation en C, C++ et Fortran. Elle regroupe un ensemble de directives permettant à du code de s'exécuter sur des GPGPU sans avoir à écrire dans un langage spécifique tel que CUDA ou OpenCL. Elle est née de la volonté d'offrir un accès plus simple à la programmation sur GPGPU tout en ayant des performances comparables à des programmes écrits dans des langages spécifiques. Cela ne l'empêche pas de pouvoir interagir avec d'autres bibliothèques ou langages orientés GPGPU. Ainsi toutes les bibliothèques citées en section 1.2 et les langages cités précédemment dans cette section peuvent interagir.

OpenACC repose sur un ensemble très restreint de directives qui peuvent être appliquées. La plus importante est `#pragma acc kernels` qui permet de demander au compilateur de générer des noyaux pour le GPGPU lorsque c'est possible. Si le compilateur ne juge pas possible de générer des noyaux, à cause de dépendances dans une boucle par exemple, il ne le fera pas.

Tous les éléments des noyaux seront transférés, entre l'hôte et l'accélérateur, à chaque entrée et sortie d'un noyau, ce qui affecte souvent les performances. Pour régler ce problème la directive `#pragma acc data` peut être utilisée. Les clauses `copy`, `copyin` et `copyout` sont à utiliser pour indiquer si les variables doivent être envoyées à l'accélérateur au début de la clause et/ou reçues de l'accélérateur à la fin de la clause.

Si le programmeur est certain que les itérations d'une boucle sont indépendantes les une des autres, la directive `#pragma acc loop` accompagnée de la clause `independent` peut être utilisée. Elle permet au compilateur de s'affranchir du calcul de dépendances sur cette boucle. De plus, les clauses `private` ou `reduction` permettent d'indiquer si des variables sont privées à une itération de boucle ou qu'une réduction s'applique sur certaines d'entre elles. Contrairement à OpenMP, elles sont optionnelles et sont souvent automatiquement déduites par le compilateur. Les clauses `collapse` ou `tile` permettent quant à elles de regrouper plusieurs nids de boucles en un seul ou de les redécouper selon les paramètres indiqués. Enfin, la possibilité de répartir les différentes itérations dans différents groupes pour effectuer les calculs est possible avec les clauses `gang`, `worker` et `vector`. Ces dernières servent aussi bien pour obtenir de la localité sur l'accélérateur qu'à utiliser au maximum toutes les unités de calculs disponibles.

1.5.4 OpenMP 4.x

Une introduction d'OpenMP a déjà été faite en section 1.2.2 présentant notamment son utilisation historique en mémoire partagée. Mais depuis la version 4.0 sortie en 2013 et 4.5 en 2015, de nouvelles directives ont été ajoutées pour profiter de la présence d'un accélérateur tel qu'un GPGPU.

La philosophie d'OpenMP par rapport à OpenACC est relativement différente, bien qu'elles utilisent toutes deux des directives pour effectuer la parallélisation.

OpenMP requiert que le programmeur garantisse la correction de son code lorsqu'il utilise des directives OpenMP. Par exemple, c'est au programmeur d'assurer la présence ou l'absence de dépendances de données. Le programmeur indique ce qui doit ou ne doit pas être parallélisé et le compilateur génère le code correspondant sans nécessairement vérifier la correction du code demandé.

OpenACC permet au programmeur d'indiquer explicitement ce qui doit être parallélisé, mais aussi de faire des suggestions de parallélisation au compilateur. Dans ce dernier cas, le compilateur devra réaliser plusieurs analyses pour déterminer si un code parallèle peut être généré et avec quelles caractéristiques : données à communiquer, répartition des données sur l'accélérateur, etc.

Chacune de ces deux approches a ses avantages et inconvénients. En OpenMP, le compilateur est relativement simple mais beaucoup d'efforts sont déportés sur le programmeur. Alors qu'en OpenACC, le programmeur a juste à indiquer ce qu'il aimerait paralléliser et le compilateur doit se charger de vérifier leur faisabilité. Malgré ces différences, OpenMP et OpenACC tentent de converger vers une norme commune. Plusieurs membres participent aux deux commissions de spécification d'OpenMP et OpenACC.

La directive OpenMP permettant d'indiquer quelle partie du code doit être exécutée sur un accélérateur est `#pragma omp target`. Elle est l'équivalente de la directive OpenACC `#pragma acc kernels`. La directive OpenMP peut avoir comme argument `map(...)` permettant de lister explicitement les variables qui doivent être envoyées et/ou reçues de l'accélérateur. Cette distinction se fait avec les indications `to`, `from` et `tofrom`.

Si plusieurs noyaux sont présents successivement, pour limier les communications effectuées, la directive `#pragma omp target data` permet de définir l'environnement que l'accélérateur utilisera. Elle s'accompagne de l'argument `map` présenté précédemment. Elle correspond à la directive OpenACC `#pragma acc data`.

Les autres directives classiques de OpenMP présentées dans la section 1.2.2 peuvent être utilisées au sein des instructions exécutées par l'accélérateur.

1.6 Conclusion

Dans ce chapitre, j'ai présenté les paradigmes de parallélisation (section 1.1) et les modèles de programmation, avec leurs langages ou API associés (sections 1.2 à 1.5), les plus couramment référencés et utilisés.

Dans le cadre de ma thèse, je me suis principalement intéressé au parallélisme de tâches et à la génération automatique d'un code source permettant de les exécuter en parallèle. Peu de travaux de ce type existent comparés au cas particulier du parallélisme de boucles. De plus, ma recherche visait un modèle de programmation à mémoire distribuée, impliquant des problématiques supplémentaires par rapport à l'usage d'une mémoire partagée. L'ensemble de ces problématiques sont exposées dans le chapitre 2. Le langage cible choisi est le MPI, car il est multi-cible C ou Fortran, et permet un meilleur contrôle des communications comparé aux langages PGAS. De plus, il peut parfaitement se combiner avec la bibliothèque OpenMP pour avoir accès à la mémoire partagée ou à OpenCL ou OpenACC pour l'utilisation d'un GPGPU.

Chapitre 2

La distribution automatique de tâches

Sommaire

La distribution automatique de code a été étudiée depuis les années 90, mais l'échec de HPF et le succès de MPI ont donné un coup de frein à ces travaux.

La section 2.1 expose les problèmes présents dans un code parallèle de tâches sur architectures à mémoire distribuée. Différents outils permettant de générer du code parallèle distribué sont présentés et discutés en section 2.2. Nous énumérons ensuite une série de critères permettant de comparer les différentes techniques les unes par rapport aux autres et les choix effectués pour nos travaux en section 2.3. La méthodologie mise en œuvre pour la génération automatique du code parallèle est décrite en section 2.4. La section 2.5 expose la modélisation des communications qui est réalisée.

2.1 La problématique de parallélisation de tâches sur architecture à mémoire distribuée

On oppose traditionnellement le parallélisme de tâches, décrit en section 1.1.4, au parallélisme de données et de boucles, décrit en section 1.1.3. Mais le parallélisme de boucles conduit naturellement à du parallélisme de tâches *via* des regroupements d'itérations quand le parallélisme est régulier, ou *via* des groupements de données quand le parallélisme est plus irrégulier comme c'est le cas dans les algorithmes de graphes.

On capitalise dans ce travail sur les tâches apparaissant naturellement dans les algorithmes et sur les tâches créées par des transformations de programmes affines et non-affines connues tel que le *blocking*, le *tiling*, le *chunking* ou encore le *strip-mining*. Les temps d'exécution que nous obtiendrons seront autant tributaires de la qualité du graphe de tâches obtenus préalablement, pour le placement, que de la qualité de notre processus de génération de code.

Le parallélisme de tâches implique également que les tâches puissent exécuter des instructions différentes, à l'opposé du parallélisme de données ou de boucles où les processus parallèles exécutent les mêmes instructions. Les contraintes ne sont donc pas les mêmes, d'autant plus si on se trouve dans un modèle à mémoire distribuée.

Pour obtenir du parallélisme de boucles sur une architecture à mémoire distribuée, il est important (1) que la valeur des données soit à jour sur le processus voulant l'utiliser et (2) de n'avoir aucune dépendance inter-itérations, à des exceptions près comme les réductions. Le point (1) doit être garanti pour avoir un code correct et (2) permet d'obtenir de meilleure performance. Pour de la parallélisation de boucles, sur une architecture à mémoire distribuée, comme tous les processus exécutent les mêmes instructions mais sur des données différentes, la gestion des données en mémoire est un vrai challenge, surtout quand il doit être assuré par le programmeur.

D'un autre coté, la parallélisation de tâches sur une architecture à mémoire partagée doit connaître les contraintes de précédences et de dépendances entre chacune des tâches pour générer un ordonnancement correct, c'est-à-dire qu'une tâche ne commence pas à s'exécuter avant que les données en entrée n'aient été mises à jour par une précédente tâche. Le parallélisme de tâches peut se faire (1) si les dépendances entre tâches sont bien définies, et (2) s'il y a des tâches qui n'ont pas de dépendances entre elles, sinon le code est séquentiel. Il faut donc faire en sorte de bien créer et répartir les tâches pour avoir le minimum de dépen-

dances inter-tâches tout en ayant des tâches suffisamment conséquentes. Étant donné que tous les processus partagent la mémoire, n'importe quel processus peut exécuter une tâche donnée sans que cela n'ait de conséquence sur le résultat obtenu. Des algorithmes de vol de tâches, *workstealing*, peuvent s'appliquer dans le cas de cette parallélisation, comme c'est le cas avec XKaapi[37]. Ainsi, la problématique principale est un problème d'ordonnancement des différentes tâches sur les différents processus disponibles.

Lorsque l'on considère la parallélisation de tâches sur une architecture à mémoire distribuée, les deux problèmes précédents doivent être considérés simultanément : (1) la gestion et la répartition des données sur les différents processus qui vont exécuter les tâches ; (2) l'ordonnancement des tâches sur les différents processus, et les instructions que ces processus devront exécuter. Le point (1) vient du fait que l'on souhaite avoir du parallélisme sur une architecture distribuée, tandis que la parallélisation de tâches entraîne (2).

2.2 L'état de l'art

Plusieurs travaux existent pour la génération automatique de code distribué. Certains s'appuient sur un langage parallèle comme HPF pour générer le code distribué correspondant, section 2.2.1. D'autres partent d'un code parallèle à mémoire partagée pour le transformer en un code à mémoire distribuée, section 2.2.2. Enfin, certains abordent le problème directement avec un code séquentiel à paralléliser, section 2.2.3.

2.2.1 Compilation de code HPF

Plusieurs travaux concernant la compilation du langage HPF (section 1.3.3) ou ces ancêtres ont été présentés dans les articles [19, 25, 111, 47, 104, 8].

Comme l'explique Kennedy et al. [55], la plupart des compilateurs HPF de l'époque ne généraient pas directement un code assembleur pouvant être interprété par la machine, mais un code intermédiaire utilisant la bibliothèque MPI ou PVM. Par exemple, la thèse de Coelho [27][11] présente une génération des communications utilisant la bibliothèque PVM, associée à son compilateur pour HPF : HPFC. Deux programmes sont générés, un pour l'hôte, ou le maître, et l'autre pour les différents nœuds. Le programme hôte est chargé de lancer le programme de calcul sur les différents nœuds, et de gérer les entrées/sorties de l'exécution. Les programmes pour l'hôte et les nœuds sont accompagnés de fonctions s'exécutant au *runtime* servant à déterminer l'allocation ou la réallocation mémoire sur chaque nœud, ainsi qu'à déterminer les communications inter-nœuds. Les éléments à communiquer sont déterminés grâce à une analyse polyédrique des références aux tableaux dans le programme.

Cette solution nécessite l'apprentissage d'un nouveau langage de programmation HPF qui a été abandonné.

2.2.2 STEP/dSTEP

STEP [68, 69] permet de générer un code distribué en MPI à partir d'un code parallèle utilisant OpenMP. Pour cela il utilise l'analyse des régions de

tableaux fournies par le compilateur PIPS [88] pour générer les communications qui doivent être faites. Un de ses points faibles est que chaque processeur alloue la même quantité de mémoire même si le processeur ne modifie qu'une partie de celle-ci. Ce défaut a été corrigé dans l'extension dSTEP réalisée par Habel [42][43].

dSTEP permet d'avoir une meilleure distribution des données sur chaque processeur et propose différents ordonnancements pour l'exécution des itérations de boucles, grâce respectivement aux pragmas `dstep distribute` et `dstep gridify`.

`dstep distribute` permet d'exprimer la distribution d'un tableau sur les différents processus. Cette distribution peut être en bloc, cyclique, répliquée ou en diagonal. Cette distribution possède également un halo. Ce halo représente les éléments des tableaux qui devront être communiqués entre les différents processus. La Figure 2.1 montre un exemple des distributions possibles avec la directive `dstep distribute` d'un tableau 2D sur 4 processus.

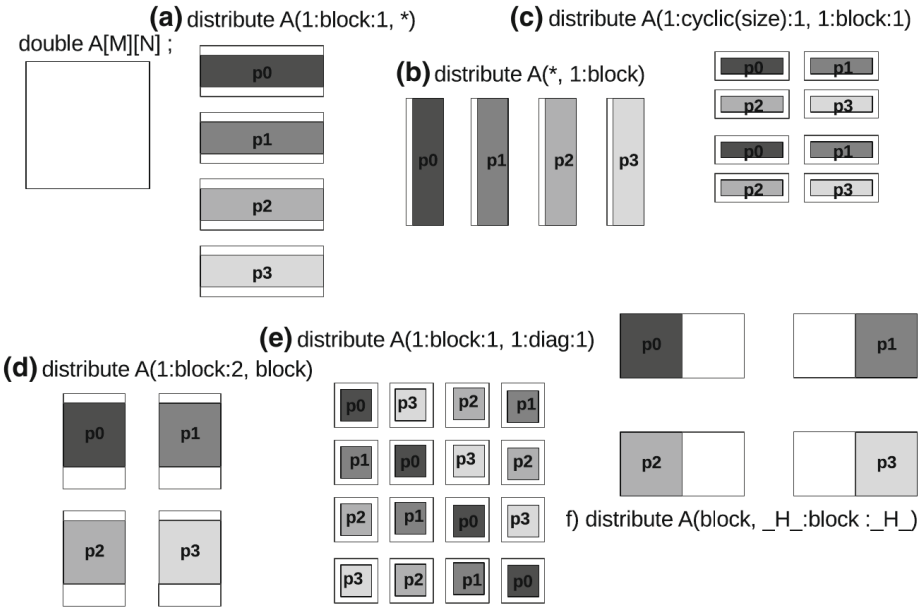


Figure 2.1 – Exemple de distributions d'un tableau 2D sur 4 processus avec `dstep distribute` (source [43])

`dstep gridify` permet d'exprimer à la fois la distribution et l'ordonnancement d'une boucle parallèle. Cette directive peut s'appliquer sur un ensemble de boucles imbriquées, en indiquant quel itérateur de boucle est concerné. La distribution s'exprime avec le paramètre `dist` et peut être par bloc (`block`), cyclique (`cyclic`), ou répliquée (`replicated`). L'ordonnancement s'exprime avec le paramètre `sched` et peut être parallèle (`parallel`), ordonné (`ordered`), ou fait sur chaque processus qui possède une donnée (`owner`). Des options similaires à une boucle parallèle en OpenMP peuvent également s'appliquer à cette directive, à savoir `private`, `firstprivate` ete `reduction`.

De plus, des fonctions propres à dSTEP sont nécessaires à l'exécution du code généré. Ainsi, une meilleure répartition de charge est faite permettant une

accélération de l'exécution et permettant de traiter des problèmes plus gros, tout en restant simple d'utilisation.

Néanmoins, STEP et dSTEP ne prennent pas en compte toutes les options proposées par OpenMP. Notamment la parallélisation de tâches n'est pas possible et seule la parallélisation de boucles est effectuée.

2.2.3 Pluto/Pluto+

Pluto et son extension Pluto+ [89][6][21] est un outil de parallélisation automatique reposant sur le modèle polyédrique. C'est un compilateur source-à-source travaillant sur du code C. Il génère un nouvel ordonnancement des itérations de boucles afin de minimiser les dépendances entre les instructions les composant. Ainsi, il permet entre autre de réordonnancer les nids de boucles de telle sorte que les boucles externes ne portent pas de dépendances, *ie.* les boucles sont totalement parallélisables. Pour cela, il utilise une abstraction polyédrique pour représenter le domaine d'itérations des boucles et les différentes dépendances qui existent entre les différents indices. Pluto+ est très performant sur de la parallélisation de boucles en réordonnançant l'ordre de ces itérations pour les rendre totalement indépendantes lorsque c'est possible.

Une autre de ces extensions [20][29], encore expérimentale, permet de générer un code distribué au moyen de la bibliothèque MPI. Le code parallèle obtenu peut s'exécuter sur un nombre arbitraire de processus et utilise des communications collectives. Le parallélisme sur les boucles est du parallélisme par bloc. La taille de ces blocs est calculé dynamiquement par rapport au nombre de processus lancés et au nombre d'itérations total. Une fonction est chargée de déterminer dynamiquement les différents processus qui doivent communiquer entre eux ainsi que le nombre d'éléments à communiquer. Une communication collective est ensuite effectuée pour partager ces informations, avant que les réelles données ne puissent être envoyées point à point entre les différents processus concernés.

Cette extension, permettant de générer du code distribué, ne marche que sur un nombre limité de code pour le moment. De plus, il est possible que du code invalide soit généré. Ce dernier est très difficilement compréhensible, ce qui rend son exploitation et sa correction difficile.

2.2.4 Bilan sur les outils existants

Cette section a présenté trois outils permettant de générer automatiquement du code pour des architectures à mémoire distribuée. La plupart de ses outils traitent de la parallélisation de boucles (dSTEP, Pluto+). De plus, des fonctions supplémentaires, exécutées dynamiquement, sont nécessaires pour déterminer les variables à communiquer. Ainsi, des surcouches au code initial doivent être ajoutées pour l'exécution du code. Ces dernières peuvent parfois faciliter la lecture du code, mais ne sont pas toujours dépourvues d'erreurs. Dans ce cas les corrections peuvent se révéler laborieuses, et rendre le code difficilement compréhensible. Enfin, il est souvent difficile de comprendre le code généré par ces outils automatiques car la démarche pour aboutir au code final n'est pas toujours bien documenté et l'outil de génération de code se comporte d'une certaine façon comme une boîte noire.

2.3 Les différentes possibilités

Comme le montre l'état de l'art, il existe déjà des approches variées pour effectuer automatiquement la distribution d'un code et pour obtenir du code distribué. Dans cette section, nous précisons les choix qui peuvent être faits pour structurer ce domaine et commençons à en effectuer certains. Le compromis entre avoir prioritairement un code pouvant être vérifié dans tous les cas, et avoir un code performant dans certains cas, mais sans garantie de correction est discuté en section 2.3.1. Les choix sur le code d'entrée et de sortie sont abordés en section 2.3.2 et section 2.3.5. Le nombre de processus et le placement souhaités sont traités dans les sections 2.3.3 et 2.3.4. Enfin les hypothèses retenues dans cette thèse sont résumées en section 2.3.6.

2.3.1 La correction avant les performances

Les codes distribués étant beaucoup plus difficiles à mettre au point que des codes séquentiels ou même parallèles à mémoire partagée, ils ne sont envisagés que pour résoudre des problèmes de performance ou de ressources comme la taille mémoire ou le débit disque.

Nous nous attachons néanmoins à trouver un processus de compilation correct et automatisable, pouvant ensuite être étendu par de nouvelles optimisations, pour réduire les surcoûts de communication et les volumes mémoire nécessaires.

Ainsi, nous souhaitons mettre en avant une solution correcte, avant de nous attacher dans un second temps aux performances, et cela avec le maximum de transparence, c'est-à-dire sans appel de fonctions, dans le code généré, dont l'utilisateur n'aurait pas connaissance. Cet objectif de correction et de transparence est l'un des points essentiels de la thèse, prioritaire sur les performances que l'on pourra obtenir.

En effet, les performances peuvent être visées en priorité et être obtenues en une seule grande transformation, mais elle est souvent dure à prouver correcte. Plusieurs tests et *benchmark* peuvent tenter de persuader l'utilisateur que la transformation est correcte. Mais une preuve sera souvent absente. De plus, lors de la recherche de performances, les caractéristiques des machines utilisées et de leur connectique influencent souvent le résultat. Ainsi, un code optimal sur une machine ne le sera pas forcément dans d'autres configurations. Pour ces raisons, il n'est pas forcément avisé de réaliser une unique grosse transformation.

2.3.2 Quel code veut-on transformer ?

Le premier problème qui se pose est : Quelle application en entrée veut-on considérer ? Quel type de code veut-on transformer ?

Le processus de distribution automatique peut se limiter à l'optimisation d'un code déjà distribué explicitement ou à la conversion d'un code distribué implicitement, par exemple avec des directives HPF ou dSTEP portant sur le placement des données et/ou des instructions.

Il peut être plus ambitieux de viser la conversion d'un code parallèle à mémoire partagée en un code parallèle à mémoire répartie comme cela est fait par STEP.

Enfin, le processus peut viser la conversion directe d'un code séquentiel en un code distribué. Cette conversion peut être atomique et faire l'objet d'une unique passe comme Pluto+, ou bien être constituée d'une séquence de passes élémentaires comme dans un compilateur séquentiel.

C'est ce dernier scénario qui nous intéresse ici puisque nous considérons le cas où le code en entrée est encore séquentiel, mais a fait l'objet d'un placement préalable à sa distribution.

En conséquence, les résultats que nous obtiendrons seront tributaires de la qualité du placement effectué préalablement à la distribution. Des algorithmes de placement et d'ordonnancement, tel que l'algorithme HBDSC (Hierarchical Bounded Dominant Sequence Clustering) de Khaldi [56], peuvent être utilisés. Ce placement peut être représenté par des annotations sur le code initial.

Partir d'un code séquentiel avec des annotations de placement a l'avantage de permettre soit l'utilisation d'un outil de génération automatique de placement, soit l'ajout manuel des directives, si le programmeur est assez expert en programmation parallèle.

2.3.3 Le nombre de processus doit-il être connu ?

Le nombre de processus doit-il être connu statiquement ou dynamiquement ? numériquement ou paramétriquement ?

Nombre numérique de processus

On entend par nombre numérique de processus le fait de connaître numériquement le nombre de processus avant l'exécution du programme et que celui-ci reste fixe durant son exécution.

L'avantage de connaître numériquement le nombre de processus est que l'on peut obtenir un placement des tâches performant et adapté à l'architecture cible.

L'inconvénient est que, si, lors de l'exécution, il y a davantage de processeurs que de processus prévus, toutes les ressources mises à disposition ne seront pas utilisées. *A contrario*, si, lors de l'exécution, il y a davantage de processus que de processeurs réels, soit le programme ne pourra pas s'exécuter du tout ; soit chaque processeur réel se verra affecter plusieurs processus virtuels, ce qui peut réduire les performances avec une mauvaise répartition de charge sur les processeurs réels.

Une connaissance numérique du nombre de processus est particulièrement adaptée pour des codes embarqués avec une plate-forme souvent bien définie. De même, tout autre application se terminant peut en profiter.

Nombre paramétrique de processus

On entend par nombre paramétrique de processus le fait de ne pas connaître numériquement le nombre de processus avant de lancer le programme, mais une fois lancé, celui-ci ne change pas.

Un code paramétrique a l'avantage d'être flexible par rapport à la plate-forme sur laquelle il va s'exécuter, par exemple sur une grappe de calcul ou encore sur le nuage, *cloud* ; tout en supportant une bonne mise à l'échelle (*scalability*).

L'inconvénient est que le code doit être très régulier sous peine d'avoir des difficultés pour générer le code parallèle et une très mauvaise répartition de charge lors de son exécution.

Un nombre paramétrique de processus est utile pour les gros codes réguliers. Les codes de calculs scientifiques, tels que du traitement du signal ou d'images, s'accordent bien avec cette configuration. De même la distribution de boucles parallèles convient à cette configuration, telle que le fait Pluto+ [89].

Nombre dynamique de processus

Un nombre dynamique de processus implique qu'on ne connaisse pas le nombre de processus avant l'exécution du programme, et que celui-ci puisse varier durant son exécution.

Un programme conçu pour s'exécuter sur un nombre de processus connu dynamiquement a l'avantage de pouvoir s'adapter à tout type de machines.

Par contre, les performances sont plus délicates à obtenir car il faut régulièrement modifier la répartition des données et l'ordonnancement du programme. De même il est impossible d'exploiter des informations de placement potentiel, car il n'est pas possible de déterminer quelles instructions sont exécutées par quel processus.

Une application devant être déployée sur un éventail de machines différentes tirera profit d'un code pouvant s'adapter dynamiquement. De même, des applications temps réel qui doivent tourner en permanence sur des serveurs peuvent préférer un code dynamique.

Choix de la connaissance du nombre de processus

Un nombre de processus dynamique rend difficile voire impossible toute preuve de correction du code. Un programme totalement dynamique implique l'introduction d'un indéterminisme qui rend difficile toute preuve sans hypothèses supplémentaires. De plus, les calculs scientifiques sont en général très réguliers et concernent principalement du parallélisme de boucles et non de tâches. Ainsi, un nombre de processus paramétrique n'est pas le plus adapté pour faire du parallélisme de tâches dans le cas général.

Ainsi, dans cette thèse, nous considérons que le nombre de processus est connu numériquement dès la phase de compilation. Nos travaux sont particulièrement adaptés aux applications embarquées, mais ils peuvent également s'appliquer aux calculs scientifiques réguliers.

2.3.4 Quid du placement ?

Le placement consiste à définir sur quel processus va exécuter une tâche pendant l'exécution d'un programme. Tout comme le nombre de processus, le placement peut être statique, paramétrique ou dynamique et s'appuyer sur un nombre de processus connu ou variable.

Placement paramétrique

Un placement paramétrique peut être interprété de deux manières différentes. La première est similaire à un nombre paramétrique de processus, le placement n'est pas connu avant l'exécution, par exemple on ne connaît pas le nombre de processus qui vont être exécutés, mais une fois lancés ce nombre ne change plus. La deuxième concerne des parties du code qui ont un placement à un moment donné, puis un autre placement à un autre instant car le nombre

de processus a changé entre temps. Par exemple, les différentes instances d'une boucle parallèle seront placées de manières différentes en fonction du nombre de processus disponibles et pouvant varier lors de l'exécution. C'est un cas particulier de placement dynamique qui passe bien à l'échelle.

Dans notre cas, nous avons un nombre de processus connu numériquement dès le début de la compilation. Un placement paramétrique n'apporte aucun avantage par rapport aux autres types de placement.

Placement dynamique

Un placement dynamique permet d'assigner, durant l'exécution, les tâches à exécuter à des processus différents. Un placement dynamique fonctionne aussi bien avec un nombre de processus numérique que paramétrique ou dynamique.

Il pose des problèmes de localité des données qui sont traités en exploitant les *affinités* entre tâches. De nombreuses heuristiques existent, pour des tâches indépendantes ou non. Elles sont fondées sur l'affectation de priorités et par l'utilisation de pools de tâches (*tasks pools*). La plupart du temps, ces pools de tâches fonctionnent soit en file d'attente, soit par vol de tâches. Il entraîne souvent un léger surcoût de gestion qui est normalement compensé par une meilleure utilisation des processus et une meilleure répartition de charges. Couplée à ce problème, l'allocation mémoire doit être considérée. La mémoire peut être allouée identiquement sur tous les processus leur permettant d'exécuter toutes les tâches. Beaucoup de ressources mémoire sont donc utilisées dans ce cas. Une allocation dynamique de la mémoire, lorsqu'elle est nécessaire, peut être faite. Mais cela est très compliquée car il faut déterminer si un processus a besoin ou non d'une variable avant même son utilisation. Une autre possibilité est de ne répliquer que certaines parties de la mémoire et contraindre l'exécution des tâches qu'aux processus disposant de ces variables.

Le placement dynamique est adapté lorsqu'il y a un ordonnanceur dynamique de tâches chargé de répartir les tâches sur les processus, comme c'est le cas en mémoire partagée. Par contre, il est plus compliqué à mettre en place lorsqu'il n'y a pas d'ordonnanceur, comme c'est le cas en mémoire distribuée, où il faut gérer à la fois la répartition des données mais également celle des tâches. Un ordonnanceur distribué implique forcément un mode de parallélisme maître/esclaves, où le maître est l'ordonnanceur. Un mode pair-à-pair est difficilement envisageable. Le maître doit à la fois communiquer aux processus leurs tâches à faire, mais également envoyer et récupérer le résultat des différentes tâches, ce qui peut être laborieux à grande échelle. Un goulot d'étranglement pourrait apparaître au niveau de celui-ci. La bibliothèque MPI, considérée à l'heure actuelle comme la référence pour du code distribué, n'est vraiment pas adaptée pour gérer un ordonnancement dynamique. Une des raisons est que MPI fait uniquement des communications que je qualifierais d'actives, c'est-à-dire que, pour communiquer, deux processus doivent explicitement s'identifier : un processus (un esclave) ne peut pas dire à un autre processus (le maître) qu'il est disponible pour effectuer du travail, *i.e.* une tâche, si le second processus n'attend pas explicitement de notification du premier.

Dans le cas où le nombre de processus est connu numériquement, le placement dynamique permet de mieux utiliser les ressources de calcul des machines quand les temps d'exécution des tâches sont inconnus ou fonction des données d'entrée. Il permet d'allouer dynamiquement une tâche sur un processus dis-

ponible à un instant donné. Une meilleure utilisation des ressources de calculs des machines disponibles est ainsi faite et les machines hétérogènes ayant des performances diverses sont mieux gérées.

Placement statique

Le placement statique permet d'optimiser les communications et la localité des données. Il est en général limité au cas où le nombre de processus est connu numériquement. Mais, il peut également s'appliquer, quand le nombre de processus est paramétrique, à du parallélisme de boucles en associant les itérations des boucles aux processus de manière équilibrée.

Lorsque la durée des tâches est connue, il permet souvent l'obtention de meilleures performances. En effet, couplé à la connaissance des temps de communication, il est théoriquement possible d'en déduire un ordonnancement des tâches optimal.

Le placement statique ne peut réellement être optimal que lorsque le nombre de processus virtuels correspond au nombre de processeurs réels. Il peut tout de même être utilisé avec un nombre de processeurs réels supérieur en ignorant certains d'entre eux ; ou avec un nombre de processeurs réels inférieur, en associant plusieurs processus virtuels à un processus réel. Dans ce cas, la répartition des charges peut être déséquilibrée entre les processeurs exécutant plusieurs processus.

Choix du placement

Le caractère dynamique du placement soulève des problèmes similaires à un nombre dynamique de processus pour la preuve de correction. Le manque de déterminisme que cela engendre rend très difficile la mise en œuvre d'une preuve de programme. De plus, l'implémentation performante d'un ordonnanceur n'est pas chose aisée, de même que la preuve de sa correction. Un placement dynamique n'est donc pas un choix judicieux pour de la génération automatique de code sur architecture à mémoire distribuée pour des tâches.

Ainsi, nous avons choisi de nous placer dans le cadre d'un placement statique. Il a l'avantage (1) de faciliter les preuves de programme et (2) de permettre l'obtention de bonnes performances tant au niveau de la vitesse d'exécution que de l'allocation mémoire. De plus, cela s'accorde avec nos choix précédents notamment avoir un nombre statique de processus, connu numériquement. Enfin, cela se démarque de la parallélisation de tâches sur mémoire partagée qui utilise traditionnellement un placement dynamique, mais qui nécessite, en contre partie, des informations sur les données en entrée et sortie des tâches.

2.3.5 Quel code veut-on obtenir ?

Est ce que de nouvelles fonctions doivent être ajoutées et utilisées pour gérer les communications et/ou l'ordonnancement lors de l'exécution ? Ou peut-on seulement utiliser des primitives de communications ?

Encore une fois, le choix que j'ai fait est celui de se passer au maximum d'une gestion dynamique, c'est-à-dire de ne pas avoir de fonctions supplémentaires pour gérer la mémoire, le placement ou l'ordonnancement des tâches. Ainsi, je souhaite faire des transformations de bout en bout qui prennent un code séquentiel en entrée et génèrent un code parallèle distribué en sortie. Ce

dernier n'utilisera que des primitives de communications. Ces communications seront explicites, au moment où le code est généré, ainsi que les données communiquées. Il n'y aura pas de fonctions calculant quelle partie d'un tableau doit être envoyée à tel ou tel processus par exemple. Une sur-approximation des données communiquées est donc possible. Les instructions d'origine seront quasiment inchangées, voire identiques à l'utilisation des variables, propres à un processus, près.

De plus, l'ordonnancement final des tâches sur l'architecture à mémoire distribuée est fait implicitement au cours de l'exécution par rapport au placement initial demandé. Par exemple, considérons deux tâches successives sur deux processus différents, si aucune dépendance n'est présente entre ces deux tâches, alors aucune communication ne sera générée entre ces deux processus, et ainsi les deux processus pourront s'exécuter simultanément. *A contrario*, si une dépendance est présente entre deux tâches, alors une communication sera ajoutée. Cette communication imposera implicitement un ordre entre ces deux tâches, et le processus exécutant la deuxième tâche devra attendre que le premier processus lui ai communiqué son résultat.

2.3.6 Les hypothèses retenues

Pour résumé, nous souhaitons avant tout mettre l'accent sur la correction du programme généré et non sur ses performances, comme expliqué en section 2.3.1. Ces dernières sont obtenues dans un second temps au moyen d'optimisations.

Ma thèse concerne principalement la parallélisation de tâches qui est moins souvent étudiée que le parallélisme de données et de boucles, surtout lorsque l'on se place dans un environnement distribué. Nous nous attachons plus à la génération de code distribué qu'à un ordonnancement ou au placement de tâches. Ainsi, nous considérons un code séquentiel et uniquement des contraintes de placement des tâches sur les différents processus, comme décrit en section 2.3.2 et section 2.3.4. Le calcul des contraintes et dépendances entre les différentes tâches sont à notre charge. Contrairement à d'autres modèles de parallélisation de tâches, décrit en section 2.3.4, le calcul des dépendances entre les différentes tâches est effectué au cours de mon processus de compilation et de génération de code parallèle. La qualité du placement statique fixe a un impact important sur les performances obtenues. Mais il permet d'effectuer plus facilement des preuves de programmes. Les applications visées sont des applications embarquées et des applications de calculs scientifiques, qui s'accommodent bien d'un nombre de processus fixe, comme expliqué en section 2.3.3. Enfin, nous voulons autant que faire se peut être transparent sur les différentes étapes de transformations et de génération du code, ainsi constitué uniquement des instructions d'origine et des instructions de communications, comme décrit en section 2.3.2 et section 2.3.5.

2.4 La méthodologie utilisée

Maintenant que notre cadre et nos hypothèses sont bien définis, penchons nous sur les solutions possibles et sur la méthodologie de résolution que nous avons mis en place.

2.4.1 Compilation source-à-source

Le code distribué qu'on veut obtenir est un nouveau code source. On utilise, pour la génération du code parallèle, la bibliothèque MPI qui est la norme de référence concernant le parallélisme distribué.

Un des avantages des compilateurs source-à-source est qu'ils permettent de suivre simplement l'évolution des transformations appliquées et de détecter facilement les erreurs éventuelles car la représentation interne du compilateur peut toujours être exprimée sous la forme d'un code source compilable et exécutable. La volonté d'être transparent quant aux transformations appliquées est respectée avec la compilation source-à-source. Ainsi, rien n'est "caché" à l'utilisateur contrairement aux compilateurs traditionnels qui peuvent effectuer des transformations sans que l'utilisateur en soit forcément informé. De plus, le code généré reste lisible et compréhensible car le nouveau code est dans le même langage de programmation que le code initial. Enfin, le code généré peut être réanalysé et retransformé par un compilateur et ainsi tirer partie de toutes les transformations et optimisations disponibles dans le compilateur pour la génération de son code machine.

Nous avons implémenté nos transformations dans le compilateur source-à-source PIPS [88], développé par les MINES de Paris. Plus de détails concernant son fonctionnement sont disponibles en Annexe A.

Un avantage du compilateur PIPS est qu'il est assez ancien et robuste. Il dispose d'un large éventail d'analyses et de transformations déjà implémentées qui ont été utilisés pour générer le code désiré. Ces analyses fonctionnent également de manière inter-procédurale. Ce qui permet d'analyser une large gamme de programmes différents. De plus, PIPS a été conçu de façon modulaire. Cette modularité permet une réutilisation de l'existant mais facilite également le déploiement de nouvelles analyses et transformations. Enfin, PIPS prend en entrée des langages de relativement haut niveau comme le C ou le Fortran et génère du code source en C ou Fortran, contrairement à des compilateurs LLVM qui ont des sorties et une représentation de beaucoup plus bas niveau. L'utilisation du C ou du Fortran permet une meilleure compréhension par l'utilisateur des différentes transformations effectuées.

2.4.2 Transformations successives "simples" et prouvées

Nous avons également décidé d'effectuer, pour la transformation du code séquentiel en code parallèle distribué, une succession de petites transformations "simples" lorsque cela est possible, plutôt qu'une seule et unique grosse transformation. Ce choix est représenté par les chemins (E) et (F) de la Figure 2.2 présentée ci-après.

La Figure 2.2 récapitule les principaux chemins possibles permettant de générer du code parallèle distribué.

Ainsi, à partir d'un code avec une description parallèle pour mémoire partagée, (A) génère un code parallèle pour mémoire distribuée. Par exemple, STEP traduit un code OpenMP en un code distribué avec MPI.

(C) est assez similaire à (A) dans le cas où la description du parallélisme pour mémoire partagée est faite au moyen d'annotations comme dans OpenMP, mais

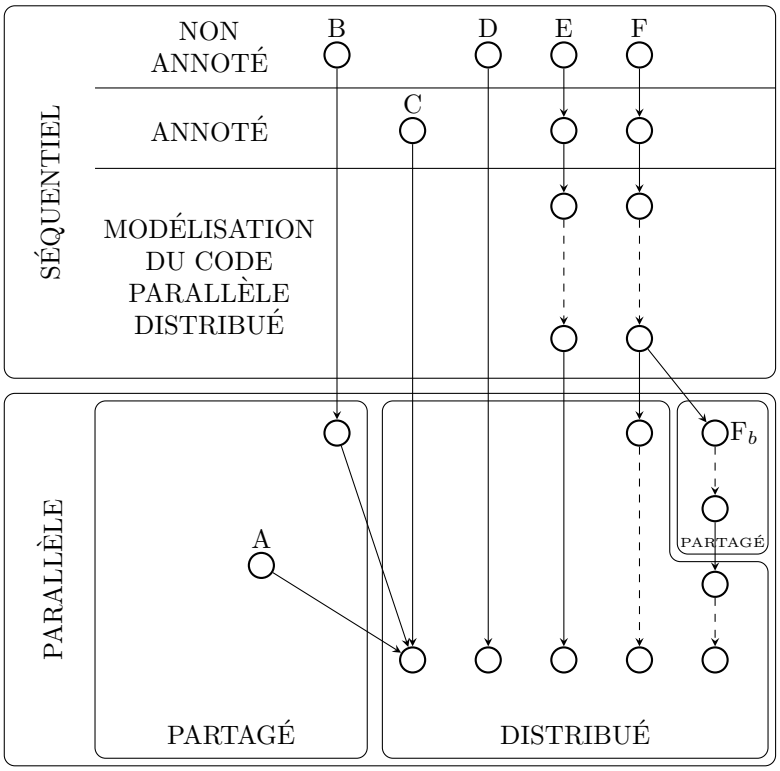


FIGURE 2.2 – Différentes possibilités pour générer du code parallèle distribué

il reste plus général. Les annotations peuvent être spécialement conçues pour d'écrire des caractéristiques des distributions de données comme par exemple dSTEP (section 2.2.2) qui introduit des pragmas personnalisés.

(B) prend en entré un code écrit purement séquentiellement. Avant de générer un code distribué, il le parallélise pour un environnement à mémoire partagée. Sachant qu'il y a plusieurs travaux qui génèrent du code parallèle pour mémoire partagée à partir de code séquentiel et qui passe de code partagé à distribué comme vue précédemment, ce cheminement peut se reposer sur une intégration de travaux préexistants.

(D) est une solution que l'on peut considérer comme étant une "boîte noire". Elle prend un code séquentiel et génère un code parallèle pour une architecture distribuée. Il faut faire confiance à la "boîte noire" comme on fait confiance à notre compilateur, `gcc`, `clang` ou `icc` par exemple. Mais le résultat obtenu est rarement compréhensible de prime abord. De plus, en cas d'erreur, à la compilation ou à l'exécution, dans le code généré, il est très difficile de réussir à le corriger. Bien que reposant sur des bases mathématiques solides, c'est ce que fait Pluto+, qui prend en entrée un code séquentiel et génère automatiquement un code parallèle pour architecture distribuée.

(E) génère un code parallèle distribué résultant de plusieurs transformations qui s'appliquent à du code séquentiel. Ainsi, dans un premier temps, une transformation ajoutant des annotations pour guider la parallélisation peut être effectuée. Puis, une succession de transformations préparant le code séquentiel

en vue d'une exécution parallèle distribuée est faite avant de générer le code distribué final. Nous verrons en section 2.5.1 que cette dernière étape soulève des difficultés pour générer un code distribué efficace juste à partir du code séquentiel. Ainsi, le dernier chemin (F) permet une traduction simple d'un code séquentiel en un code parallèle avant de continuer les optimisations appliquées au code parallèle.

C'est ce dernier chemin de compilation que nous avons choisi dans cette thèse, avec une succession de transformations "simples".

L'objectif est de prouver que le code final généré est correct. L'avantage d'exprimer le problème en une suite de plusieurs transformations "simples" est d'avoir des preuves pour chacune des transformations successives.

Une succession de transformations permet également de suivre l'évolution et les modifications faites au fur et à mesure. Cela facilite la compréhension pour l'utilisateur. Il peut à tout moment "garder la main" sur son code et faire d'autres modifications, dans une certaine mesure, à chaque étape des transformations si cela ne lui convient pas.

Faire une succession de transformations permet également de réutiliser des analyses et transformations déjà présentes dans le compilateur source-à-source PIPS. De même, toutes les analyses et transformations implémentées pourront être réutilisées dans PIPS pour des travaux futurs.

Il y a tout de même un inconvénient à une suite de transformations, c'est qu'il est parfois difficile de comprendre l'utilité d'une de ces transformations en tant que telle. C'est notamment le cas pour la phase de préparation de code que nous présentons en section 3.3. Une succession de transformations implique également un contexte et des hypothèses sur les codes d'entrée qui sont nécessaires avant chaque transformation.

2.4.3 Réutilisation d'analyses et de transformations existantes

Les analyses et transformations portant sur du code séquentiel sont très nombreuses et bien connues [9]. Ainsi, nous avons décidé de capitaliser autant que faire se peut sur les analyses et transformations existantes. Pour maximiser cette réutilisation dans un compilateur séquentiel, il faut conserver le code sous une forme séquentielle le plus longtemps possible. L'utilisation du compilateur source-à-source PIPS est tout à fait adaptée pour gérer ces analyses et transformations.

Idéalement, il faudrait que toutes les analyses et transformations s'appliquent sur du code séquentiel et qu'une dernière transformation génère du code distribué. Pour cela, comme montré à la Figure 2.2, nous faisons une modélisation du code parallèle en séquentiel avant de passer dans le monde parallèle. La suite de transformations successives introduite dans la section précédente s'applique donc principalement sur du code séquentiel.

Nous montrons, en section 2.5.1, qu'il n'est malheureusement pas possible de totalement modéliser du parallélisme de tâches distribué en séquentiel. Certaines transformations et optimisations du code ne peuvent se faire que dans un environnement parallèle. Une extension du compilateur PIPS peut être envisagée

pour introduire une gestion minimale de code parallèle, permettant de détecter les variables utilisées ou modifiées par des fonctions de communications.

2.4.4 Introduction des communications sous forme d'affectations

```
x_1 = x_2;
```

Code 2.1 – Affectation représentant une communication synchrone

```
if (p==2)
  send(x_2, 1);
else if (p==1)
  receive(x_1, 2);
```

Code 2.2 – Traduction parallèle du Code 2.1

Les communications, bloquantes ou non bloquantes, caractérisent les programmes distribués. Le résultat d'une communication est équivalent à une affectation ou bien un ensemble d'affectations. Réciproquement, une affectation peut être un précurseur d'une communication. Il suffit que les deux variables concernées soient implicitement ou explicitement placées sur des processus différents. Une passe ultérieure sera chargée de détecter la nécessité de remplacer l'affectation par une paire *send*/*receive* ou bien par un *remote read* ou un *remote write*. Par exemple, le Code 2.1 (`x_1 = x_2`) peut être interprété comme la copie de la valeur de la variable `x` sur le processus 2 à l'adresse de la variable `x` sur le processus 1. Le Code 2.2 correspond à une représentation parallèle distribuée de cette instruction de copie.

```
buf_1_2 = x_2;

...
x_1 = buf_1_2;
```

Code 2.3 – Affectation représentant une communication asynchrone

```
if (p==2)
  asend(x_2, 1);
else if (p==1)
  areceive(x_1, 2,
           notif);
...
if (p==1)
  wait(notif);
```

Code 2.4 – Traduction parallèle du Code 2.3

```
if (p==2)
  asend(x_2, 1);


...
if (p==1)
  receive(x_1, 2);
```

Code 2.5 – Traduction parallèle du Code 2.3

Les communications asynchrones sont plus compliquées à représenter. Mais elles peuvent être explicitées par un `buffer` intermédiaire, comme dans le Code 2.3. Ces deux affectations permettent d'exprimer indépendamment les différentes étapes de la communication, par exemple un `send` non bloquant associé à un `receive` asynchrone puis à un `wait`, ou bien un `send` non bloquant accompagné d'un `receive` bloquant. Entre ces deux affectations, d'autres instructions peuvent s'exécuter.

Le premier cas est représenté par le Code 2.4 où `buf_1_2 = x_2;` est traduit par un envoi et une réception asynchrone sur respectivement les processus 2 et 1,

et où `x_1 = buf_1_2` est traduit par une attente de confirmation de la réception sur le processus 1.

Le deuxième cas est représenté par le Code 2.5 où `buf_1_2 = x_2;` est traduit par un envoi asynchrone sur le processus 2, et où `x_1 = buf_1_2` est traduit par une réception bloquante sur le processus 1.

De plus, étant donné que le nombre de processus est connu numériquement, cette représentation par des affectations permet également d'exprimer différents schémas de diffusion d'une variable. Par exemple, un processus unique peut envoyer séquentiellement la valeur d'une variable à chacun des autres processus, comme le montre le Code 2.6. Mais on peut aussi utiliser un autre schéma de communication couvrant tous les processus pour diffuser une valeur tout en utilisant mieux le réseau d'interconnexion, comme le montre le Code 2.7. Ces deux types de communication peuvent être exprimés par un ensemble d'affectations.

```
x_1 = x_0;
x_2 = x_0;
x_3 = x_0;
x_4 = x_0;
```

Code 2.6 – représentation d'une diffusion séquentielle

```
x_1 = x_0;
x_3 = x_0;
x_2 = x_1;
x_4 = x_3;
```

Code 2.7 – représentation d'une diffusion arborescente

La diffusion séquentielle de la valeur de la variable `x` possédée par le processus 0 s'effectue en quatre étapes sérialisées (Code 2.6).

Une diffusion parallèle (aussi appelée en papillon ou arborescente) peut s'effectuer en seulement trois étapes, les deux dernières affectations étant exécutables en parallèle (Code 2.7). Le gain de cette diffusion augmente avec le nombre de processus présents.

Il faut tout de même faire attention à l'équilibre entre *send* et *receive* en présence d'instructions de contrôle ou de transferts asynchrones. Une solution concernant la présence d'instructions de contrôle est discuté en section 2.5.2.

Une représentation plus complexe ou tout au moins une interprétation plus raffinée est faite pour la génération des communications concernant les tableaux ou les structures. Par exemple, il faut éviter d'avoir une boucle qui communique un à un chaque élément d'un tableau. Il est préférable d'avoir une communication unique envoyant et recevant en une seule fois tout le tableau. Pour cela, on pourrait utiliser une représentation de tableau à la Cilk [49], comme `a_1[0:n] = a_2[0:n]`. Cette représentation permet d'éviter d'avoir une boucle itérant sur tous les éléments de tableaux à communiquer et donc d'avoir une traduction directe de tous les éléments qui doivent être communiqués.

Néanmoins, nous préférons, dans le cas des tableaux et des structures, laisser des transformations parallèles faire ces optimisations. Ainsi, par exemple, la génération des communications pour les tableaux devra se passer en deux temps. Dans un premier temps, la copie des éléments de tableau dans une boucle sera traduite par la même boucle mais effectuant des communications. Puis, une optimisation parallèle pourra détecter les boucles qui itèrent sur plusieurs éléments

d'un tableau et donc transformer les boucles de communication en communication unique, en ajoutant au besoin des déclarations de nouveaux type de données dans le cas de communications non adjacentes. De même, si une structure doit être communiquée.

En conclusion, il est donc possible d'exprimer des communications synchrones ou asynchrones, bloquantes ou non bloquantes, et des schémas de diffusion variés avec des séquences affectations.

2.4.5 Conclusion sur la méthodologie retenue

Dans la mesure où les communications du code MPI peuvent être exprimées à l'aide de séquence d'affectations ou de code de copies séquentielles, il est possible de réutiliser des analyses et des transformations définies pour des codes séquentiels, pour effectuer une très grande partie des étapes menant au code distribué. Ainsi, l'utilisation d'un compilateur source-à-source travaillant dans un environnement séquentiel, tel que PIPS, est tout à fait possible.

Il reste à définir une séquence de transformations qui permet de traduire un code séquentiel en une de ses versions distribuées et à déterminer l'étape à partir de laquelle il faut absolument passer à des analyses et des transformations de code parallèle ou distribué.

2.5 Les différents méthodes de modélisation des communications

Comme illustré sur la Figure 2.2 (p. 35), avec les chemins (E) et (F), et avec notre représentation des communications au moyen d'affectations, en section 2.4.4, il existe plusieurs chemins possibles pour passer d'un code séquentiel placé à un code parallèle distribué.

Il est à priori plus judicieux de rester dans le domaine séquentiel afin d'effectuer un maximum d'analyses et de transformations connues, comme le suggère le chemin (E) de la Figure 2.2, puis de réaliser en une transformation unique le passage à un code distribué. Mais nous verrons en section 2.5.1 que cette étape unique n'est pas possible. Dans la section 2.5.2, nous discutons de deux points de vue pour la génération des affectations représentant les communications.

2.5.1 Génération du code distribué par une passe finale ?

Le code distribué est un code MPI qui commence avec un appel à `MPI_Init`. Cet appel correspond à l'initialisation de P processus concurrents. Le code distribué peut être modélisé par une boucle parallèle engobant le code, avec ses communications et synchronisations, avant une transformation finale qui fait appel à des fonctions MPI. Le Code 2.8 illustre le code d'entrée d'une telle transformation, où l'instruction `forall` correspond à `MPI_Init` et la fin de son *scope* à `MPI_Finalize`. Les fonctions `raise` et `wait` correspondent respectivement à `MPI_Send` et `MPI_Recv`.

La transformation la plus simple permettant d'obtenir une boucle parallèle est une parallélisation de boucle. Avant cette parallélisation de boucle, le code d'entrée possède une boucle classique `for` qui va être transformée en une boucle

```
forall(p=0; p<P; p++) {
  if(p==1) {
    a = 1;
    raise(p1);
  }
  if(p==2) {
    b = 2;
    raise(p2);
  }
  if(p==0) {
    wait(p1); wait(p2);
    c = a + b;
  }
}
```

Code 2.8 – boucle parallèle `forall`

```
for(p=0; p<P; p++) {
  if(p==1) {
    a = 1;
    raise(p1);
  }
  if(p==2) {
    b = 2;
    raise(p2);
  }
  if(p==0) {
    wait(p1); wait(p2);
    c = a + b;
  }
}
```

Code 2.9 – représentation séquentielle de la boucle Code 2.8

parallèle `forall`. Dans notre exemple, ce code doit correspondre au Code 2.9. Or le Code 2.9 séquentiel est incorrect. En effet, étant donné que la boucle s'exécute séquentiellement, la première itération, `p=0`, est en attente de `p1` qui est obtenue uniquement avec la deuxième itération de la boucle. Or la deuxième itération ne peut pas s'exécuter avant la fin de la première itération, ce qui engendre un interblocage, *deadlock*. Ce programme ne peut donc jamais fonctionner.

Ainsi, l'ordre séquentiel n'est en général pas compatible avec l'ordre *happens-before* défini par les synchronisations ou les communications contenues dans une boucle parallèle. Comme le montre notre exemple Code 2.9, les synchronisations et *a fortiori* les communications peuvent conduire à une exécution de calculs concurrents débutant par n'importe quelle itération, alors que le code séquentiel commence nécessairement par l'itération 0, et donc le processus 0. On pourrait envisager de trier les itérations afin de pouvoir les sérialiser, mais il n'existe en général pas d'ordre total des itérations car plusieurs points de synchronisation peuvent conduire à un ordre différent des processus.

Le passage du code séquentiel au code parallèle réparti ne peut donc pas s'effectuer systématiquement par une simple parallélisation de boucles. Nous n'avons donc pas trouvé de cheminement de type (E) (Figure 2.2), avec un code distribué résultant d'une dernière passe transformant une boucle séquentielle externe en boucle parallèle et les affectations de copie en paires de *send/receive*.

Nous avons donc décidé de ne pas essayer de partir d'une boucle parallèle pour générer le code parallèle distribué. Mais de nous limiter au placement initial donnée et du graphe de tâches qui en découle pour réaliser notre parallélisation distribuée. Ainsi, certaines optimisations doivent être faites sur du code parallèle distribué, comme par exemple, l'*outlining* de fonctions par processus.

2.5.2 Génération des communications sous forme d'affectations

Dans la section 2.4.4, nous avons montré que nous pouvions simuler les communications sous forme d'affectations entre des variables indexées par des numéros de processus. Dans cette section, nous étudions comment peuvent être

générées ces affectations, et à quel endroit dans le code elles doivent être positionnées pour pouvoir générer nos communications de façon correcte.

2.5.2.1 Génération de communications locales dans une tâche

On souhaite avant tout appliquer des transformations simples et correctes. Afin de conserver la cohérence en mémoire des données, une solution simple consiste, après chaque affectation de variables, à communiquer cette modification à tous les processus. L'idée est de copier la valeur d'une variable modifiée sur les autres processus systématiquement, et cela après chaque instruction d'affectation. Toutefois, cette méthode présente plusieurs inconvénients lors du passage à la parallélisation distribuée : des problèmes de performance d'une part et de correction d'autre part.

Les problèmes de performance peuvent être dus à des communications multiples d'une même variable à d'autres processus ou à des communications inutiles par exemple. Toutefois, des solutions simples peuvent être mises en œuvre pour les résoudre, comme par exemple une élimination de code mort ou inutile. En effet, si au sein d'une même tâche, une variable est modifiée plusieurs fois, seule la dernière valeur sera utile aux autres processus. Étant donné que le code est, à cette étape, séquentiel, toutes les valeurs intermédiaires de la variable n'ont pas besoin d'être connues, et donc transférées aux autres processus.

Les problèmes de correction lors du passage à du code parallèle sont beaucoup plus compliqués à résoudre. En effet, considérant un graphe de flot de contrôle, les branches du programme exécutées ne sont souvent connues que lors de l'exécution du programme. Or parmi les instructions contenues dans ces branches, il peut y avoir des copies de communications. Bien que ces embranchements ne posent pas de problèmes pour le processus émetteur, le processus récepteur des valeurs de ces variables rencontrera des problèmes lorsqu'il devra déterminer la possible réception. Chaque émission de message via un *send* doit avoir une réception *receive* correspondante, or cette information n'est connue que dynamiquement. Le processus récepteur doit également connaître les informations de flot de contrôle, c'est-à-dire qu'il doit soit lui-même les calculer, au détriment du parallélisme ; soit recevoir ces informations de contrôle de la part du processus émetteur, au détriment du nombre de communications. Le programme peut être ralenti avec cette surcharge de communications.

Une solution permettant de régler ces problèmes, est de générer des copies pour chaque affectation mais de les déplacer à la fin de la tâche. Une surapproximation des variables à copier, et donc à communiquer, peut être faite. Mais elle est moins coûteuse qu'une communication systématique du flot de contrôle. La question est de savoir si ce déplacement est faisable, et sous quelles conditions le code reste correct ?

2.5.2.2 Génération de communications globales au niveau des tâches

Au lieu de générer des copies de communication au sein d'une tâche, puis de tenter de les déplacer à la fin de la tâche, pourquoi ne pas directement générer les copies des variables entre les tâches ?

En effet, seule la dernière valeur de chaque variable est utile pour les copies de communications. Étant donné que le code initial est séquentiel, si une variable est écrite au sein d'une tâche puis modifiée par la suite dans cette même tâche,

alors seule sa dernière valeur est utile si jamais elle est utilisée dans une autre tâche par la suite. Ainsi, la cohérence mémoire de notre modélisation distribuée n'a besoin d'être vraiment vérifiée qu'à la fin de chaque tâche.

Entre chaque tâche, surestimer les variables à communiquer, par l'intermédiaire de copies, peut être une solution simple, quitte à ensuite réduire le nombre de copies aux copies utiles. Par exemple, on peut choisir de copier toutes les variables présentes dans la fonction à paralléliser, ou uniquement les variables écrites ou potentiellement écrites au sein de chaque tâche traitée.

Dans un premier temps, il n'est pas nécessaire de déterminer les processus qui auront besoin de la valeur des variables. Une approche conservative consistant à copier chaque variable ayant pu être modifiée au sein d'une tâche à tous les autres processus est adoptée. La suppression des copies inutiles se fera dans un second temps.

2.5.3 Conclusion sur la modélisation des communications

Cette section a mis en évidence qu'il n'est malheureusement pas possible de modéliser notre code parallèle de tâches par une boucle séquentielle globale dont chaque itération aurait représentée l'exécution d'un processus et qui pourrait être traduite en boucle parallèle, comme décrit en section 2.5.1.

De plus, comme expliquer en section 2.5.2, le choix de l'emplacement des copies et donc des communications à générer n'est pas simple, si l'on souhaite considérer le flot de contrôle. Afin de pouvoir réaliser une preuve de correction du code généré, une approche conservative est adoptée. La cohérence des mémoires modélisées est assurée entre chaque tâche pour tous les processus et pour toutes les variables modifiées dans la tâche qui précède.

2.6 Conclusion

Ce chapitre a permis de présenter les différentes problématiques concernant la parallélisation de tâches sur une architecture à mémoire distribuée (section 2.1) : (1) la gestion et la répartition des données sur les différents processus qui vont exécuter les tâches ; (2) la gestion de l'ordonnancement des tâches sur les différents processus, et les instructions que ces processus doivent exécuter. J'ai proposé de résoudre le point (1) par une modélisation de la mémoire des différents processus et leur mise en cohérence telles que présentées dans notre méthodologie en sections 2.4 et 2.5. Le point (2) est résolu implicitement grâce à cette même modélisation et parce que le code initial est séquentiel. Ainsi, l'ordonnancement des tâches est dépendant des communications réalisées.

Un état de l'art des outils de génération de code distribué a été présenté en section 2.2. Ils sont néanmoins limités soit au cas particulier de la parallélisation de boucles (dSTEP, Pluto+), soit par l'apprentissage d'un nouveau langage (HPFC). De plus, leur fonctionnement est souvent une boîte noire ne permettant que difficilement de ré-exploiter ou corriger le code généré.

Nos différents choix, hypothèses et priorités concernant la parallélisation, autres que le fait que ce soit de la parallélisation de tâches sur mémoire distribuée que l'on souhaite effectuer, ont été présentés en section 2.3. L'accent a particulièrement été mis sur la correction de la génération du code parallèle distribué. La possibilité d'un nombre dynamique de processus a été écartée, et

j'ai fait le choix d'une succession "simple" de transformations pour obtenir notre code parallèle (section 2.4).

La Figure 2.3 permet d'illustrer deux chemins de compilation permettant d'obtenir le code parallèle distribué souhaité. Ils illustrent les réflexions faites en section 2.5.2. Cette succession de transformations est présentée au chapitre 3 et les preuves associées au chapitre 4.

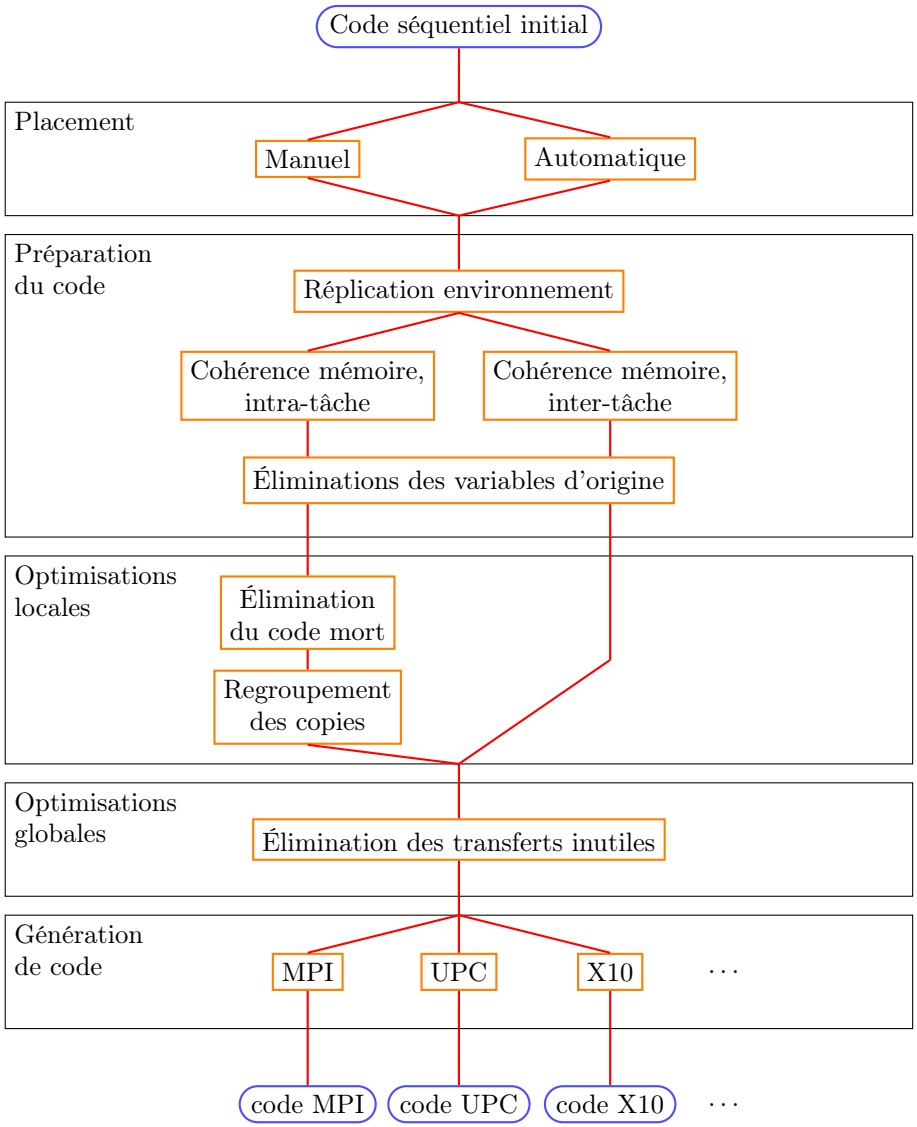


FIGURE 2.3 – Schéma des différentes possibilités de compilation étudiées jusqu'à l'obtention d'un code parallèle

Chapitre 3

Transformations successives du code

Sommaire

Après avoir analysé les avantages et inconvénients des différentes solutions de génération de code distribué pour des tâches au chapitre 2, nous allons dans ce chapitre décrire les différentes transformations successives que nous avons mises en place pour obtenir notre solution. Ce chapitre est corrélé au suivant qui

présente les preuves de correction de ces transformations. De même, diverses optimisations faites ou à faire seront décrites dans le chapitre 5. Ce chapitre correspond en partie à mon article [3], auquel une syntaxe et une sémantique ont été ajoutées.

La syntaxe et la sémantique utilisées dans la suite de cette thèse sont décrites en section 3.1. La détection et le placement des différentes tâches sont rapidement décrits dans la section 3.2. Puis, les différentes phases de transformations sont décrites successivement dans les sections 3.3 à 3.5 : la préparation du code séquentiel avant la mise en parallèle, les optimisations simples à faire pour limiter au minimum les communications, et la transformation du code séquentiel en code parallèle. Ces différentes phases de compilation sont résumées avec la Figure 3.1.

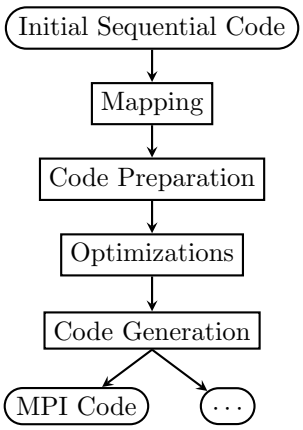


FIGURE 3.1 – Schéma des différentes phases de compilation

3.1 La syntaxe et la sémantique du code

Cette première section présente la syntaxe puis la sémantique du langage utilisé pour exprimer nos programmes et nos transformations. Elles sont ensuite utilisées pour réaliser nos preuves.

3.1.1 Syntaxe

La syntaxe suivante décrit une fonction séquentielle pour laquelle on souhaite effectuer une parallélisation de tâches sur une architecture à mémoire distribuée. Cette syntaxe utilise une notation BNF [13].

$$
\begin{array}{lll}
\langle \textit{function} \rangle & ::= \langle \textit{loc-declaration} \rangle \ (\ \langle \textit{loc-declaration} \rangle^{*}\ ) & \\
 & \quad \{\ \langle \textit{block} \rangle\ ;\ [\text{return}\ \langle \textit{expression} \rangle\ ;]\ \} & \\
\langle \textit{block} \rangle & ::= \varnothing \mid \langle \textit{block} \rangle\ ;\ \langle \textit{block} \rangle & \\
 & \quad \mid \langle \textit{fun-declaration} \rangle & \\
 & \quad \mid (\langle \textit{statements} \rangle, p\ ) & p \in \mathbb{N} \\
\langle \textit{statements} \rangle & ::= \varnothing \mid \langle \textit{statement} \rangle\ ;\ \langle \textit{statements} \rangle & \\
\langle \textit{statement} \rangle & ::= \langle \textit{loc-declaration} \rangle & \\
 & \quad \mid \langle \textit{affectation} \rangle & \\
 & \quad \mid \text{if}\ \langle \textit{expression} \rangle\ \text{then}\ \langle \textit{statements} \rangle\ [\text{else}\ \langle \textit{statements} \rangle] & \\
 & \quad \mid \text{while}\ \langle \textit{expression} \rangle\ \text{do}\ \langle \textit{statements} \rangle & \\
 & \quad \mid \text{for}\ (\ \langle \textit{affectation} \rangle\ ;\ \langle \textit{expression} \rangle\ ;\ \langle \textit{affectation} \rangle\ )\ \langle \textit{statements} \rangle & \\
\langle \textit{loc-declaration} \rangle & ::= \langle \textit{type} \rangle\ \textit{loc-identifier} & \\
\langle \textit{fun-declaration} \rangle & ::= \langle \textit{type} \rangle\ \textit{fun-identifier} & \\
\langle \textit{affectation} \rangle & ::= \langle \textit{variable} \rangle \leftarrow \langle \textit{expression} \rangle & \\
\langle \textit{expression} \rangle & ::= \langle \textit{variable} \rangle & \\
 & \quad \mid \textit{constant} & \\
 & \quad \mid \text{true} \mid \text{false} & \\
 & \quad \mid \langle \textit{unary-op} \rangle\ \langle \textit{expression} \rangle & \\
 & \quad \mid \langle \textit{expression} \rangle\ \langle \textit{binary-op} \rangle\ \langle \textit{expression} \rangle & \\
 & \quad \mid \langle \textit{expression} \rangle\ \langle \textit{relational-op} \rangle\ \langle \textit{expression} \rangle & \\
\langle \textit{variable} \rangle & ::= \langle \textit{identifier} \rangle & \\
 & \quad \mid \langle \textit{identifier} \rangle.\langle \textit{identifier} \rangle & \\
 & \quad \mid \langle \textit{identifier} \rangle[\ \langle \textit{index} \rangle\ ] & \\
\langle \textit{identifier} \rangle & ::= \textit{loc-identifier} \mid \textit{fun-identifier} & \\
\langle \textit{index} \rangle & ::= \langle \textit{identifier} \rangle \mid n & n \in \mathbb{N}
\end{array}
$$

$$
\begin{array}{lll}
\langle \textit{int-type} \rangle & ::= (\text{unsigned} \mid \text{signed})\ (\text{char} \mid \text{short} \mid \text{int} \mid \text{long}) & \\
\langle \textit{float-type} \rangle & ::= \text{float} \mid \text{double} & \\
\langle \textit{scalar-type} \rangle & ::= \langle \textit{int-type} \rangle \mid \langle \textit{float-type} \rangle & \\
\langle \textit{type} \rangle & ::= \langle \textit{scalar-type} \rangle & \\
 & \quad \mid \langle \textit{scalar-type} \rangle[\ n\ ] & n \in \mathbb{N}^{*} \\
 & \quad \mid \text{struct}\{\langle \textit{identifier} \rangle_0 : \langle \textit{scalar-type} \rangle_0, \ldots, & \\
 & \qquad \langle \textit{identifier} \rangle_{n-1} : \langle \textit{scalar-type} \rangle_{n-1}\} & \\
\langle \textit{unary-op} \rangle & ::= - \mid \sim & \\
\langle \textit{binary-op} \rangle & ::= + \mid - \mid * \mid / \mid \% & \\
\langle \textit{relational-op} \rangle & ::= == \mid \neq \mid \leq \mid \geq \mid < \mid > &
\end{array}
$$

Les mots clefs de cette syntaxe sont présentés en rouge, à l'exception des opérateurs qui sont restés en noir et les mots terminaux en vert.

On peut noter dans cette syntaxe la différenciation entre ⟨*loc-declaration*⟩ et ⟨*fun-declaration*⟩. ⟨*fun-declaration*⟩ est la déclaration d'une variable qui a une portée, un *scope*, au niveau de la fonction entière, alors que ⟨*loc-declaration*⟩ est la déclaration d'une variable dont la portée est plus locale, au niveau d'une séquence d'instructions, ⟨*statements*⟩. De ce fait, on effectue une distinction entre ⟨*block*⟩ et ⟨*statements*⟩. ⟨*block*⟩ ne définit que des instructions de premier niveau par rapport à la fonction. Ces dernières sont soit une déclaration ⟨*fun-declaration*⟩, au niveau de la fonction, soit une séquence d'instructions ⟨*statements*⟩ affectée à un processus. Notre syntaxe considère ainsi que la phase de détection et de placement des tâches sur les processus a déjà été effectuée. ⟨*statements*⟩ peut correspondre à la déclaration d'une séquence d'instructions

$\langle statement \rangle$ avec une nouvelle portée locale. Ces dernières peuvent être une déclaration locale $\langle loc\text{-}declaration \rangle$, une affectation $\leftarrow$, un test if, une boucle while ou for. La boucle for n'est pas aussi générale que la traditionnelle boucle `for` du langage C. Les types possibles correspondent aux principaux types de C à l'exception des pointeurs.

Dans la suite de la thèse, le terme variable et identifiant désigneront un $\langle identifier \rangle$ de notre syntaxe. Les crochets seront présents lorsqu'il s'agit d'une $\langle variable \rangle$ de la syntaxe.

Cette syntaxe définit un sous-ensemble du langage C simplifié dans lequel on distingue explicitement la portée des variables relatives à la fonction ou locales à un $\langle statements \rangle$.

3.1.2 Sémantique séquentielle

La sémantique utilisée est une sémantique dénotationnelle [41]. On suppose que le programme est bien typé. Cela permet de simplifier la description de notre sémantique à plusieurs endroits. Le symbole représentant une erreur ε est toujours présent dans notre sémantique mais il est de ce fait beaucoup moins souvent utilisé.

On définit un emplacement mémoire, notée $\mathcal{L}$, comme un couple (identifiant, index) :

$$\mathcal{L} : \langle identifier \rangle \times \mathbb{N} \cup \varepsilon$$

Les variables scalaires ont un index toujours égale à 0. Les fonctions $indexof$ et $typeof$ permettent respectivement de retourner la position d'un élément dans une structure et de renvoyer le type d'une variable.

$$typeof : \langle identifier \rangle \rightarrow \langle type \rangle$$
$$indexof : \langle type \rangle \times \langle identifier \rangle \rightarrow \mathbb{N}$$

On définit la fonction d'état mémoire *Store* qui associe un emplacement mémoire $\mathcal{L}$ à une valeur *Value*. *Value* peut prendre n'importe quelle valeur entière ou réelle ainsi que $\perp$ et ε. $\perp$ représente une valeur non assignée. ε représente une erreur. On aura :

$$Value : \mathbb{N} \cup \mathbb{R} \cup \perp \cup \varepsilon$$
$$Store : \mathcal{L} \rightarrow Value$$

On définit une fonction sur les $\langle variable \rangle$, notée $\mathbb{L}$, qui prend une $\langle variable \rangle$ et un état mémoire *Store* en entrée et retourne un emplacement mémoire $\mathcal{L}$

$$\mathbb{L} : \langle variable \rangle \rightarrow Store \rightarrow \mathcal{L}$$

Elle est définie comme suit :

$$\sigma \in Store,\ id_1, id_2 \in \langle identifier \rangle,\ n \in \mathbb{N}$$
$$\mathbb{L}[\![id_1]\!]\langle\sigma\rangle = (id_1, 0) \tag{3.1}$$
$$\mathbb{L}[\![id_1[n]]\!]\langle\sigma\rangle = (id_1, n) \tag{3.2}$$
$$\mathbb{L}[\![id_1[id_2]]\!]\langle\sigma\rangle = (id_1, \sigma(id_2, 0)) \tag{3.3}$$
$$\mathbb{L}[\![id_1.id_2]\!]\langle\sigma\rangle = (id_1, indexof(typeof(id_1), id_2)) \tag{3.4}$$

Rappel : Comme on suppose que le programme est bien typé, dans l'Équation 3.3 concernant la sémantique d'un tableau $id_1[id_2]$, id_2 doit être un entier inférieur

à la taille du tableau id_1. Sans cette hypothèse il faudrait ajouter des tests supplémentaires, qui auraient alourdis la sémantique, pour renvoyer des erreurs de typage éventuelles.

De plus, on définit $\langle \textit{loc-variable} \rangle$, respectivement $\langle \textit{fun-variable} \rangle$, un sous-ensemble de $\langle \textit{variable} \rangle$ tel que l'identifiant terminal appartient à *loc-identifer*, respectivement *fun-identifier*.

On définit une fonction sur les opérateurs, notée $\mathbb{O}$. Cette fonction s'applique sur un opérateur unaire, binaire ou relationnel, et prend comme argument une valeur ou un couple de valeurs et renvoie une nouvelle valeur correspondant au résultat de cet opérateur. Elle a la forme :

$$\mathbb{O} : \langle \textit{unary-op} \rangle \cup \langle \textit{binary-op} \rangle \cup \langle \textit{relational-op} \rangle \to (\textit{Value} \cup \textit{Value} \times \textit{Value}) \to \textit{Value}$$

Cette fonction est assez intuitive et ne sera pas détaillée. En voici quelques exemples : $\mathbb{O}[\![+]\!]\langle(40, 2)\rangle = 42$, $\mathbb{O}[\![==]\!]\langle(0, 1)\rangle = \text{false}$, $\mathbb{O}[\![-]\!]\langle 1\rangle = -1$.

La sémantique des expressions, notée $\mathbb{E}$, a la forme :

$$\mathbb{E} : \langle \textit{expression} \rangle \to \textit{Store} \to \textit{Value}$$

Elle est définie comme suit :

$$\begin{aligned}
&\sigma \in \textit{Store},\ \textit{var} \in \langle \textit{variable} \rangle,\ e_1, e_2 \in \langle \textit{expression} \rangle,\\
&\textit{uop} \in \langle \textit{unary-op} \rangle,\ \textit{bop} \in \langle \textit{binary-op} \rangle,\ \textit{rop} \in \langle \textit{relational-op} \rangle\\
&\mathbb{E}[\![\textit{const}]\!]\langle\sigma\rangle = \textit{const} && (3.5)\\
&\mathbb{E}[\![\text{true}]\!]\langle\sigma\rangle = \text{true} \qquad \mathbb{E}[\![\text{false}]\!]\langle\sigma\rangle = \text{false} && (3.6)\\
&\mathbb{E}[\![\textit{var}]\!]\langle\sigma\rangle = \sigma(\mathbb{L}[\![\textit{var}]\!]\langle\sigma\rangle) && (3.7)\\
&\mathbb{E}[\![\textit{uop}\ e_1]\!]\langle\sigma\rangle = \mathbb{O}[\![\textit{uop}]\!]\langle\mathbb{E}[\![e_1]\!]\langle\sigma\rangle\rangle && (3.8)\\
&\mathbb{E}[\![e_1\ \textit{bop}\ e_2]\!]\langle\sigma\rangle = \mathbb{O}[\![\textit{bop}]\!]\langle(\mathbb{E}[\![e_1]\!]\langle\sigma\rangle, \mathbb{E}[\![e_2]\!]\langle\sigma\rangle)\rangle && (3.9)\\
&\mathbb{E}[\![e_1\ \textit{rop}\ e_2]\!]\langle\sigma\rangle = \mathbb{O}[\![\textit{rop}]\!]\langle(\mathbb{E}[\![e_1]\!]\langle\sigma\rangle, \mathbb{E}[\![e_2]\!]\langle\sigma\rangle)\rangle && (3.10)
\end{aligned}$$

La sémantique des instructions, notée $\mathbb{S}$, a la forme :

$$\mathbb{S} : \langle \textit{block} \rangle \cup \langle \textit{statements} \rangle \to \textit{Store} \to \textit{Store}$$

Elle utilise la fonction *update* permettant de mettre à jour un état mémoire. Cette dernière prend en argument un état mémoire, un emplacement mémoire et une nouvelle valeur et renvoie un nouvel état mémoire.

$$\begin{aligned}
&\textit{update} : \textit{Store} \times \mathcal{L} \times \textit{Value} \to \textit{Store}\\
&\textit{update}(\sigma, (id, n), v) = \lambda x.\ \text{if } x = (id, n) && (3.11)\\
&\qquad\qquad\qquad\qquad \text{then } \mathbb{E}[\![v]\!]\langle\sigma\rangle\\
&\qquad\qquad\qquad\qquad \text{else } \sigma(x)
\end{aligned}$$

La sémantique séquentielle des instructions est définie comme suit :

$$\sigma \in \mathit{Store},\ s_1, s_2 \in \langle \mathit{statements}\rangle,\ \mathit{init}, \mathit{inc} \in \langle \mathit{affectation}\rangle,\ p \in \mathbb{N},$$
$$\mathit{lhs} \in \langle \mathit{variable}\rangle,\ e, \mathit{cond} \in \langle \mathit{expression}\rangle,\ t \in \langle \mathit{type}\rangle,\ \mathit{id}_1 \in \langle \mathit{identifier}\rangle$$

$$\mathbb{S}[\![\varnothing]\!]\langle\sigma\rangle = \sigma \tag{3.12}$$
$$\mathbb{S}[\![s_1\,;\ s_2]\!]\langle\sigma\rangle = \mathbb{S}[\![s_2]\!]\langle\mathbb{S}[\![s_1]\!]\langle\sigma\rangle\rangle \tag{3.13}$$
$$\mathbb{S}[\![(s_1, p)]\!]\langle\sigma\rangle = \mathbb{S}[\![s_1]\!]\langle\sigma\rangle \tag{3.14}$$
$$\mathbb{S}[\![\mathit{lhs} \leftarrow\ e]\!]\langle\sigma\rangle = \mathit{update}(\sigma, \mathbb{L}[\![\mathit{lhs}]\!]\langle\sigma\rangle, \mathbb{E}[\![e]\!]\langle\sigma\rangle) \tag{3.15}$$
$$\mathbb{S}[\![\text{if } \mathit{cond} \text{ then } s_1 \text{ else } s_2]\!]\langle\sigma\rangle = \text{ if } \mathbb{E}[\![\mathit{cond}]\!]\langle\sigma\rangle == \text{true} \text{ then } \mathbb{S}[\![s_1]\!]\langle\sigma\rangle \text{ else } \mathbb{S}[\![s_2]\!]\langle\sigma\rangle \tag{3.16}$$
$$\mathbb{S}[\![\text{while } \mathit{cond} \text{ do } s]\!]\langle\sigma\rangle = \text{ if } \mathbb{E}[\![\mathit{cond}]\!]\langle\sigma\rangle == \text{true} \text{ then } \mathbb{S}[\![\text{while } \mathit{cond} \text{ do } s]\!]\langle\mathbb{S}[\![s]\!]\langle\sigma\rangle\rangle \text{ else } \sigma \tag{3.17}$$
$$\mathbb{S}[\![\text{for}(\mathit{init}\,;\mathit{cond}\,;\mathit{inc})\ s]\!]\langle\sigma\rangle = \mathbb{S}[\![\mathit{init}; \text{while } \mathit{cond} \text{ do } \{s; \mathit{inc}\}]\!]\langle\sigma\rangle \tag{3.18}$$
$$\mathbb{S}[\![t\ \mathit{id}_1]\!]\langle\sigma\rangle = \mathit{update}(\sigma, (\mathit{id}_1, \_), \bot) \tag{3.19}$$

L'hypothèse d'un typage correct du programme permet de simplifier la sémantique en supprimant la nécessité de vérifier la cohérence du typage de la partie gauche et de la partie droite des affectations, Équation 3.15. De même, la vérification des conditions du test, Équation 3.16 et des boucles, Équations 3.17 et 3.18, n'est pas nécessaire.

Par ailleurs, pour améliorer la lisibilité, le cas de la déclaration d'une variable, Équation 3.19, a été simplifié. On aurait dû avoir :

$$\mathbb{S}[\![t\ \mathit{id}_1]\!]\langle\sigma\rangle = \text{ if } t \in \langle \mathit{scalar\text{-}type}\rangle\ \mathit{update}(\sigma, (\mathit{id}_1, 0), \bot) \text{ otherwise } \lambda(x, i).\ \text{if } x = \mathit{id}_1 \wedge 0 \leq i < \mathit{sizeof}(t) \text{ then } \bot \text{ else } \sigma(x, i)$$

où *sizeof* donne la taille d'un tableau ou le nombre d'éléments dans une structure.

Les séquences de ⟨*block*⟩ et de ⟨*statements*⟩ sont sémantiquement similaires et sont définies par l'Équation 3.13. Les déclarations ⟨*fun-declaration*⟩ et ⟨*loc-declaration*⟩ sont également similaires et correspondent à l'Équation 3.19.

3.1.3 Transformation de récursion structurelle

Les sections suivantes décrivent des transformations de programme à partir notre syntaxe. Ces transformations sont définies par $\mathcal{T}_{\mathbb{E}\ldots}$ pour des transformations des expressions ou par $\mathcal{T}_{\mathbb{S}\ldots}$ pour des transformations des instructions. La Transformation 3.1 représente une identité. Dans la suite, seules les équations des transformations différentes de cette dernière sont données afin d'alléger leurs définitions

Transformation 3.1 (Transformation de récursion structurelle)**.** Cette transformation effectue une identité sur le programme. Elle est définie par les $\mathcal{T}_{\mathbb{E}}$ et $\mathcal{T}_{\mathbb{S}}$ suivantes :

$$id_1, id_2 \in \langle identifier \rangle,\ e_1, e_2 \in \langle expression \rangle,$$
$$uop \in \langle unary\text{-}op \rangle,\ bop \in \langle binary\text{-}op \rangle,\ rop \in \langle relational\text{-}op \rangle$$

$$\mathcal{T}_{\mathbb{E}}[\![const]\!] = const \tag{3.20}$$
$$\mathcal{T}_{\mathbb{E}}[\![\text{true}]\!] = \text{true} \qquad \mathcal{T}_{\mathbb{E}}[\![\text{false}]\!] = \text{false} \tag{3.21}$$
$$\mathcal{T}_{\mathbb{E}}[\![id_1]\!] = id_1 \tag{3.22}$$
$$\mathcal{T}_{\mathbb{E}}[\![id_1[id_2]]\!] = \mathcal{T}_{\mathbb{E}}[\![id_1]\!][\mathcal{T}_{\mathbb{E}}[\![id_2]\!]] \tag{3.23}$$
$$\mathcal{T}_{\mathbb{E}}[\![id_1.id_2]\!] = \mathcal{T}_{\mathbb{E}}[\![id_1]\!].\mathcal{T}_{\mathbb{E}}[\![id_2]\!] \tag{3.24}$$
$$\mathcal{T}_{\mathbb{E}}[\![uop\ e_1]\!] = uop\ \mathcal{T}_{\mathbb{E}}[\![e_1]\!] \tag{3.25}$$
$$\mathcal{T}_{\mathbb{E}}[\![e_1\ bop\ e_2]\!] = \mathcal{T}_{\mathbb{E}}[\![e_1]\!]\ bop\ \mathcal{T}_{\mathbb{E}}[\![e_2]\!] \tag{3.26}$$
$$\mathcal{T}_{\mathbb{E}}[\![e_1\ rop\ e_2]\!] = \mathcal{T}_{\mathbb{E}}[\![e_1]\!]\ rop\ \mathcal{T}_{\mathbb{E}}[\![e_2]\!] \tag{3.27}$$

$b_1, b_2 \in \langle block \rangle,\ s_1, s_2 \in \langle statements \rangle,\ lhs \in \langle variable \rangle,\ e, cond \in \langle expression \rangle,$
$init, inc \in \langle affectation \rangle,\ p \in \mathbb{N},\ t \in \langle type \rangle,\ id \in \langle identifier \rangle$

$$\mathcal{T}_{\mathbb{S}}[\![\varnothing]\!] = \varnothing \tag{3.28}$$
$$\mathcal{T}_{\mathbb{S}}[\![b_1; b_2]\!] = \mathcal{T}_{\mathbb{S}}[\![b_1]\!]; \mathcal{T}_{\mathbb{S}}[\![b_2]\!] \tag{3.29}$$
$$\mathcal{T}_{\mathbb{S}}[\![(s_1, p)]\!] = (\mathcal{T}_{\mathbb{S}}[\![s_1]\!], p) \tag{3.30}$$
$$\mathcal{T}_{\mathbb{S}}[\![s_1; s_2]\!] = \mathcal{T}_{\mathbb{S}}[\![s_1]\!]; \mathcal{T}_{\mathbb{S}}[\![s_2]\!] \tag{3.31}$$
$$\mathcal{T}_{\mathbb{S}}[\![\text{if } cond \text{ then } s_1 \text{ else } s_2]\!] = \text{if } \mathcal{T}_{\mathbb{E}}[\![cond]\!] \text{ then } \mathcal{T}_{\mathbb{S}}[\![s_1]\!] \text{ else } \mathcal{T}_{\mathbb{S}}[\![s_2]\!] \tag{3.32}$$
$$\mathcal{T}_{\mathbb{S}}[\![\text{while } cond \text{ do } s_1]\!] = \text{while } \mathcal{T}_{\mathbb{E}}[\![cond]\!] \text{ do } \mathcal{T}_{\mathbb{S}}[\![s_1]\!] \tag{3.33}$$
$$\mathcal{T}_{\mathbb{S}}[\![\text{for}(init, cond, inc)\ s_1]\!] = \text{for}(\mathcal{T}_{\mathbb{S}}[\![init]\!], \mathcal{T}_{\mathbb{E}}[\![cond]\!], \mathcal{T}_{\mathbb{S}}[\![inc]\!])\ \mathcal{T}_{\mathbb{S}}[\![s_1]\!] \tag{3.34}$$
$$\mathcal{T}_{\mathbb{S}}[\![lhs \leftarrow e]\!] = \mathcal{T}_{\mathbb{E}}[\![lhs]\!] \leftarrow \mathcal{T}_{\mathbb{E}}[\![e]\!] \tag{3.35}$$
$$\mathcal{T}_{\mathbb{S}}[\![t\ id]\!] = t\ \mathcal{T}_{\mathbb{E}}[\![id]\!] \tag{3.36}$$

3.2 La détection et le placement des différentes tâches

On suppose que le placement ne peut être défini que pour des instructions se trouvant “au premier niveau” de la fonction. Ainsi, une tâche, *ie.* une instruction placée, ne peut pas se trouver à l’intérieur du corps d’une boucle ou d’un test. Des transformations de boucles telles que du déroulement de boucle, *loop unrolling*, ou du clivage de boucle, *loop splitting*, peuvent toutefois être appliquées et permettre la parallélisation du corps d’une boucle en tant que différentes tâches sur différents processus.

La définition d’une tâche au sein du code source s’effectue au moyen de directives `pragma`. L’avantage des directives est que le placement des tâches reste transparent quand à la sémantique du code. Celles-ci peuvent tout simplement être ignorées.

Un `pragma distributed` permet de définir les tâches distribuables. Cette directive est accompagnée d'une information de placement `on_cluster`. Elle indique sur quel processus on souhaite exécuter la tâche. Contrairement aux autres outils de parallélisation de tâches, tels qu'OpenMP [61] ou OmpSs [24][100], qui ont besoin explicitement des informations de dépendances entre les différentes tâches pour effectuer un ordonnancement dynamique efficace, notre processus de parallélisation ne les requiert pas explicitement. Plus précisément, ces informations de dépendances sont calculées durant celui-ci. Par contre, nous avons besoin de connaître le processus sur lequel la tâche doit être exécutée. Cette directive unique permet d'offrir une interface simple et minimale de programmation pour définir des tâches dans un programme.

La détection et le placement des tâches peuvent être faits soit manuellement, soit automatiquement via un ordonnanceur statique. Par exemple, on peut utiliser l'ordonnanceur de tâches BDSC développé par Khaldi et al. [57]. L'utilisateur peut également mettre lui-même ces directives dans le code, et librement définir ses tâches ainsi que le processus qui doit l'exécuter.

Ce placement des tâches est représenté dans notre syntaxe par $(\langle statements \rangle, p)$ où p représente le processus qui exécutera la tâche. p correspond donc à la valeur passée à `on_cluster`. De plus, syntaxiquement, $(\langle statements \rangle, p)$ ne peut être placé qu'au premier niveau de la fonction comme notre placement l'exige.

3.3 La préparation du code séquentiel

Cette phase consiste à préparer la modélisation du code parallèle avec du code séquentiel. Pour cela, de nouvelles variables sont ajoutées. Elles représentent les variables initiales, ayant une portée globale à la fonction, associées à chacun des processus. Quatre passes de transformations sont appliquées dans cette phase. Elles correspondent aux transformations présentes dans le groupe "Préparation du code" de la Figure 2.3 (p. 43), avec une étape de nettoyage de code en plus.

3.3.1 Réplication de l'environnement

La première transformation consiste à répliquer l'environnement. Pour cela, de nouvelles déclarations de variables sont faites. Cette transformation est décrite avec la Transformation 3.5.

On définit dans un premier temps l'ensemble des variables d'origine, en Définition 3.2, et l'ensemble de nos variables fraîches ou variables répliquées, en Définition 3.4. L'obtention des variables fraîches est donnée par la Définition 3.3.

Définition 3.2 (Ensemble des variables d'origine)**.** L'ensemble des variables (identifiants) ayant pour portée la fonction est noté Id_0.

$$Id_0 \stackrel{\text{def}}{=} dom(\sigma_0) \cap \mathit{fun\text{-}identifier}$$

avec σ_0 l'état mémoire à la fin de la fonction d'origine à transformer,
et *dom* la fonction qui renvoie le domaine d'une fonction.

Définition 3.3 (Variable fraîche)**.** La fonction *idf* permet de générer des variables fraîches. Elle prend en entrée un couple composé d'un identifiant appartenant à l'ensemble Id_0 et un entier inférieur à P, le nombre maximum de processus. Elle génère une nouvelle variable fraîche dans *fun-identifier*.

$$idf : (Id_0, \mathbb{N}) \rightarrow \textit{fun-identifier}$$

idf est injective et inversible. Elle a les propriétés suivantes :

$$\begin{aligned} \forall\, id \in Id_0\ \forall p \in [0..P[, &\quad idf(id,p) \notin Id_0 \\ \forall\, id, id' \in Id_0\ \forall p, p' \in [0..P[, &\quad id \neq id' \vee p \neq p' \Rightarrow idf(id,p) \neq idf(id',p') \\ \forall\, id' \in \textit{fun-identifier}, &\quad id' \in Id_0 \vee \exists!(id,p)\ idf(id,p) = id' \end{aligned}$$

Par abus, on utilisera la notation id_p pour représenter l'identifiant renvoyé par $idf(id,p)$ avec $id \in Id_0$ et $p \in [0..P[$.

Définition 3.4 (Ensemble des variables fraîches)**.** L'ensemble des variables fraîches ou répliquées créées est noté Id_{new}. Il correspond à l'image de la fonction *idf*.

$$Id_{new} \stackrel{\text{def}}{=} im(idf)$$

Transformation 3.5 (Réplication des déclarations)**.** La transformation permettant de répliquer l'environnement pour un nombre de processus P est appelée la transformation de réplication des déclarations, notée $\mathcal{T}_{\mathbb{S}_{ddd}}$. Elle ne transforme que les déclarations appartenant à *fun-declaration*.

$$\begin{aligned} &t \in \langle \textit{type} \rangle,\ \textit{fid} \in \textit{fun-identifier} \\ &\qquad \mathcal{T}_{\mathbb{S}_{ddd}}[\![t\ \textit{fid}]\!] = t\ \textit{fid}\,;t\ \textit{fid}_0\,;\ldots;t\ \textit{fid}_{P-1} \end{aligned} \tag{3.37}$$

Une transformation structurelle, comme décrit en Transformation 3.1, sera appliquée aux autres éléments de notre sémantique.
Concernant le code généré, nous avons choisi de préfixer l'identifiant par `__` et de le suffixer par `_p` et de vérifier l'absence de conflit de nom.

À cette étape, aucune instruction, autre que les déclarations introduites, ne fait usage de ces nouveaux symboles. La sémantique du nouveau programme après cette transformation est identique à la sémantique du programme initial. Néanmoins, le domaine de définition de σ est étendu, *ie.* les emplacements mémoires de $\mathcal{L}$ ayant une valeur ($\neq \varepsilon$) par σ sont étendus aux nouvelles variables déclarées.

3.3.2 Maintien de la cohérence mémoire

L'objectif de cette étape est de faire en sorte que tous les nouveaux identificateurs aient la même valeur que leur identificateur d'origine. Une fonction S_{cp}, permettant de conserver la cohérence mémoire entre une variable d'origine et l'ensemble de ses variables répliquées, est définie dans la Définition 3.6.

Définition 3.6 (Instructions de copie mémoire totale)**.** On définit la fonction S_{cp} qui renvoie une séquence d'instructions de copies à partir d'une variable ayant pour portée la fonction.

$$S_{cp} : \textit{fun-identifier} \to \langle \textit{statements} \rangle$$

L'identifiant doit appartenir à Id_0. Les copies de la valeur de la variable d'origine sont faites pour chaque variable répliquée.

Par exemple, si $id \in Id_0$ est un scalaire :

$$S_{cp}(id) = \bigcup_{0 \leq p < P} id_p \leftarrow id\,;$$

si $id \in Id_0$ est un tableau :

$$S_{cp}(id) = \text{int } i_m; \bigcup_{0 \leq p < P} \text{for}(i_m \leftarrow 0;\ i_m < sizeof(id);\ i_m \leftarrow i_m + 1)\ id_p[i_m] \leftarrow id[i_m];$$

De façon similaire pour chaque champs d'une structure.

Remarque 3.1. Le symbole $\cup$ représente une séquence d'instructions.
Par exemple : $\bigcup_{0 \leq p < P} id_p \leftarrow id\,; = id_0 \leftarrow id\,;\ \ldots\ ; id_{P-1} \leftarrow id\,;$

La cohérence mémoire s'exprime avec la Propriété 3.1 suivante :

Propriété 3.1 (Cohérence mémoire entre la variable d'origine et les variables répliquées)**.** *La cohérence mémoire entre la valeur d'une variable d'origine et la valeur des variables répliquées est vérifiée pour un état mémoire σ donné si :*

$$\forall\, id \in Id_0\ \forall\, p \in [0..P[, \quad \sigma(id, \_) = \sigma(id_p, \_)$$

La Transformation 3.7 permet d'assurer la cohérence mémoire (Propriété 3.1) entre chacune des tâches. On pourrait vouloir imposer la cohérence après chaque instruction. Mais comme expliqué en section 2.5.2, des problèmes d'indéterminisme pourraient apparaître lors de la génération des communications. Le fait de les grouper à la fin de la tâche, quitte à avoir une sur-approximation, simplifie grandement les preuves qui sont données dans le chapitre suivant.

Transformation 3.7 (Cohérence complète de la mémoire entre tâches)**.** Les tâches sont représentées dans notre syntaxe par $(\langle \textit{statements} \rangle, p)$. Étant donnée qu'on ne souhaite vérifier la cohérence mémoire qu'entre tâches, seuls les $\langle \textit{block} \rangle$ de tâche sont transformés :

$$b \in (\langle \textit{statements} \rangle, p), \qquad \mathcal{T}_{\mathbb{S}_{com}}[\![b]\!] = b\,;\ (\bigcup_{id \in Id_0} S_{cp}(id), p) \tag{3.38}$$

Pour être totalement corrects, les id ne sont pas forcément tous ceux présents dans Id_0, mais tous ceux qui sont dans Id_0 et qui ont été déclarés à l'instant où le $\langle \textit{block} \rangle$ est exécuté. Il faudrait donc normalement avoir l'état mémoire σ à cette instant et avoir $id \in Id_0 \cap dom(\sigma)$.

La Transformation 3.7 peut néanmoins entraîner des redondances de copies. En effet, elle ne considère pas l'historique d'exécution du programme et le fait que l'on puisse déjà avoir une cohérence mémoire entre les variables d'origine et leurs variables répliquées. C'est le cas lorsque la tâche n'a pas modifié une variable. Ainsi, en considérant que cette cohérence est déjà garantie avant l'exécution d'une tâche, la Transformation 3.7 peut être raffinée par la Transformation 3.8. La Transformation 3.8 ne considère que les variables qui ont pu être modifiées au sein d'une tâche. Elle utilise la fonction $\overline{WRITE}$ permettant d'obtenir l'ensemble des variables potentiellement modifiées par une tâche. $\overline{WRITE}$ vérifie la Propriété 3.2 attestant que les valeurs des variables n'appartenant pas à $\overline{WRITE}$ restent inchangées avant et après l'exécution d'une tâche.

Propriété 3.2 (Ensemble des variables écrites par une tâche)**.** *$\overline{WRITE}$ est une fonction prenant en argument une tâche $(\langle statements \rangle, p)$, et renvoyant l'ensemble des variables pouvant être modifiées par cette dernière. Toutes les valeurs des variables absentes $\overline{WRITE}$ ne sont pas modifiées.*

$$\begin{aligned} &b \in (\langle statements \rangle, p),\ \sigma \in Store, \\ &\quad \forall id \notin \overline{WRITE}(b),\quad \sigma(id) = \mathbb{S}[\![b]\!]\langle\sigma\rangle(id) \end{aligned} \tag{3.39}$$

Transformation 3.8 (Cohérence de la mémoire entre tâches)**.** Les tâches sont représentées dans notre syntaxe par $(\langle statements \rangle, p)$. Étant donné qu'on ne souhaite garantir la cohérence mémoire qu'entre chaque tâche, seuls les $\langle block \rangle$ de tâche sont transformés :

$$b \in (\langle statements \rangle, p), \qquad \mathcal{T}_{\mathbb{S}_{com}}[\![b]\!] = b\,;\,(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id), p) \tag{3.40}$$

Les Transformations 3.7 et 3.8 permettent de vérifier la Propriété 3.1 entre les états mémoire atteints avant l'exécution de chacune des tâches. C'est pourquoi la même dénomination $\mathcal{T}_{\mathbb{S}_{com}}$ est utilisée pour décrire ces transformations. Néanmoins, la Transformation 3.7 conduit à de moins bonne performance. Cet aspect est discuté en section 3.4.

D'autres optimisations quant à la génération des variables devant être copiées sont envisageables. Mais elles complexifient la garantie de la cohérence mémoire entre la variable d'origine et ses variables répliquées. Par exemple, en plus de l'état mémoire avant l'exécution d'une tâche, la vivacité peut également être considérée. Si une variable n'est pas utilisée ou est réécrite par la suite, alors la copie de la valeur de cette variable sur ses variables répliquées n'est pas nécessaire. De même, la considération du placement des tâches est source d'optimisation. Elle entraînerait une modification de la Définition 3.6 donnant les variables répliquées à mettre à jour. Ces optimisations de génération des copies sont rediscutées plus tard en section 5.2.

Pour pouvoir faire nos preuves, nous avons opté pour une cohérence mémoire sur l'ensemble des variables, d'où l'utilisation de la Transformation 3.8. La phase d'optimisation, en section 3.4, permettra de supprimer *a posteriori* les copies, représentant des communications, qui sont inutiles.

3.3.3 Matérialisation du placement d'une tâche sur un processus

Cette transformation permet d'affecter concrètement chacune des tâches au processus qui lui est assigné. Ceci est fait au moyen d'une substitution des identifiants d'origine par l'identifiant répliqué correspondant au processus sur lequel la tâche doit être exécutée. La Transformation 3.9 réalise cette substitution. Cette transformation s'apparente à de l'*alpha-renaming* [40].

Transformation 3.9 (Substitution des variables d'origine par leur variable répliquée)**.** Cette transformation a besoin d'un argument supplémentaire par rapport aux précédentes transformations : le processus p sur lequel on souhaite exécuter la tâche et par extension les instructions qui la composent. Cet argument p est transmis récursivement à toutes les transformations structurelles (Transformation 3.1), à l'exception de (s, p) qui définit cet argument.

$$
\begin{aligned}
&s \in \langle \mathit{statements} \rangle,\ p \in \mathbb{N},\\
&t \in \langle \mathit{type} \rangle,\ \mathit{id} \in \langle \mathit{identifier} \rangle,\\
&\qquad \mathcal{T}_{\mathbb{S}_{map}}[\![t\ \mathit{id}]\!]\langle \_ \rangle = t\ \mathit{id} && (3.41)\\
&\qquad \mathcal{T}_{\mathbb{S}_{map}}[\![(s,p)]\!]\langle \_ \rangle = (\mathcal{T}_{\mathbb{S}_{map}}[\![s]\!]\langle p \rangle, p) && (3.42)\\
&\qquad \mathcal{T}_{\mathbb{E}_{map}}[\![id]\!]\langle p \rangle = \text{ if } \mathit{id} \in \mathit{Id}_0 \text{ then } \mathit{id}_p \text{ else } \mathit{id} && (3.43)
\end{aligned}
$$

Cette transformation modifie la sémantique du code car les noms des variables utilisées dans le programme sont modifiées. De plus, étant donnée que la transformation appliquée est une substitution, on a tout de même la Propriété 3.3 permettant d'affirmer qu'au moins un groupe de variables répliquées a la même valeur que le groupe de variables d'origine pour chacune des tâches. La Propriété 3.1 de cohérence mémoire est modifiée en la Propriété 3.4. Cette dernière diffère par l'absence de référence aux valeurs des variables d'origine. Elle est vérifiée après chaque couple "tâche de calculs" suivie d'une "tâche de copies" résultant de la Transformation 3.8 précédente.

Propriété 3.3. *La valeur d'une variable après l'exécution d'une tâche est identique à la valeur d'une variable répliquée après la Transformation 3.9. Cette variable répliquée est celle qui a servi pour la substitution lors de cette transformation.*

$$
\begin{aligned}
&s \in \langle \mathit{statements} \rangle,\ p \in \mathbb{N},\ \sigma \in \mathit{Store},\\
&\quad \forall \mathit{id} \in \mathit{Id}_0, \qquad \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,p)]\!]\langle \_ \rangle]\!]\langle \sigma \rangle(\mathit{id}_p, \_) = \mathbb{S}[\![(s,p)]\!]\langle \sigma \rangle(\mathit{id}, \_)
\end{aligned}
$$

Propriété 3.4 (Cohérence mémoire entre les variables répliquées)**.** *Après chacune des tâches copies, toutes les variables répliquées, d'une variable d'origine, ont la même valeur que l'une d'entre elle.*

$$\forall \mathit{id} \in \mathit{Id}_0\ \exists p \in [0..P[, \forall q \in [0..P[, \quad \sigma(\mathit{id}_q, \_) = \sigma(\mathit{id}_p, \_)$$

Et par extension, on peut dire que toutes les variables répliquées, d'une variable d'origine, ont la même valeur.

$$\forall \mathit{id} \in \mathit{Id}_0\ \forall p, q \in [0..P[, \quad \sigma(\mathit{id}_q, \_) = \sigma(\mathit{id}_p, \_)$$

Après cette transformation, les variables d'origine ne sont plus utilisées pour réaliser les calculs. Elles ont toutes été remplacées par une de leur équivalente répliquée, dépendant du processus sur lequel la tâche doit s'exécuter. Néanmoins, leurs déclarations sont encore présentes. Des transformations permettant de nettoyer à minima le code sont donc nécessaires.

3.3.4 Nettoyage du code

La dernière étape de préparation du code en vue de la génération de code parallèle distribuée est un nettoyage du code. Il s'effectue en deux transformations. La première, décrite en section 3.3.4.1, permet de supprimer les déclarations des variables d'origine. La seconde, décrite en section 3.3.4.2, consiste à supprimer les copies identités, par exemple `__x_0=__x_0;`, qui ont été introduites lors de nos transformations.

3.3.4.1 Nettoyage des déclarations

Ce nettoyage permet de conserver uniquement les variables qui sont affectées à des processus spécifiques. La Transformation 3.10 examine chaque déclaration pour déterminer si la variable est une variable d'origine, c'est-à-dire appartenant à Id_0, ou non. Si c'est le cas la déclaration est supprimée. Dans le cas contraire, aucune modification n'est faite. Cette transformation est valide car la Transformation 3.9 a supprimé toutes les utilisations des variables d'origine.

Transformation 3.10 (Nettoyage des déclarations d'origine)**.** Cette transformation supprime les déclarations des variables d'origine.

$$\begin{aligned} &t \in \langle type \rangle,\ id \in \langle identifier \rangle, \\ &\qquad \mathcal{T}_{\mathbb{S}_{idc}}[\![t\ id]\!] = \text{if } id \in Id_0 \text{ then } \varnothing \text{ else } t\ id \end{aligned} \tag{3.44}$$

La traditionnelle transformation consistant à nettoyer toutes les déclarations inutiles, *clean declaration*, peut également être utilisée. Elle permet d'obtenir un résultat équivalent à la Transformation 3.10 car la Transformation 3.9, substituant les variables d'origine par une variable répliquée, a supprimé toutes les utilisations des variables d'origine, rendant leur déclaration inutile.

3.3.4.2 Élimination des identités

Lors de la Transformation 3.8, des instructions de copie ont été ajoutées pour toutes les variables répliquées, y compris pour les variables du processus sur lequel les instructions vont s'exécuter. De ce fait, lors de la Transformation 3.9 des instructions de copies affectant une variable à elle-même sont créées. La Transformation 3.11 a pour but de supprimer de telles instructions de copies.

Transformation 3.11 (Élimination des identités)**.** Cette transformation supprime les instructions d'affectation modifiant une variable par elle-même.

$$\begin{aligned} &var \in \langle variable \rangle, \\ &\qquad \mathcal{T}_{\mathbb{S}_{eid}}[\![var \leftarrow var]\!] = \varnothing \end{aligned} \tag{3.45}$$

3.3.5 Résultat de la préparation du code

Cette phase de préparation est composée de cinq transformations. Ces transformations ont permis de modifier le code de telle façon qu'une modélisation du code parallèle distribué soit intégrée au code séquentiel. Pour cela, on a ajouté des variables pour chaque processus avec la Transformation 3.5. Puis la cohérence mémoire entre chacune de ces variables sur les divers processus est assurée avec la Transformation 3.8. La spécialisation de l'exécution de chacune des tâches par un processus a ensuite été réalisée au moyen de la Transformation 3.9. Enfin, la suppression des variables d'origine et de certaines instructions de copies inutiles s'effectue avec les transformations 3.10 et 3.11.

À partir de ce code modélisant une exécution sur différents processus, la génération du code parallèle distribué pourrait directement se faire (section 3.5). Néanmoins, la Transformation 3.8 a introduit des instructions de copies, représentant les futures instructions de communication pouvant être inutiles de part son aspect conservateur. Une phase d'optimisation est donc nécessaire si on souhaite réduire le nombre de communications qui vont être générés et qui sont très coûteuses pour l'exécution d'un programme. De plus, si les copies persistent entre chacune des tâches pour toutes les variables répliquées, le code parallèle distribué sera en réalité un code séquentiel distribué, c'est-à-dire un code séquentiel avec un surcoût dû au communication. La transformation présentée dans la section suivante est donc nécessaire pour générer un véritable code parallèle qui soit performant.

3.4 Les optimisations de code

Concernant les optimisations réalisées, on s'attache particulièrement aux transformations permettant de réduire le nombre de copies représentant des communications. Ces optimisations sont de plusieurs natures. Elles peuvent majoritairement être réalisées par une seule transformation d'optimisation, à savoir l'élimination de code mort.

Cette phase d'optimisation correspond aux groupes "Optimisations locales" et "Optimisations globales" de la Figure 2.3.

3.4.1 Élimination de code mort

L'élimination de code mort, ou *dead code elimination*, est une passe traditionnelle d'optimisation de code. Elle détecte les écritures d'une variable qui ne sera jamais lue par la suite et les supprime. Elle permet également d'éliminer les écritures d'une variables qui sera mise à jour, sans qu'aucune lecture de cette variable ne soit faite entre temps. Elle utilise pour cela les *USE-DEF chains* ou le graphe de dépendances de données. Cette transformation est déjà présente dans le compilateur source-à-source PIPS utilisé dans cette thèse.

Cette optimisation permet d'éliminer des instructions de copies que l'on peut grouper dans trois catégories différentes :

— les copies de variables qui ne sont pas modifiées dans la tâche de calculs (section 3.4.1.1), si la Transformation 3.7 est utilisée au lieu de la Transformation 3.8 ;

— les copies de variables qui ne seront jamais réutilisées par la suite, (section 3.4.1.2), sont qualifiées de copies inutiles dans la suite, bien que dans les deux autres cas leurs copies soient également inutiles ;
— les copies de variables qui sont modifiées à nouveau dans une tâche ultérieure et ne sont pas utilisées entre temps (section 3.4.1.3), sont qualifiées de copies redondantes dans la suite.

Les points 2 et 3 sont illustrés dans le Tableau 3.1.

	case 2	case 3
task on P	write __x_P ~~__x_Q = __x_P~~ __x_R = __x_P	write __x_P __x_Q = __x_P ~~__x_R = __x_P~~
task on Q Q ≠P	...	read __x_Q write __x_Q __x_P = __x_Q __x_R = __x_Q
task on R R ≠{P,Q}	read __x_R	read __x_R

TABLE 3.1 – Suppression des copies inutiles et redondantes

3.4.1.1 Élimination des copies non modifiées

Lorsque la Transformation 3.7 (p. 54) est utilisée pour générer les copies, les informations sur les variables modifiées par la tâche ne sont pas particulièrement considérées. Par conséquent, une suppression des copies des variables non modifiées est nécessaire pour réduire les communications qui seront générées et pour permettre l'apparition d'un parallélisme potentiel entre tâches.

	En théorie	Souhaité	En pratique
task on P	write __x_P ~~S_{cp}(x)~~	write __x_P ...	write __x_P __x_Q = __x_P ~~__x_R = __x_P~~
task on Q Q ≠P	... S_{cp}(x)	... __x_R = __x_P	... ~~__x_P = __x_Q~~ __x_R = __x_Q
task on R R ≠{P,Q}	read __x_R ...	read __x_R ...	read __x_R __x_P = __x_R __x_Q = __x_R

TABLE 3.2 – Élimination des copies non modifiées

En théorie, une élimination du code mort peut faire le travail en ne gardant qu'une fonction de copies S_{cp} (Définition 3.6, p. 54) pour une variable donnée. Par contre, après l'élimination du code mort, cette copie ne sera pas placée après l'exécution de la tâche modifiant la variable, mais plutôt avant la tâche ayant besoin de cette variable. La Transformation 3.7 suivie d'une élimination de code mort ne sera donc pas identique à la Transformation 3.8 (p. 55) suivie d'une élimination de code mort. Mais elles devraient rester équivalentes en

terme du nombre de copies effectuées. En pratique, l'élimination de code mort ne permet pas d'éliminer le caractère séquentiel du code résultant de la Transformation 3.7. La Transformation 3.9 (p. 56) attribue les différentes copies aux processus correspondants. De ce fait, les dépendances de données sont toujours présentes avec les copies. Ainsi, pour transmettre une variable `x` modifiée par le processus `P` au processus `R` qui l'utilise, si une tâche intermédiaire sur le processus `Q` existe, alors même si `Q` n'utilise pas la variable `x`, elle devra tout de même transiter par `Q`. Le Tableau 3.2 illustre le résultat souhaité et celui obtenu avec l'élimination du code mort.

L'élimination de code mort traditionnelle ne donne pas le résultat souhaité dans ce cas, bien qu'elle élimine effectivement des copies. Une modification de celle-ci est nécessaire si l'on souhaite obtenir le résultat voulu. Néanmoins, si la Transformation 3.8 est utilisée à la place de la Transformation 3.7, ce problème ne se pose plus. Il est traité en amont. Étant donné qu'on utilise la Transformation 3.8 dans notre suite de transformations pour générer notre code parallèle, on peut mettre de côté cette problématique.

3.4.1.2 Élimination des copies inutiles

J'entends par copie inutile l'affectation d'une variable qui n'est jamais réutilisée par la suite. Je la différencie ainsi d'une copie redondante qui est décrite dans la section 3.4.1.3.

L'élimination des copies inutiles correspond au cas 2 du Tableau 3.1. Un processus `Q` dont aucune tâche utilise une variable `x` n'a pas besoin de recevoir la valeur de cette variable. Toutes les copies affectant `__x_Q` peuvent être supprimées.

Cette optimisation concerne principalement les copies qui sont effectuées vers la fin d'un programme, où il y a de moins en moins de tâches devant faire des calculs. Lorsque cette optimisation se produit sur des tâches en début de programme, elle permet d'introduire le réel parallélisme. En effet, en reprenant l'exemple du cas 2 du Tableau 3.1, si aucune des valeurs des variables présentes sur la tâche s'exécutant sur le processus `P` n'est copiée pour la tâche associée au processus `Q`, alors les dépendances dues aux copies entre ces deux tâches sont supprimées. De ce fait, lors du passage au code parallèle, ces deux tâches vont pouvoir être exécutées simultanément.

3.4.1.3 Élimination des copies redondantes

Une copie redondante est une affectation d'une variable qui est réaffectée par la suite et qui n'est pas lue entre temps. La valeur qui est affectée à cette variable n'est pas forcément, voire rarement, la même.

L'élimination des copies redondantes correspond au cas 3 du Tableau 3.1. Il ne considère que la variable `x` qui est utilisée et/ou définie par trois tâches sur trois processus différents. La première tâche sur le processus `P` définit `x` à une certaine valeur, puis la transmet aux processus `Q` et `R`. La deuxième tâche sur `Q` utilise cette valeur de `x` puis modifie `x`. De fait, elle transmet cette nouvelle valeur à `P` et `R`. Enfin la dernière tâche sur `R` a besoin de `x` pour faire ces calculs. La copie de `x` du processus `P` à `Q` est nécessaire car la tâche sur `Q` l'utilise pour faire des calculs. Par contre la copie de `x` du processus `P` à `R` est redondante avec celle du processus `Q` à `R` qui affecte une valeur plus à jour. Seule cette dernière

valeur est utilisée par le processus `R`. Ainsi, la copie de `x` du processus `P` à `R` peut être supprimée.

Contrairement à l'élimination des copies inutiles, cette élimination des copies redondantes ne permet pas de supprimer des dépendances entre tâches. Elle ne permet donc pas de mettre en exergue plus de parallélisme. Néanmoins, elle permet de réduire grandement le nombre de copies présentes et donc de réduire le nombre de communications qui vont être générées par la suite. Or dans un programme distribué, une instruction de communication est considérée comme étant très lente comparée à une instruction faisant un calcul. Ainsi, supprimer un maximum de copies, et donc de communications potentielles, permet d'améliorer les performances du programme résultant.

En réutilisant une passe d'optimisation bien connue et déjà présente dans la plupart des compilateurs, à savoir l'élimination de code mort, on peut optimiser fortement notre code en vue de la génération du code parallèle. L'élimination du code mort permet à la fois (1) d'introduire le parallélisme potentiel des futures tâches qui vont être exécutées simultanément (section 3.4.1.2), mais également (2) de réduire fortement le nombre de communications qui devront être faites en ne communiquant pas de valeurs de variables qui ne seront jamais utilisées (section 3.4.1.3). On peut désormais effectuer la phase de génération du code parallèle (section 3.5).

3.4.2 Les autres optimisations possibles

D'autres optimisations peuvent encore être effectuées, mais elles ne sont pas détaillées dans ce chapitre car elles ne sont pas considérées comme indispensables pour l'obtention d'un code parallèle. Le chapitre suivant n'en fera d'ailleurs aucune preuve. Une explication succincte est tout de même donnée et leur détail présenté dans le chapitre 5.

Une première optimisation est similaire à l'élimination de code mort traditionnelle, mais au lieu de supprimer les instructions d'affectations inutiles, elle supprimera les itérations de boucles inutiles. Elle se concentre sur les tableaux et leur utilisation dans des boucles. Cette optimisation sera appelée élimination d'itérations inutiles, *dead iteration elimination* et est présenté en section 5.3.

Une autre optimisation est de considérer l'espace mémoire qui est utilisé par les différents processus lorsque le code parallèle est généré, et de le réduire. Pour cela, un nettoyage des déclarations inutiles peut être fait pour les variables scalaires, ou lorsque qu'un tableau entier n'est pas utilisé par un processus. Mais on peut aller encore plus loin en redimensionnant la taille des tableaux déclarés sur chacun des processus à leur minimum. Ce redimensionnement de la taille des tableaux est décrit en section 5.4.

3.5 La génération automatique du code parallèle

La phase de génération du code parallèle correspond au groupe "Génération de code" de la Figure 2.3.

Le code séquentiel est maintenant prêt à être traduit en un code parallèle. La préparation de celui-ci est terminée et des optimisations ont été réalisées pour permettre l'obtention d'un réel parallélisme. Cette dernière phase de compilation

peut ainsi s'appliquer. Le modèle d'exécution parallèle distribué est décrit, ainsi qu'une nouvelle syntaxe et sémantique parallèle. Le passage au code parallèle est expliqué en deux étapes.

3.5.1 Le modèle *Bulk Synchronous Parallel*

Le modèle BSP, *Bulk Synchronous Parallel*, est un modèle de programmation parallèle pour architecture distribuée, développé par Valiant [107] en 1990 et largement étudié par McColl [46][66].

Le modèle BSP consiste à définir l'exécution d'un programme parallèle distribué au moyen de super-étapes, *supersteps*. Ces dernières se décomposent en trois sous-étapes :

1. Les **calculs locaux** permettent à chaque processus d'effectuer ses calculs sans tenir compte des calculs effectués par les autres processus.
2. Les **communications** échangent les données entre les différents processus. Un même processus ne pouvant pas recevoir plusieurs valeurs pour la même variable.
3. La **synchronisation globale** permet d'attendre que l'exécution des calculs locaux et des communications sur tous les processus soient finis.

Dans ce modèle, toutes les données communiquées durant une super-étape ne sont pas disponibles avant la super-étape suivante. Ce principe de super-étapes a inspiré le système Pregel de Google [65].

BSP représente également un modèle de coût d'exécution. McColl [66] référence plusieurs articles décrivant la complexité et le coût d'algorithmes classiques parallèles.

Bien que la présence de synchronisations globales et donc de barrières est assez coûteuse et dépréciée, ce modèle permet de décrire simplement l'exécution d'un programme parallèle distribué. Je m'inspire en partie de ce modèle dans la description de la sémantique parallèle utilisée.

3.5.2 Sémantique parallèle

La sémantique parallèle est une extension de la sémantique séquentielle présentée au début de ce chapitre, en section 3.1. Elle est inspirée du modèle BSP [107][101].

Avant d'introduire la sémantique parallèle, on étend la syntaxe séquentielle avec des communications $\langle com \rangle$ et on transforme les $\langle block \rangle$ en des $\langle block_p \rangle$ parallèles comme suit :

$$
\begin{array}{lcll}
\langle com \rangle & ::= & \mathrm{com}(s\,;\, r\,;\, \langle variable \rangle;\, \langle variable \rangle) & 0 \leq s < P,\, 0 \leq r < P \\
\langle block_p \rangle & ::= & \varnothing \mid \langle block_p \rangle\,;\, \langle block_p \rangle & \\
 & \mid & \langle fun\text{-}declaration \rangle & \\
 & \mid & (\langle statements \rangle, \ldots, \langle statements \rangle) & \\
 & \mid & \langle com \rangle &
\end{array}
$$

Avec cette syntaxe, $(\langle statements \rangle, \ldots, \langle statements \rangle)$ représente la sous-étape de calculs locaux de BSP, et une succession de $\langle com \rangle$ la sous-étape de communications. $\langle com \rangle$ permet de faire des communications point-à-point entre deux processus, un émetteur s et un récepteur r. L'émetteur envoie le troisième argument et le récepteur reçoit sur le quatrième argument comme la sémantique (Équation 3.50) le décrit par la suite.

On définit également une notation pour une liste d'éléments dans la Définition 3.12 permettant d'avoir plusieurs états mémoire ou plusieurs instructions à exécuter simultanément et de façon ordonnée.

Définition 3.12 (Notation liste d'éléments)**.** On indexe une liste d'éléments avec des exposants. Ainsi, on a par exemple $(s^0, \ldots, s^{P-1})$ avec $s^0, \ldots, s^{P-1} \in \langle statements \rangle$.

De plus, on note ${}^{max}_{min}X$ la liste $X^{min}, \ldots, X^{max-1}$. Par exemple, pour une liste d'états mémoire,

$$ {}^{P}_{0}\sigma = (\sigma^0, \ldots, \sigma^{P-1}) $$

pour une liste de séquences d'instructions,

$$ {}^{P}_{0}s = (s^0, \ldots, s^{P-1}) $$

La sémantique parallèle, notée $\mathbb{P}$, a la forme :

$$ \mathbb{P} : \langle block_p \rangle \to Store^P \to Store^P $$

Elle est définie comme suit :

$b_1, b_2 \in \langle block_p \rangle$, $decl \in \langle fun\text{-}declaration \rangle$, $s^0, \ldots, s^{P-1} \in \langle statements \rangle$, $var_r, var_s \in \langle variable \rangle$, $r,s \in \mathbb{N}$

$$ \mathbb{P}[\![\varnothing]\!]\langle {}^{P}_{0}\sigma \rangle = {}^{P}_{0}\sigma \tag{3.46} $$

$$ \mathbb{P}[\![b_1 \,;\, b_2]\!]\langle {}^{P}_{0}\sigma \rangle = \mathbb{P}[\![b_2]\!]\langle \mathbb{P}[\![b_1]\!]\langle {}^{P}_{0}\sigma \rangle \rangle \tag{3.47} $$

$$ \mathbb{P}[\![decl]\!]\langle {}^{P}_{0}\sigma \rangle = (\mathbb{S}[\![decl]\!]\langle \sigma^0 \rangle, \ldots, \mathbb{S}[\![decl]\!]\langle \sigma^{P-1} \rangle) \tag{3.48} $$

$$ \mathbb{P}[\![{}^{P}_{0}s]\!]\langle {}^{P}_{0}\sigma \rangle = (\mathbb{S}[\![s^0]\!]\langle \sigma^0 \rangle, \ldots, \mathbb{S}[\![s^{P-1}]\!]\langle \sigma^{P-1} \rangle) \tag{3.49} $$

$$ \begin{aligned} \mathbb{P}[\![\mathrm{com}(s\,;r\,;var_s\,;var_r)]\!]\langle {}^{P}_{0}\sigma \rangle = (&\sigma^0, \\ &\vdots \\ &\sigma^{r-1}, \\ &update(\sigma^r, \mathbb{L}[\![var_r]\!]\langle \sigma^r \rangle, \mathbb{E}[\![var_s]\!]\langle \sigma^s \rangle), \\ &\sigma^{r+1}, \\ &\vdots \\ &\sigma^{P-1}) \end{aligned} \tag{3.50} $$

D'après notre sémantique parallèle, on suppose qu'une sous-étape de synchronisation globale est présente après chacune des instructions de la forme $(\langle statements \rangle, \ldots, \langle statements \rangle)$ ou $\langle com \rangle$.

3.5.3 Passage à un code syntaxiquement parallèle

Le passage du code séquentiel au code parallèle correspond au passage de la syntaxe séquentielle définie en section 3.1.1 à la syntaxe parallèle présentée précédemment. La différence majeure entre ces deux syntaxes est le remplacement des $\langle block \rangle$ par des $\langle block_p \rangle$ et de ce fait l'apparition de communication $\langle com \rangle$.

Une fonction *copy2com* traduisant les affectations de copie, générées automatiquement lors des transformations, en une instruction de communication est présentée dans la Définition 3.13.

Définition 3.13 (Affectation séquentielle vers communication parallèle)**.** On définit la fonction *copy2com* qui permet à partir d'une instruction d'affectation de copie appartenant au domaine séquentiel de générer une instruction de communication correspondante appartenant au domaine parallèle.

$$copy2com : \langle affectation \rangle \rightarrow \langle com \rangle$$

$$\forall id \in Id_0,\ \forall elem \in \langle identifier \rangle,\ \forall k, p \in \mathbb{N}$$

$$copy2com(id_k \leftarrow id_p) = \text{com}(\, p\, ;\, k\, ;\, id_p\, ;\, id_k\,) \tag{3.51}$$

$$copy2com(id_k.elem \leftarrow id_p.elem) = \text{com}(p\, ; k\, ;\, id_p.elem\, ; id_k.elem) \tag{3.52}$$

$$copy2com\begin{pmatrix}\text{for}(i_m \leftarrow 0;\\ \quad i_m < sizeof(id_p);\\ \quad i_m \leftarrow i_m + 1)\\ id_k[i_m] \leftarrow id_p[i_m]\end{pmatrix} = \begin{array}{l}\text{com}(\, p\, ;\, k\, ;\, id_p[0]\, ;\, id_k[0]\,)\, ;\\ \dots\ ;\\ \text{com}(\, p\, ;\, k\, ;\, id_p[sizeof(id_p)]\, ;\, id_k[sizeof(id_p)]\,)\, ;\end{array} \tag{3.53}$$

La Transformation 3.14 permet de réaliser le passage du code séquentiel au code parallèle. Elle différencie les $\langle block \rangle$ de la forme $(\langle statements \rangle, p)$ effectuant des calculs de ceux effectuant des copies S_{cp} représentant des communications. Pour ceux effectuant des calculs, elle affecte ces calculs aux processus correspondant. Pour ceux effectuant des copies, elle utilise la fonction *copy2com* introduite précédemment pour les traduire en communications.

Transformation 3.14 (Du séquentiel au parallèle)**.** Cette transformation permet de transposer les $\langle block \rangle$ en des $\langle block_p \rangle$.

$$S_t \in \langle statements \rangle (S_t \neq S_{cp}),\ decl \in \langle fun\text{-}declaration \rangle,\ b_1, b_2 \in \langle block \rangle$$

$$\mathcal{T}_{\mathbb{S}_{s2p}}[\![\varnothing]\!] = \varnothing \tag{3.54}$$

$$\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1\, ; b_2]\!] = \mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1]\!]\, ; \mathcal{T}_{\mathbb{S}_{s2p}}[\![b_2]\!] \tag{3.55}$$

$$\mathcal{T}_{\mathbb{S}_{s2p}}[\![decl]\!] = decl \tag{3.56}$$

$$\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_t, p)]\!] = (\underbrace{\varnothing, \dots, \varnothing}_{p}, S_t, \underbrace{\varnothing, \dots, \varnothing}_{P-p-1}) \tag{3.57}$$

$$\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_{cp}, p)]\!] = \bigcup_{s \in S_{cp}} copy2com(s) \tag{3.58}$$

3.5.4 Mise en parallèle de l'exécution

La transformation précédente a permis de traduire notre code séquentiel sous forme parallèle. Par contre, si on étudie le code généré, son exécution est toujours séquentielle étant donné que dans les sous-étapes de calculs locaux $(\langle statements \rangle, \dots, \langle statements \rangle)$ un seul de ces $\langle statements \rangle$ est non vide. On remarque tout de même que dans la série de communications effectuées après ces calculs, un seul processus est émetteur, celui effectuant les calculs de la tâche précédente, et que tous les processus ne sont pas forcément récepteurs, grâce aux optimisations effectuées à la phase précédente (section 3.4).

La Propriété 3.5 permet d'indiquer quand la sous-étape de calculs locaux d'une super-étape peut en réalité s'exécuter pendant la super-étape précédente. Elle permet ainsi d'avoir une exécution parallèle.

Propriété 3.5 (Exécution locale non contrainte)**.** *Si pour deux sous-étapes de calculs (a et b) qui se suivent, un processus (j) n'est impliqué dans aucune communication entre ces deux sous-étapes, alors les instructions de ce processus de la deuxième sous-étape (b) peuvent s'exécuter durant la première sous-étape de calcul (a).*

$$\forall(..., S_a^j, ...), (..., S_b^j, ...) \in (\langle statements \rangle, \ldots, \langle statements \rangle), \forall_0^P \sigma,$$
$$\text{if } \forall \text{com}(\, k\,;\, i\,;\, id_k\,;\, id_i\,) \text{ such that } i \neq j \wedge k \neq j,$$
$$\mathbb{P}\left[\!\!\left[\begin{array}{l} (..., S_a^j, ...)\,; \\ ...\,; \\ \text{com}(\, k\,;\, i\,;\, id_k\,;\, id_i\,)\,; \\ ...\,; \\ (..., S_b^j, ...) \end{array} \right]\!\!\right] \left\langle {}_0^P \sigma \right\rangle = \mathbb{P}\left[\!\!\left[\begin{array}{l} (..., S_a^j\,; S_b^j, ...)\,; \\ ...\,; \\ \text{com}(\, k\,;\, i\,;\, id_k\,;\, id_i\,)\,; \\ ...\,; \\ (..., \varnothing, ...) \end{array} \right]\!\!\right] \left\langle {}_0^P \sigma \right\rangle$$

L'application d'une transformation respectant la Propriété 3.5 permet une réelle exécution parallèle du code généré. Contrairement aux autres transformations qui s'appliquent de façon descendante, celle-ci est ascendante.

3.6 Conclusion

Ce chapitre a décrit les différentes phases successives réalisées pour effectuer une parallélisation semi-automatique ou automatique des tâches sur une architecture à mémoire distribuée. Le placement peut être fait de façon automatique, ce qui rend alors cette génération complètement automatique (section 3.2).

Les descriptions formelles de la syntaxe et de la sémantique utilisées ont été présentées en sections 3.1 pour la partie séquentielle et 3.5.2 pour la partie parallèle. Elles ont été utilisées pour décrire formellement les différentes phases et les transformations qui composent notre processus de génération du code parallèle.

La description du code parallèle utilisée repose en partie sur le modèle BSP (section 3.5.1). Le code séquentiel a donc été préparé et décrit en section 3.3 pour permettre une traduction dans ce modèle en section 3.5. Des optimisations séquentielles ont été effectuées en réutilisant une transformation présente dans la plupart des compilateurs (section 3.4.1).

Ce chapitre montre qu'il est possible de générer un code parallèle distribué pour des tâches au moyen de transformations très simples. Bien que ces transformations doivent être appliquées dans un certain ordre, notamment pour la préparation du code, elles ont l'avantage d'être modulaires. Par exemple, plusieurs possibilités de mise en cohérence de la mémoire sont envisagées avec une plus ou moins grande précision (section 3.3.2). Avec une phase d'optimisation simple (section 3.4), il est déjà possible d'avoir un code pouvant s'exécuter de façon parallèle. D'autres optimisations de ce processus de compilation sont encore possibles comme nous le verrons au chapitre 5.

Le chapitre 4 suivant complète la description formelle des transformations effectuées durant ce chapitre en y ajoutant les preuves de correction du code parallèle généré.

Chapitre 4

Preuves de correction des transformations

Sommaire

La description des transformations successives, nécessaire à notre génération de code distribué, a été donnée dans le chapitre précédent. Ce chapitre à pour objectif de montrer que le code généré est correct, en donnant les preuves de chacune de ces transformations.

Ces preuves consistent à montrer que chaque transformation génère un code équivalent au précédent, c'est-à-dire que les états mémoire avant et après la transformation du code peuvent être mis en relation. Par rapport aux différentes transformations effectuées, plusieurs relations d'équivalence sont définies, plus ou moins conservatives. Ces relations d'équivalence peuvent ne porter que sur une partie des variables, ou bien renommer les variables qui doivent être comparées. Pour vérifier que les programmes avant et après une transformation sont équivalents, une preuve par induction sur l'évaluation des différentes étapes du programme est réalisée. Plus formellement, pour deux états mémoire σ, σ' équivalents par une relation $\approx$, l'évaluation d'un programme $[\![P]\!]$ dans $\langle\sigma\rangle$ est équivalent à l'évaluation de sa transformée $[\![\mathcal{T}[\![P]\!]]\!]$ dans $\langle\sigma'\rangle$. Dans notre cas, le programme est représenté par le corps s de la fonction à transformer.

$$\forall\sigma,\sigma',\forall s \in \langle block\rangle, \forall\mathcal{T},$$
$$\sigma \approx \sigma' \Rightarrow \mathbb{S}[\![s]\!]\langle\sigma\rangle \approx \mathbb{S}[\![\mathcal{T}[\![s]\!]]\!]\langle\sigma'\rangle$$

Les sections 4.1 à 4.3 présentent les preuves faites pour les différentes phases de transformation présentées au chapitre précédent. Le formalisme de ces transformations est rappelé au début de chaque section lorsque c'est nécessaire. La sémantique utilisée est celle présentée au chapitre 3 pages 49 et 50 pour la sémantique séquentielle et page 62 pour la sémantique parallèle. La section 4.4 conclut ce chapitre et présente un tableau récapitulatif des transformations et des relations d'équivalence qui sont utilisées dans ce chapitre.

4.1 Les preuves sur la préparation du code

Le placement des tâches sur les différents processus consiste en l'ajout d'annotations `pragma` sur le code. Donc sémantiquement le code est identique avec ou sans placement. Aucune preuve n'est donc nécessaire.

La première phase de transformations pour laquelle on réalise les preuves concerne la préparation du code, décrite en section 3.3. Cette phase est composée de plusieurs étapes de transformation qui sont prouvées correctes dans les sections suivantes : réplication de l'environnement, maintien de la cohérence mémoire, matérialisation du placement, nettoyage des déclarations et élimination des identités.

4.1.1 Réplication de l'environnement

La transformation de réplication de l'environnement a été décrite en section 3.3.1. La définition de la Transformation 3.5 est rappelée :

Transformation 3.5 (Réplication des déclarations)**.**

$$t \in \langle type \rangle,\ fid \in \textit{fun-identifier}, \quad \mathcal{T}_{\mathbb{S}_{add}}[\![t\ fid]\!] = t\ fid ; t\ fid_0 ; \ldots ; t\ fid_{P-1}$$

Après cette transformation, il faut prouver que des variables répliquées, appartenant à Id_{new} (Définition 3.4 p. 53), sont bien déclarées pour toutes les variables d'origine, appartenant à Id_0 (Définition 3.2 p. 52). La sémantique utilisée (section 3.1.2) fait la distinction entre une variable qui n'existe pas et une variable déclarée mais non initialisée. L'évaluation d'une variable qui n'existe pas retourne une erreur ε, alors que l'évaluation d'une variable déclarée mais non initialisée retourne $\perp$.

La Relation 4.1 $\approx_0$ définit une relation d'équivalence classique entre deux états mémoire pour l'ensemble des variables d'origine.

Relation 4.1 (Équivalence sur Id_0)**.** On définit la relation d'équivalence $\approx_0$ sur l'ensemble des variables d'origine Id_0 entre deux états mémoire par :

$$\sigma \approx_0 \sigma' \stackrel{\text{def}}{=} \forall x \in Id_0, \forall i \in \mathbb{N}, \sigma(x,i) = \sigma'(x,i) \tag{4.1}$$

Proposition 4.1 (Égalité des variables d'origines)**.** *Le programme résultant de la Transformation 3.5 est équivalent, d'après la Relation 4.1, au programme d'origine. Cette équivalence est vérifiable au niveau de toutes les instructions,* $\langle block \rangle$*,* $\langle statements \rangle$ *et* $\langle statement \rangle$*, d'une fonction.*

$$\forall s \in \langle block \rangle \cup \langle statements \rangle \cup \langle statement \rangle, \forall \sigma_0, \sigma_1,$$
$$\sigma_0 \approx_0 \sigma_1 \Rightarrow \mathbb{S}[\![s]\!]\langle \sigma_0 \rangle \approx_0 \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{add}}[\![s]\!]]\!]\langle \sigma_1 \rangle$$

Démonstration. Triviale étant donné que la Transformation 3.5 ne modifie pas l'état mémoire du programme pour les variables d'origine, appartenant à Id_0. □

La Relation 4.2 $\mathcal{R}_d$ définit une relation d'extension du domaine d'un état mémoire par rapport à un autre. En d'autres termes, la Relation 4.2 permet d'affirmer que de nouvelles variables ont été déclarées.

Relation 4.2 (Extension d'un domaine)**.** On définit la relation d'extension du domaine d'un état mémoire $\mathcal{R}_d$ entre deux états mémoire par :

$$\begin{aligned} F_\sigma &\overset{\text{def}}{=} \{fid_k | fid \in (dom(\sigma) \cap \textit{fun-identifier}) \wedge 0 \leq k < P\} \\ \sigma \,\mathcal{R}_d\, \sigma' &\overset{\text{def}}{=} dom(\sigma') = dom(\sigma) \cup F_\sigma \\ &\quad \wedge \forall x \in dom(\sigma), \forall i \in \mathbb{N},\ \sigma(x,i) = \sigma'(x,i) \\ &\quad \wedge \forall x \in F_\sigma, \forall 0 \leq i < sizeof(typeof(x)),\ \sigma(x,i) = \bot \end{aligned}$$

Remarque 4.1. Id_{new} de la Définition 3.4 correspond à F_σ dans la Relation 4.2 précédente, avec σ l'état mémoire de la fin du programme.

$$Id_{new} = F_\sigma$$

Proposition 4.2 (Extension du domaine aux variables répliquées)**.** *La Transformation 3.5 permet de déclarer les variables répliquées pour les différents processus.*

$$\begin{aligned} &\forall s \in \langle block \rangle \cup \langle statements \rangle \cup \langle statement \rangle, \forall \sigma_0, \sigma_1, \\ &\quad \sigma_0 \,\mathcal{R}_d\, \sigma_1 \Rightarrow \mathbb{S}[\![s]\!]\langle\sigma_0\rangle \,\mathcal{R}_d\, \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{ddd}}[\![s]\!]]\!]\langle\sigma_1\rangle \end{aligned}$$

Démonstration. Seules les déclarations de nouvelles variables au niveau de la fonction sont introduites par la transformation $\mathcal{T}_{\mathbb{S}_{ddd}}$. Ainsi, seules les déclarations sont étudiées pour prouver le respect de la relation d'équivalence.
Soit σ_0, σ_1 tel que $\sigma_0 \,\mathcal{R}_d\, \sigma_1$.
Soit $t \in \langle type \rangle$ et $fid \in \textit{fun-identifier}$.

$$\left.\begin{aligned} &\mathcal{T}_{\mathbb{S}_{ddd}}[\![t\ fid]\!] = t\ fid; t\ fid_0; ...; t\ fid_{P-1} \\ &\mathbb{S}[\![t\ fid]\!]\langle\sigma_0\rangle \overset{(3.19)}{=} \lambda(x,i).\ \text{if } x = fid \wedge \\ &\qquad\qquad 0 \leq i < sizeof(t) \\ &\qquad\qquad \text{then } \bot \\ &\qquad\qquad \text{else } \sigma_0(x,i) \\ &\mathbb{S}[\![t\ fid; t\ fid_0; ...; t\ fid_{P-1}]\!]\langle\sigma_1\rangle \\ &\quad \overset{(3.19)}{=} \lambda(x,i).\ \text{if } (x = fid \vee \\ &\qquad\qquad x = fid_0 \vee \\ &\qquad\qquad ... \vee \\ &\qquad\qquad x = fid_{P-1}) \\ &\qquad\qquad \wedge 0 \leq i < sizeof(t) \\ &\qquad \text{then } \bot \\ &\qquad \text{else } \sigma_1(x,i) \end{aligned}\right\} \Rightarrow \begin{array}{l}\text{par définition de la Relation 4.2} \\ \mathbb{S}[\![t\ fid]\!]\langle\sigma_0\rangle \,\mathcal{R}_d\, \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{ddd}}[\![t\ fid]\!]]\!]\langle\sigma_1\rangle\end{array}$$

□

4.1.2 Maintien de la cohérence mémoire

Le maintien de la cohérence mémoire a été décrit en section 3.3.2. Deux transformations y ont été décrites pour réaliser cette cohérence mémoire. Ces deux transformations vérifient les mêmes propriétés après leur application.

La première transformation est la Transformation 3.7, rappelée ci-dessous.

Transformation 3.7 (Cohérence complète de la mémoire entre tâches)**.**

$$b \in (\langle statements \rangle, p),$$
$$\mathcal{T}_{\mathbb{S}_{com}}[\![b]\!] = b\,;\,(\bigcup_{id \in Id_0} S_{cp}(id), p)$$

La Relation 4.3 $\approx_{1\to P}$ définit une relation d'équivalence entre deux programmes en utilisant la Relation 4.1 d'équivalence précédente où, de plus, toutes les variables répliquées d'un des programmes ont la même valeur que leur variable associée dans l'ensemble des variables d'origine. La Relation 4.3 permet de garantir la Propriété 3.1 du chapitre précédent.

Relation 4.3 (Équivalence avec cohérence mémoire)**.** On définit la relation d'équivalence $\approx_{1\to P}$ avec cohérence mémoire sur P processus entre 2 états mémoire par :

$$\sigma \approx_{1\to P} \sigma' \stackrel{\text{def}}{=} \sigma \approx_0 \sigma' \wedge \forall x \in Id_0, \forall p \in [0..P[, \forall i \in \mathbb{N}, \sigma(x,i) = \sigma'(x_p, i) \tag{4.2}$$

Le Lemme 4.1 garantit qu'après la séquence d'instructions générées par la fonction S_{cp}, les variables répliquées sur tous les processus, associée à une variable d'origine donnée en paramètre, ont la même valeur que cette variable.

Lemme 4.1 (Mise en cohérence par S_{cp})**.** *La fonction S_{cp} permet la mise en cohérence de l'état mémoire σ pour toute variable d'origine x avec leurs variables répliquées.*

$$\forall \sigma, \forall x \in Id_0, \forall p \in [0..P[, \forall i \in \mathbb{N},$$
$$\sigma(x,i) = \mathbb{S}[\![S_{cp}(x)]\!]\langle\sigma\rangle(x,i) = \mathbb{S}[\![S_{cp}(x)]\!]\langle\sigma\rangle(x_p, i) \tag{4.3}$$

Démonstration. Avec la Définition 3.6 (p. 54) de S_{cp} et l'Équation 3.15 (p. 50) de la sémantique des affectations, on déduit l'Équation 4.3 du lemme précédent. □

Proposition 4.3 (Cohérence complète de la mémoire)**.** *Le programme résultant de la Transformation 3.7 est équivalent, d'après la Relation 4.3, au programme avant transformation. Cette équivalence est vérifiable au niveau des $\langle block \rangle$ d'une fonction.*

$$\forall b \in \langle block \rangle, \forall \sigma_1, \sigma_2,$$
$$\sigma_1 \approx_{1\to P} \sigma_2 \Rightarrow \mathbb{S}[\![b]\!]\langle\sigma_1\rangle \approx_{1\to P} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b]\!]]\!]\langle\sigma_2\rangle$$

Démonstration. Seule la preuve pour un $\langle block \rangle$ de tâche est décrite.
Soit σ_1, σ_2 tel que $\sigma_1 \approx_{1 \to P} \sigma_2$.
Soit $b_t \in (\langle statements \rangle, p)$.
D'après la sémantique sur les instructions,

$$\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!] \overset{(3.38)}{=} b_t\ ;(\bigcup_{id \in Id_0} S_{cp}(id), p)$$

$$\mathbb{S}[\![b_t; (\bigcup_{id \in Id_0} S_{cp}(id), p)]\!]\langle\sigma_2\rangle \overset{(3.13)}{=} \mathbb{S}[\![(\bigcup_{id \in Id_0} S_{cp}(id), p)]\!]\langle\mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle\rangle$$
$$\overset{(3.14)}{=} \mathbb{S}[\![\bigcup_{id \in Id_0} S_{cp}(id)]\!]\langle\mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle\rangle$$

$\mathbb{S}[\![\bigcup_{id \in Id_0} S_{cp}(id)]\!]\langle\mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle\rangle \overset{(4.3)}{\Rightarrow}$ d'après le Lemme 4.1,
$\forall x \in Id_0, \forall p \in [0..P[, \forall i \in \mathbb{N},$
$\mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x,i) = \mathbb{S}[\![\bigcup_{id \in Id_0} S_{cp}(id)]\!]\langle\mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle\rangle(x,i)$
$= \mathbb{S}[\![\bigcup_{id \in Id_0} S_{cp}(id)]\!]\langle\mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle\rangle(x_p,i)$

$\Rightarrow$ par définition de la Transformation 3.7 $\mathcal{T}_{\mathbb{S}_{com}}$,
$\forall x \in Id_0, \forall p \in [0..P[, \forall i \in \mathbb{N},$
$\mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x,i)$
$= \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_p,i)$

Par définition de la Transformation 3.7, $\mathcal{T}_{\mathbb{S}_{com}}$ ne modifie pas les instructions internes d'un $\langle block \rangle$, donc comme $\sigma_1 \approx_{1 \to P} \sigma_2 \Rightarrow \sigma_1 \approx_0 \sigma_2$, on a $\mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle \approx_0 \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle$.

Par définition de la Relation 4.3 $\approx_0$,
$\mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle \approx_0 \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle \overset{(4.1)}{\Rightarrow} \forall x \in Id_0, \forall i \in \mathbb{N}, \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x,i)$

D'après les égalités précédentes,

$\forall x \in Id_0, \forall i \in \mathbb{N},$
$\left.\begin{array}{l} \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x,i) \wedge \\ \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x,i) \end{array}\right\}$
$\Rightarrow \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x,i)$
$\overset{(4.1)}{\Rightarrow} \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle \approx_0 \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle$ par définition de $\approx_0$

De même pour x_p,

$\forall x \in Id_0, \forall p \in [0..P[, \forall i \in \mathbb{N},$
$\left.\begin{array}{l} \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x,i) \wedge \\ \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_p,i) \end{array}\right\}$
$\Rightarrow \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_p,i)$

D'après les deux précédents résultats et la définition de la Relation 4.3 $\approx_{1 \to P}$,

$\left.\begin{array}{l} \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle \approx_0 \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle \\ \forall x \in Id_0, \forall p \in [0..P[, \forall i \in \mathbb{N}, \\ \quad \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_p,i) \end{array}\right\} \Rightarrow \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle \approx_{1 \to P} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle$

□

La seconde transformation, permettant de garantir la cohérence mémoire, est la Transformation 3.8, rappelée ci-dessous. Le programme résultant de cette transformation possède les mêmes propriétés d'équivalence que le programme résultant de la transformation précédente.

Transformation 3.8 (Cohérence de la mémoire entre tâches)**.**

$$b \in (\langle statements \rangle, p),$$
$$\mathcal{T}_{\mathbb{S}_{com}}[\![b]\!] = b\,;\,(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id), p)$$

Proposition 4.4 (Cohérence de la mémoire)**.** *Le programme résultant de la Transformation 3.8 est équivalent par $\approx_{1 \to P}$, d'après la Relation 4.3, au programme avant transformation. Cette équivalence est vérifiable au niveau des $\langle block \rangle$ d'une fonction.*

$$\forall b \in \langle block \rangle, \forall \sigma_1, \sigma_2,$$
$$\sigma_1 \approx_{1 \to P} \sigma_2 \Rightarrow \mathbb{S}[\![b]\!]\langle\sigma_1\rangle \approx_{1 \to P} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b]\!]]\!]\langle\sigma_2\rangle$$

Démonstration. Comme précédemment, seule la preuve pour un $\langle block \rangle$ de tâche est décrite.
Soit σ_1, σ_2 tel que $\sigma_1 \approx_{1 \to P} \sigma_2$.
Soit $b_t \in (\langle statements \rangle, p)$.
De façon similaire à la preuve pour la Proposition 4.3, on obtient

$$\mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle \approx_0 \mathbb{S}[\![b_t\,;\,(\bigcup_{id \in \overline{WRITE}(b_t)} S_{cp}(id), p)]\!]\langle\sigma_2\rangle$$
$$\Rightarrow \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle \approx_0 \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle$$

$$\forall x \in \overline{WRITE}(b_t), \forall p \in [0..P[, \forall i \in \mathbb{N},$$
$$\mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![b_t\,;\,(\bigcup_{id \in \overline{WRITE}(b_t)} S_{cp}(id), p)]\!]\langle\sigma_2\rangle(x_p, i)$$
$$\Rightarrow \forall x \in \overline{WRITE}(b_t), \forall p \in [0..P[, \forall i \in \mathbb{N},$$
$$\mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_p, i)$$

Il reste à montrer que :

$$\forall x \in Id_0 \wedge x \notin \overline{WRITE}(b_t), \forall p \in [0..P[, \forall i \in \mathbb{N},$$
$$\mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_p, i)$$

Soit $x \in Id_0 \wedge x \notin \overline{WRITE}(b_t)$, $p \in [0..P[$ et $i \in \mathbb{N}$,
D'après la Propriété 3.2 sur $\overline{WRITE}$, $\sigma_1(x,i) = \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i)$
Or x_k n'apparaît pas dans b_t (x_k est une variable fraîche à sa création), donc

$$\begin{aligned}\sigma_2(x_k, i) &= \mathbb{S}[\![b_t]\!]\langle\sigma_2\rangle(x_k, i)\\ &= \mathbb{S}[\![b_t\,;\,(\bigcup_{id \in \overline{WRITE}(b_t)} S_{cp}(id), p)]\!]\langle\sigma_2\rangle(x_k, i)\\ &= \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_k, i)\end{aligned}$$

Par définition de la Relation 4.3 $\approx_{1\to P}$,

$$\sigma_1 \approx_{1\to P} \sigma_2 \overset{(4.2)}{\Rightarrow} \sigma_1(x,i) = \sigma_2(x_k,i)$$

$$\forall x \in Id_0 \wedge x \notin \overline{WRITE}(b_t), \forall p \in [0..P[, \forall i \in \mathbb{N},$$
$$\left.\begin{array}{l} \sigma_1(x,i) = \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) \\ \sigma_2(x_k,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_k,i) \\ \sigma_1(x,i) = \sigma_2(x_k,i) \end{array}\right\} \Rightarrow \mathbb{S}[\![b_t]\!]\langle\sigma_1\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b_t]\!]]\!]\langle\sigma_2\rangle(x_k,i)$$

Donc $\mathbb{S}[\![b]\!]\langle\sigma_1\rangle \approx_{1\to P} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{com}}[\![b]\!]]\!]\langle\sigma_2\rangle$
CQFD. □

Quelque soit la transformation $\mathcal{T}_{\mathbb{S}_{com}}$ utilisée, 3.7 ou 3.8, il en résulte un programme équivalent garantissant la cohérence mémoire sur ce nouveau programme pour les variables répliquées.

4.1.3 Matérialisation du placement d'une tâche sur un processus

La matérialisation du placement d'une tâche sur un processus a été décrite en section 3.3.3. Sa définition est rappelée ci-dessous.

Transformation 3.9 (Substitution des variables d'origine par leur variable répliquée)**.**

$$\begin{array}{l} s \in \langle statements\rangle,\ p \in \mathbb{N}, \\ t \in \langle type\rangle,\ id \in \langle identifier\rangle, \end{array}$$
$$\mathcal{T}_{\mathbb{S}_{map}}[\![t\ id]\!]\langle\_\rangle = t\ id \tag{4.4}$$
$$\mathcal{T}_{\mathbb{S}_{map}}[\![(s,p)]\!]\langle\_\rangle = (\mathcal{T}_{\mathbb{S}_{map}}[\![s]\!]\langle p\rangle, p) \tag{4.5}$$
$$\mathcal{T}_{\mathbb{E}_{map}}[\![id]\!]\langle p\rangle = \text{if } id \in Id_0 \text{ then } id_p \text{ else } id \tag{4.6}$$

La Relation 4.4 $\approx_{1\rightsquigarrow p}$ est une relation d'équivalence entre deux états mémoire où une permutation des variables considérées est faite. En l'occurrence, l'équivalence est faite entre l'ensemble des variables d'origine Id_0 et l'ensemble des variables correspondantes sur un processus p donné. Cette équivalence est appelée équivalence par substitution. La Relation 4.4 permet de garantir la Propriété 3.3 du chapitre précédent.

Relation 4.4 (Équivalence par substitution)**.** On définit la relation d'équivalence $\approx_{1\rightsquigarrow p}$ par substitution d'une variable d'origine par sa variable correspondante sur le processus p entre 2 états mémoire par :

$$\sigma \approx_{1\rightsquigarrow p} \sigma' \overset{\text{def}}{=} \forall x \in Id_0, \forall i \in \mathbb{N}, \sigma(x,i) = \sigma'(x_p,i) \tag{4.7}$$

Remarque 4.2. L'équivalence avec cohérence mémoire précédente, Relation 4.3, peut s'exprimer en utilisant cette nouvelle relation d'équivalence :

$$\sigma \approx_{1\to P} \sigma' \overset{\text{def}}{=} \begin{cases} \sigma \approx_0 \sigma' \\ \wedge\ \forall p \in [0..P[, \sigma \approx_{1\rightsquigarrow p} \sigma' \end{cases}$$

Proposition 4.5 (Substitution dans une tâche)**.** *Chaque tâche du programme générée par la Transformation 3.9 est équivalente par substitution à la tâche avant transformation.*

$$\forall s \in \langle statements \rangle \cup \langle statement \rangle, \forall \sigma_2, \sigma_3, \forall p \in [0..P[,$$
$$\sigma_2 \approx_{1\rightsquigarrow p} \sigma_3 \Rightarrow \mathbb{S}[\![s]\!]\langle\sigma_2\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s]\!]\langle p\rangle]\!]\langle\sigma_3\rangle$$

Démonstration. Une preuve par induction de la Proposition 4.5 est réalisée. Les différents cas de l'induction sont traités successivement.
Soit σ_2, σ_3 tel que $\sigma_2 \approx_{1\rightsquigarrow p} \sigma_3$.

- Soit $lhs \in \langle variable \rangle, rhs \in \langle expression \rangle$.

$$\mathcal{T}_{\mathbb{S}_{map}}[\![lhs \leftarrow rhs]\!]\langle p\rangle = \mathcal{T}_{\mathbb{E}_{map}}[\![lhs]\!]\langle p\rangle \leftarrow \mathcal{T}_{\mathbb{E}_{map}}[\![rhs]\!]\langle p\rangle$$

Pour chaque idr de rhs,
soit il n'appartient pas à Id_0 et reste inchangé après $\mathcal{T}_{\mathbb{E}_{map}}[\![rhs]\!]\langle p\rangle$,
donc $\mathbb{E}[\![\mathcal{T}_{\mathbb{E}_{map}}[\![idr]\!]\langle p\rangle]\!]\langle\sigma_3\rangle = \mathbb{E}[\![idr]\!]\langle\sigma_3\rangle = \mathbb{E}[\![idr]\!]\langle\sigma_2\rangle$
soit il appartient à Id_0 et devient idr_p après $\mathcal{T}_{\mathbb{E}_{map}}[\![rhs]\!]\langle p\rangle$, or $\sigma_2 \approx_{1\rightsquigarrow p} \sigma_3$,
donc $\mathbb{E}[\![\mathcal{T}_{\mathbb{E}_{map}}[\![idr]\!]\langle p\rangle]\!]\langle\sigma_3\rangle = \mathbb{E}[\![idr_p]\!]\langle\sigma_3\rangle = \mathbb{E}[\![idr]\!]\langle\sigma_2\rangle$
Donc $\mathbb{E}[\![\mathcal{T}_{\mathbb{E}_{map}}[\![rhs]\!]\langle p\rangle]\!]\langle\sigma_3\rangle = \mathbb{E}[\![rhs]\!]\langle\sigma_2\rangle$

Si l'identifiant idl de lhs n'appartient pas à Id_0,
alors il est inchangé et on a $\mathbb{S}[\![lhs \leftarrow rhs]\!]\langle\sigma_2\rangle = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![lhs \leftarrow rhs]\!]\langle p\rangle]\!]\langle\sigma_3\rangle$

Si l'identifiant idl de lhs appartient à Id_0,
alors il devient idl_p et on a $\mathbb{S}[\![lhs \leftarrow rhs]\!]\langle\sigma_2\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![lhs \leftarrow rhs]\!]\langle p\rangle]\!]\langle\sigma_3\rangle$

Donc dans tous les cas, on a bien

$$\mathbb{S}[\![lhs \leftarrow rhs]\!]\langle\sigma_2\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![lhs \leftarrow rhs]\!]\langle p\rangle]\!]\langle\sigma_3\rangle$$

- Soit $s_1 \in \langle statement \rangle, s_2 \in \langle statements \rangle$.
Par récurrence, supposons

$$H1 : \sigma_{2_1} \approx_{1\rightsquigarrow p} \sigma_{3_1} \Rightarrow \mathbb{S}[\![s_1]\!]\langle\sigma_{2_1}\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s_1]\!]\langle p\rangle]\!]\langle\sigma_{3_1}\rangle$$
$$H2 : \sigma_{2_2} \approx_{1\rightsquigarrow p} \sigma_{3_2} \Rightarrow \mathbb{S}[\![s_2]\!]\langle\sigma_{2_2}\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s_2]\!]\langle p\rangle]\!]\langle\sigma_{3_2}\rangle$$
$$\begin{aligned}\sigma_2 \approx_{1\rightsquigarrow p} \sigma_3 &\overset{H1}{\Rightarrow} \mathbb{S}[\![s_1]\!]\langle\sigma_2\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s_1]\!]\langle p\rangle]\!]\langle\sigma_3\rangle \\ &\overset{H2}{\Rightarrow} \mathbb{S}[\![s_2]\!]\langle\mathbb{S}[\![s_1]\!]\langle\sigma_2\rangle\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s_2]\!]\langle p\rangle]\!]\langle\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s_1]\!]\langle p\rangle]\!]\langle\sigma_3\rangle\rangle \\ &\overset{(3.13)}{\Rightarrow} \mathbb{S}[\![s_1;\ s_2]\!]\langle\sigma_2\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s_1;\ s_2]\!]\langle p\rangle]\!]\langle\sigma_3\rangle\end{aligned}$$

- De même, par récurrence, on prouve l'équivalence pour les tests et les boucles.

□

Cependant, la Relation 4.4 d'équivalence ne permet pas de mettre en relation le résultat de la Transformation 3.9 dans son ensemble, mais seulement les tâches qui le composent.

La Relation 4.5 $\approx_{new}$ définit une relation d'équivalence entre deux programmes pour l'ensemble des variables répliquées. Elle est similaire à la Relation 4.1 à ceci prêt qu'elle porte sur l'ensemble Id_{new} au lieu de l'ensemble Id_0.

Relation 4.5 (Équivalence sur Id_{new})**.** On définit la relation d'équivalence $\approx_{new}$ sur l'ensemble des variables d'origine Id_{new} entre deux états mémoire par :

$$\sigma \approx_{new} \sigma' \stackrel{\text{def}}{=} \forall x \in Id_{new}, \forall i \in \mathbb{N}, \sigma(x,i) = \sigma'(x,i) \tag{4.8}$$

Proposition 4.6 (Substitution pour l'ensemble du programme)**.** *Considérant un premier programme obtenu après application des transformations 3.5 et 3.8, le nouveau programme résultant de la Transformation 3.9 est équivalent, d'après la Relation 4.5, à ce premier. Cette équivalence est vérifiable après chaque tâche de calculs suivie d'une tâche de copies.*

$$\begin{array}{l}\forall t \in \langle type\rangle, \forall id \in \langle identifier\rangle, \forall b_1, b_2 \in \langle block\rangle, \forall s \in \langle statements\rangle, \forall p \in \mathbb{N},\\ \quad \mathcal{P}(t\ id) \equiv true\\ \quad \mathcal{P}(b_1;\ b_2) \equiv \mathcal{P}(b_1) \wedge \mathcal{P}(b_2)\\ \quad \mathcal{P}((s,\ p);\ (\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)) \equiv true\end{array}$$

Ce prédicat $\mathcal{P}$ permet de garantir que les transformations 3.5 et 3.8, ou des transformations équivalentes, ont été réalisées.

$$\begin{array}{l}\forall b \in \langle block\rangle \text{ tel que } \mathcal{P}(b), \forall \sigma_2, \sigma_3,\\ \quad \sigma_2 \approx_{new} \sigma_3 \Rightarrow \mathbb{S}[\![b]\!]\langle\sigma_2\rangle \approx_{new} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle\end{array}$$

Démonstration. La Proposition 4.6 est prouvée par induction sur les différents cas possible.
Soit $b \in \langle block\rangle$ tel que $\mathcal{P}(b)$.
Soit σ_2, σ_3 tel que $\sigma_2 \approx_{new} \sigma_3$.

- $b \in \langle fun\text{-}declaration\rangle \Rightarrow \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle = \mathbb{S}[\![b]\!]\langle\sigma_3\rangle$
  $\Rightarrow \mathbb{S}[\![b]\!]\langle\sigma_2\rangle \approx_{new} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle$
- $b = b_1;\ b_2 \Rightarrow$ par induction, on a $\mathbb{S}[\![b]\!]\langle\sigma_2\rangle \approx_{new} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle$
- $b = (s,\ p);\ (\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)$ avec $s \in \langle statements\rangle$ et $p \in \mathbb{N}$

$$\begin{array}{rl}\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle & \stackrel{(3.42)}{=} (\mathcal{T}_{\mathbb{S}_{map}}[\![s]\!]\langle p\rangle,\ p)\\ & \Rightarrow \text{ d'après Proposition 4.5, } \mathbb{S}[\![s]\!]\langle\sigma_2\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![s]\!]\langle p\rangle]\!]\langle\sigma_3\rangle\\ & \Rightarrow \mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle \approx_{1\rightsquigarrow p} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle\\ & \stackrel{(4.7)}{\Rightarrow} \text{ d'après la définition de la Relation 4.4 } \approx_{1\rightsquigarrow p},\\ & \quad \forall x \in Id_0, \forall i \in \mathbb{N},\\ & \quad\quad \mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle(x,i) = \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle(x_p,i)\end{array}$$

D'après le Lemme 4.1,
$\forall x \in Id_0, \forall 0 \leq k < P, \forall i \in \mathbb{N},$

$$\begin{array}{rl}\mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle(x,i) & = \mathbb{S}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle\rangle(x,i)\\ & = \mathbb{S}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle\rangle(x_k,i)\end{array}$$

d'après Lemme 4.1 et la substitution (3.43) de la Transformation 3.9 $\mathcal{T}_{\mathbb{E}_{map}}$,
$\forall x \in Id_0, \forall 0 \leq k < P, \forall i \in \mathbb{N}$,
$\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle(x_p, i)$
$= \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\_\rangle]\!]\langle\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle\rangle(x_p, i)$
$= \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\_\rangle]\!]\langle\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle\rangle(x_k, i)$

$\Rightarrow$ d'après les égalités précédemment calculées,
$\forall x \in Id_0, \forall 0 \leq k < P, \forall i \in \mathbb{N}$,
$\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\_\rangle]\!]\langle\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle\rangle(x_k, i)$
$= \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle(x_p, i)$
$= \mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle(x, i)$
$= \mathbb{S}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle\rangle(x_k, i)$

$\Rightarrow$ d'après la Définition 3.4 Id_{new},
$\forall x \in Id_{new}, \forall i \in \mathbb{N}$,
$\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\_\rangle]\!]\langle\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![(s,\ p)]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle\rangle(x, i)$
$= \mathbb{S}[\![(\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)]\!]\langle\mathbb{S}[\![(s,\ p)]\!]\langle\sigma_2\rangle\rangle(x, i)$

$\Rightarrow$ d'après la Relation 4.5 $\approx_{new}$,
$\mathbb{S}[\![b]\!]\langle\sigma_2\rangle \approx_{new} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle$

□

4.1.4 Nettoyage du code

Le nettoyage du code est composé de deux transformations décrites en section 3.3.4, le nettoyage des déclarations des variables qui ne sont plus utilisées et la suppression des affectations correspondant à une identité.

4.1.4.1 Nettoyage des déclarations

La première transformation consistant à supprimer les déclarations des variables d'origine est rappelée ci-dessous.

Transformation 3.10 (Nettoyage des déclarations d'origine)**.**

$$t \in \langle type\rangle,\ id \in \langle identifier\rangle,\quad \mathcal{T}_{\mathbb{S}_{idc}}[\![t\ id]\!] = \text{if } id \in Id_0 \text{ then } \varnothing \text{ else } t\ id$$

Proposition 4.7 (Nettoyage des déclarations d'origine)**.** *La Transformation 3.10 donne un programme équivalent, d'après la Relation 4.5, au programme avant transformation. Cette équivalence est vérifiable au niveau de toutes les instructions, $\langle block\rangle$, $\langle statements\rangle$ et $\langle statement\rangle$, d'une fonction.*

$$\forall s \in \langle block\rangle \cup \langle statements\rangle \cup \langle statement\rangle, \forall \sigma_3, \sigma_4,$$
$$\sigma_3 \approx_{new} \sigma_4 \Rightarrow \mathbb{S}[\![s]\!]\langle\sigma_3\rangle \approx_{new} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{idc}}[\![s]\!]]\!]\langle\sigma_4\rangle$$

Démonstration. Triviale étant donné que la Transformation 3.10 ne modifie pas l'état mémoire du programme pour les variables répliquées, appartenant à Id_{new}.
□

En considérant la suite de transformations précédentes, la Transformation 3.10 est correcte car toutes les utilisations, en écriture et en lecture, des variables appartenant à Id_0 ont été remplacées par leurs correspondantes sur les différents processus lors de la Transformation 3.9 de substitution. On peut le vérifier, soit en analysant syntaxiquement le code et constater l'absence de leur utilisation, soit en analysant la valeur des variables d'origine à la fin du programme qui est $\perp$, ce qui dénote une déclaration de variable sans affectation de valeur. Cette caractéristique s'exprime dans la Propriété 4.1. Cette dernière peut être ajoutée à la Proposition 4.6 précédente pour devenir la Proposition 4.8.

Propriété 4.1 (Variable inutilisée)**.** *Les variables d'origine ne sont plus utilisées après application de la Transformation 3.9.*

$$\forall x \in Id_0, \forall i \in \mathbb{N}, \sigma(x,i) = \perp$$

Proposition 4.8 (Substitution pour le programme 2)**.** *Considérant un premier programme obtenu après les transformations 3.5 et 3.8, le nouveau programme résultant de la Transformation 3.9 est équivalent, d'après la Relation 4.5, à ce premier. Cette équivalence est vérifiable après chaque tâche de calculs suivie d'une tâche de copies.*

$$\begin{array}{l}\forall t \in \langle type\rangle, \forall id \in \langle identifier\rangle, \forall b_1, b_2 \in \langle block\rangle, \forall s \in \langle statements\rangle, \forall p \in \mathbb{N},\\ \quad \mathcal{P}(t\ id) \equiv true\\ \quad \mathcal{P}(b_1;\ b_2) \equiv \mathcal{P}(b_1) \wedge \mathcal{P}(b_2)\\ \quad \mathcal{P}((s,\ p);\ (\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)) \equiv true\end{array}$$

Ce prédicat $\mathcal{P}$ permet de garantir que les transformations 3.5 et 3.8, ou des transformations équivalentes, ont été réalisées.

$$\begin{array}{l}\forall b \in \langle block\rangle \text{ tel que } \mathcal{P}(b), \forall \sigma_2, \sigma_3, \forall x \in Id_0, \forall i \in \mathbb{N},\\ \left.\begin{array}{r}\sigma_2 \approx_{new} \sigma_3\\ (\sigma_3(x,i) = \perp \vee \sigma_3(x,i) = \varepsilon)\end{array}\right\} \Rightarrow \begin{cases}\mathbb{S}[\![b]\!]\langle\sigma_2\rangle \approx_{new} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle\\ (\mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle(x,i) = \perp\\ \quad \vee \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{map}}[\![b]\!]\langle\_\rangle]\!]\langle\sigma_3\rangle(x,i) = \varepsilon)\end{cases}\end{array}$$

Démonstration. Similaire à la preuve de la Proposition 4.6, en remarquant de plus que les variables d'origine ne peuvent qu'être déclarées et non modifiées dans le programme transformé. □

4.1.4.2 Élimination des identités

La deuxième transformation consiste à supprimer les affectations correspondant à des identités. Elle est rappelée ci-dessous.

Transformation 3.11 (Élimination des identités)**.**

$$var \in \langle variable\rangle, \qquad \mathcal{T}_{\mathbb{S}_{eid}}[\![var \leftarrow var]\!] = \varnothing$$

La Relation 4.6 $\approx$ définit une relation d'équivalence classique entre deux programmes sur l'ensemble des variables des programmes.

Relation 4.6 (Équivalence complète)**.** On définit la relation d'équivalence $\approx$ sur l'ensemble des variables $\langle identifier \rangle$ entre deux états mémoire par :

$$\sigma \approx \sigma' \overset{\text{def}}{=} \forall x \in \langle identifier \rangle, \forall i \in \mathbb{N}, \sigma(x,i) = \sigma'(x,i) \tag{4.9}$$

La Transformation 3.11 est totalement indépendante de toutes les autres transformations et peut s'appliquer dans le cas général d'optimisations de programme. La Proposition 4.9 permet de garantir cette utilisation générale.

Proposition 4.9 (Élimination des identités)**.** *La Transformation 3.11 donne un programme complètement équivalent, d'après la Relation 4.6, au programme avant sa transformation. Cette équivalence est vérifiable au niveau de toutes les instructions, $\langle block \rangle$, $\langle statements \rangle$ et $\langle statement \rangle$, d'une fonction.*

$$\begin{aligned} &\forall s \in \langle block \rangle \cup \langle statements \rangle \cup \langle statement \rangle, \forall \sigma_4, \sigma_5, \\ &\quad \sigma_4 \approx \sigma_5 \Rightarrow \mathbb{S}[\![s]\!]\langle\sigma_4\rangle \approx \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{eid}}[\![s]\!]]\!]\langle\sigma_5\rangle \end{aligned}$$

Démonstration. Soit σ_4, σ_5 tel que $\sigma_4 \approx \sigma_5$.
Soit $var \in \langle variable \rangle$ et $s \in \langle statements \rangle$.

$$\begin{aligned} \mathbb{S}[\![var \leftarrow var]\!]\langle\sigma_4\rangle &\overset{(3.15)}{=} update(\sigma_4, \mathbb{L}[\![var]\!]\langle\sigma\rangle, \mathbb{E}[\![var]\!]\langle\sigma\rangle) \\ &= \sigma_4 \\ \Rightarrow \mathbb{S}[\![var \leftarrow var;\ s]\!]\langle\sigma_4\rangle &\overset{(3.13)}{=} \mathbb{S}[\![s]\!]\langle\mathbb{S}[\![var \leftarrow var]\!]\langle\sigma_4\rangle\rangle \\ &= \mathbb{S}[\![s]\!]\langle\sigma_4\rangle \\ \Rightarrow \mathbb{S}[\![var \leftarrow var;\ s]\!]\langle\sigma_4\rangle &\approx \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{eid}}[\![var \leftarrow var;\ s]\!]]\!]\langle\sigma_5\rangle \end{aligned}$$

□

4.2 La preuve sur l'optimisation de code

La seule optimisation utilisée à ce niveau est une élimination de code mort. C'est une optimisation classique de programme très largement connue [9]. Nous ne ferons donc pas de preuves de sa correction. Néanmoins, les propriétés sur le code obtenu sont décrites.

Pour réaliser une élimination de code mort, l'utilisation d'une analyse de variables vivantes est effectuée. La Définition 4.3 définit une variable vivante. On définit $LIVE(s)$ l'ensemble des variables vivantes après l'exécution de l'instruction s.

Définition 4.3 (Variable vivante)**.** Une variable est dite vivante si elle est lue dans la continuation du programme.

L'élimination de code mort permet de supprimer toutes les écritures de variables non vivantes. Elle est, en particulier, utile pour éliminer les copies de variables répliquées.

Relation 4.7 (Équivalence des variables répliquées vivantes)**.** On définit la relation d'équivalence $\approx_{LIVE(s)}$ sur l'ensemble des variables vivantes entre deux états mémoire par :

$$\sigma \approx_{LIVE(s)} \sigma' \stackrel{\text{def}}{=} \forall x \in LIVE(s), \exists id \in \langle identifier \rangle, \exists l \in [0..P[, \exists p \in [0..P[, p \neq l \\ \forall i \in \mathbb{N}, x = id_l \wedge \sigma(x,i) = \sigma'(id_p, i) \tag{4.10}$$

Proposition 4.10 (Élimination de code mort)**.** *Considérant le programme obtenu après les transformations précédentes, le nouveau programme résultant de l'élimination de code mort $\mathcal{T}_{\mathbb{S}_{dce}}$ est équivalent, d'après la Relation 4.7, au programme avant cette optimisation. Cette équivalence est vérifiable après chaque tâche de calculs suivie d'une tâche de copies.*

$$\begin{array}{l} \forall t \in \langle type \rangle, \forall id \in \langle identifier \rangle, \forall b_1, b_2 \in \langle block \rangle, \forall s \in \langle statements \rangle, \forall p \in \mathbb{N}, \\ \quad \mathcal{P}(\varnothing) \equiv true \\ \quad \mathcal{P}(t\ id) \equiv true \\ \quad \mathcal{P}(b_1;\ b_2) \equiv \mathcal{P}(b_1) \wedge \mathcal{P}(b_2) \\ \quad \mathcal{P}((s,\ p);\ (\bigcup_{id \in \overline{WRITE}(b)} S_{cp}(id),\ p)) \equiv true \\ \forall b \in \langle block \rangle\ tel\ que\ \mathcal{P}(b)\ et\ b = b_1;\ b_2, \forall \sigma_5, \sigma_6, \\ \quad \mathbb{S}[\![b_1]\!]\langle\sigma_5\rangle \approx_{LIVE(b_1)} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{dce}}[\![b_1]\!]]\!]\langle\sigma_6\rangle \Rightarrow \mathbb{S}[\![b]\!]\langle\sigma_5\rangle \approx_{LIVE(b_2)} \mathbb{S}[\![\mathcal{T}_{\mathbb{S}_{dce}}[\![b]\!]]\!]\langle\sigma_6\rangle \end{array}$$

L'élimination de code mort va particulièrement éliminer des instructions de copies générées par les fonctions S_{cp}. Les variables à gauche de ces instructions d'affectation sont, après l'élimination de code mort, toutes des variables vivantes.

4.3 Les preuves sur le code parallèle généré

La dernière phase de transformations à prouver est la génération du code parallèle. Elle est faite en deux temps. Dans un premier temps, le code parallèle est vérifié équivalent au code séquentiel. Puis, on montre que ce code peut exécuter des tâches simultanément.

4.3.1 Passage à la sémantique parallèle

La Transformation 3.14 permet de passer de la syntaxe séquentielle à la syntaxe parallèle que nous avons définie dans le chapitre précédent. Cette transformation est rappelée ci-dessous.

Transformation 3.14 (Du séquentiel au parallèle)**.**

$$
\begin{aligned}
S_t \in \langle statements\rangle (S_t \neq S_{cp}),\ decl \in \langle fun\text{-}declaration\rangle,\ b_1, b_2 \in \langle block\rangle \\
\mathcal{T}_{\mathbb{S}_{s2p}}[\![\varnothing]\!] &= \varnothing \\
\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1\ ; b_2]\!] &= \mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1]\!]\ ; \mathcal{T}_{\mathbb{S}_{s2p}}[\![b_2]\!] \\
\mathcal{T}_{\mathbb{S}_{s2p}}[\![decl]\!] &= decl \\
\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_t, p)]\!] &= (\underbrace{\varnothing, \ldots, \varnothing}_{p}, S_t, \underbrace{\varnothing, \ldots, \varnothing}_{P-p-1}) \\
\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_{cp}, p)]\!] &= \bigcup_{s \in S_{cp}} copy2com(s)
\end{aligned}
$$

La Relation 4.8 $\approx_{\hookrightarrow /\!/}$ définit une relation d'équivalence entre un programme séquentiel et un programme parallèle. Elle permet d'exprimer le fait que chacune des variables répliquées, devant être affectée à un processus, est effectivement affectée à celui-ci. De même pour les instructions, celles-ci sont exécutées par le processus désiré concernant les tâches de calculs et les communications sont bien effectuées par les processus devant émettre et recevoir.

Relation 4.8 (Équivalence de mise en parallèle)**.** On définit la relation d'équivalence $\approx_{\hookrightarrow /\!/}$ entre un état mémoire séquentiel et un état mémoire parallèle, correspondant à un ensemble d'état mémoire, par :

$$\sigma \approx_{\hookrightarrow /\!/}\ {}^{P}_{0}\sigma' \stackrel{\text{def}}{=} \forall id \in Id_0, \forall p \in [0..P[, \forall i \in \mathbb{N}, \sigma(id_p, i) = \sigma'^{p}(id_p, i) \qquad (4.11)$$

Proposition 4.11. *Le programme parallèle résultant de la Transformation 3.14 est équivalent, d'après la Relation 4.8, au programme séquentiel avant transformation. Cette équivalence est vérifiable au niveau des $\langle block\rangle$ d'une fonction.*

$$
\begin{aligned}
&\forall b \in \langle block\rangle, \forall \sigma_6, {}^{P}_{0}\sigma_7, \\
&\quad \sigma_6 \approx_{\hookrightarrow /\!/}\ {}^{P}_{0}\sigma_7 \Rightarrow \mathbb{S}[\![b]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle
\end{aligned}
$$

Démonstration. Une preuve par induction sur les instruction d'origine est réalisée.
Soit $\sigma_6, {}^{P}_{0}\sigma_7$ tel que $\sigma_6 \approx_{\hookrightarrow /\!/}\ {}^{P}_{0}\sigma_7$.

- Soit $decl \in \langle fun\text{-}declaration\rangle$.

$$
\begin{aligned}
\mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![decl]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle &\overset{(3.56)}{=} \mathbb{P}[\![decl]\!]\langle {}^{P}_{0}\sigma_7\rangle \\
&\overset{(3.48)}{=} (\mathbb{S}[\![decl]\!]\langle\sigma_7^0\rangle, \ldots, \mathbb{S}[\![decl]\!]\langle\sigma_7^p\rangle, \ldots, \mathbb{S}[\![decl]\!]\langle\sigma_7^{P-1}\rangle) \\
\Rightarrow \mathbb{S}[\![decl]\!]\langle\sigma_6\rangle &\approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![decl]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle
\end{aligned}
$$

- Soit $b_1, b_2 \in \langle block \rangle$.

Par induction,

$$H1 : \sigma_{6_1} \approx^{P}_{\hookrightarrow /\!/ 0} \sigma_{7_1} \Rightarrow \mathbb{S}[\![b_1]\!]\langle\sigma_{6_1}\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1]\!]]\!]\langle {}^{P}_{0}\sigma_{7_1}\rangle$$
$$H2 : \sigma_{6_2} \approx^{P}_{\hookrightarrow /\!/ 0} \sigma_{7_2} \Rightarrow \mathbb{S}[\![b_2]\!]\langle\sigma_{6_2}\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_2]\!]]\!]\langle {}^{P}_{0}\sigma_{7_2}\rangle$$
$$\sigma_6 \approx^{P}_{\hookrightarrow /\!/ 0} \sigma_7 \overset{H1}{\Rightarrow} \mathbb{S}[\![b_1]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle$$
$$\overset{H2}{\Rightarrow} \mathbb{S}[\![b_2]\!]\langle\mathbb{S}[\![b_1]\!]\langle\sigma_6\rangle\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_2]\!]]\!]\langle\mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle\rangle$$
$$\overset{(3.13)}{\Rightarrow} \mathbb{S}[\![b_1;\ b_2]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_2]\!]]\!]\langle\mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle\rangle$$
$$\overset{(3.47)}{\Rightarrow} \mathbb{S}[\![b_1;\ b_2]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![b_1;\ b_2]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle$$

- Soit $S_t \in \langle statements \rangle$ tel que $S_t \neq S_{cp}$, $p \in \mathbb{N}$ tel que $p \in [0..P[$.

Soit ${}^{P}_{0}\sigma'_7 = \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_t,\ p)]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle$.

$$\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_t,\ p)]\!] \overset{(3.57)}{=} (\underbrace{\varnothing, \ldots, \varnothing}_{p}, S_t, \underbrace{\varnothing, \ldots, \varnothing}_{P-p-1})$$
$$\Rightarrow \mathbb{P}[\![(\underbrace{\varnothing, \ldots, \varnothing}_{p}, S_t, \underbrace{\varnothing, \ldots, \varnothing}_{P-p-1})]\!]\langle {}^{P}_{0}\sigma_7\rangle \overset{(3.49)}{=} (\sigma_7^0, \ldots, \sigma_7^{p-1}, \mathbb{S}[\![S_t]\!]\langle\sigma_7^p\rangle, \sigma_7^{p+1}, \ldots, \sigma_7^{P-1})$$
$$\Rightarrow {}^{P}_{0}\sigma'_7 = (\sigma_7^0, \ldots, \sigma_7^{p-1}, \mathbb{S}[\![S_t]\!]\langle\sigma_7^p\rangle, \sigma_7^{p+1}, \ldots, \sigma_7^{P-1})$$
$$\mathbb{S}[\![(S_t,\ p)]\!]\langle\sigma_6\rangle \overset{(3.14)}{=} \mathbb{S}[\![S_t]\!]\langle\sigma_6\rangle$$
$\Rightarrow$ d'après les deux égalités précédentes,
$\forall id \in Id_0,\ \mathbb{S}[\![(S_t,\ p)]\!]\langle\sigma_6\rangle(id_p, i) = \sigma'^{p}_{7}(id_p, i)$

or $\sigma_6 \approx^{P}_{\hookrightarrow /\!/ 0} \sigma_7$
$\Rightarrow \forall k \neq p, \forall id \in Id_0,\ \mathbb{S}[\![(S_t,\ p)]\!]\langle\sigma_6\rangle(id_k, i) = \sigma'^{k}_{7}(id_k, i)$
$\Rightarrow$ d'après la définition de la Relation 4.8 $\approx_{\hookrightarrow /\!/}$,
$\mathbb{S}[\![(S_t,\ p)]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_t,\ p)]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle$

- Soit S_{cp}, et $p \in \mathbb{N}$ tel que $p \in [0..P[$.

Soit ${}^{P}_{0}\sigma'_7 = \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_{cp},\ p)]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle$.

$$\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_{cp},\ p)]\!] \overset{(3.58)}{=} \bigcup_{s \in S_{cp}} copy2com(s)$$

Soit $id \in Id_0$, $k \in \mathbb{N}$ tel que $0 \leq k < P$,
d'après la Définition 3.13 de la fonction $copy2com$,

$$copy2com(id_k \leftarrow id_p) \overset{(3.51)}{=} \mathrm{com}(\ p\ ;\ k\ ;\ id_p\ ;\ id_k\ )$$
$$\Rightarrow \mathbb{S}[\![id_k \leftarrow id_p]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathrm{com}(\ p\ ;\ k\ ;\ id_p\ ;\ id_k\ )]\!]\langle {}^{P}_{0}\sigma_7\rangle$$
$$\Rightarrow \mathbb{S}[\![id_k \leftarrow id_p]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![copy2com(id_k \leftarrow id_p)]\!]\langle {}^{P}_{0}\sigma_7\rangle$$

De même pour des éléments de structure ou les tableaux.

$$\Rightarrow \forall s \in S_{cp}, \mathbb{S}[\![s]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![copy2com(s)]\!]\langle {}^{P}_{0}\sigma_7\rangle$$
$$\Rightarrow \mathbb{S}[\![(S_{cp},\ p)]\!]\langle\sigma_6\rangle \approx_{\hookrightarrow /\!/} \mathbb{P}[\![\mathcal{T}_{\mathbb{S}_{s2p}}[\![(S_{cp},\ p)]\!]]\!]\langle {}^{P}_{0}\sigma_7\rangle$$

□

4.3.2 Mise en parallèle de l'exécution

Cette dernière étape effectue une transformation qui n'est pas totalement décrite formellement mais dont la Propriété 3.5 exprime ce qu'elle doit retourner.

Propriété 3.5 (Exécution locale non contrainte)**.**

$$\forall(..., S_a^j, ...), (..., S_b^j, ...) \in (\langle statements\rangle, \dots, \langle statements\rangle), \forall_0^P \sigma,$$
$$\text{if } \forall com(\,k\,;\,i\,;\,id_k\,;\,id_i\,) \text{ such that } i \neq j \wedge k \neq j, \text{ then}$$
$$\mathbb{P}\left[\!\!\left[\begin{array}{l}(..., S_a^j, ...)\,;\\ ...\,;\\ com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;\\ ...\,;\\ (..., S_b^j, ...)\end{array}\right]\!\!\right]\left\langle {}_0^P\,\sigma\right\rangle = \mathbb{P}\left[\!\!\left[\begin{array}{l}(..., S_a^j\,;S_b^j, ...)\,;\\ ...\,;\\ com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;\\ ...\,;\\ (..., \varnothing, ...)\end{array}\right]\!\!\right]\left\langle {}_0^P\,\sigma\right\rangle$$

Démonstration. Soit $(..., S_a^j, ...), (..., S_b^j, ...) \in (\langle statements\rangle, \dots, \langle statements\rangle)$ et quelles que soient les communications présentes entre ces deux tâches, elles ne concernent pas le processus j, *ie.* $\forall com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)$ tel que $\{(..., S_a^j, ...)\,;...\,;com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;...\,;(..., S_b^j, ...)\}$, $i \neq j$ et $k \neq j$.
Soit $(\sigma^0, ..., \sigma^j, ..., \sigma^{P-1})$.

$$\begin{aligned}
&\mathbb{P}[\![(..., S_a^j, ...)\,;...\,;com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;...\,;(..., S_b^j, ...)]\!\langle(\sigma^0, ..., \sigma^j, ..., \sigma^{P-1})\rangle\\
&\overset{(3.49)}{=} \mathbb{P}[\![...\,;com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;...\,;(..., S_b^j, ...)]\!\langle(..., \mathbb{S}[\![S_a^j]\!\langle\sigma^j\rangle, ...)\rangle\\
&\overset{(3.50)}{=} ... \overset{(3.50)}{=} \mathbb{P}[\![(..., S_b^j, ...)]\!\langle(..., \mathbb{S}[\![S_a^j]\!\langle\sigma^j\rangle, ...)\rangle\\
&\overset{(3.49)}{=} \mathbb{P}[\![\varnothing]\!\langle(..., \mathbb{S}[\![S_b^j]\!\langle\mathbb{S}[\![S_a^j]\!\langle\sigma^j\rangle\rangle, ...)\rangle\\
&\overset{(3.13)}{=} \mathbb{P}[\![\varnothing]\!\langle(..., \mathbb{S}[\![S_a^j\,;S_b^j]\!\langle\sigma^j\rangle, ...)\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathbb{P}[\![(..., S_a^j\,;S_b^j, ...)\,;...\,;com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;...\,;(..., \varnothing, ...)]\!\langle(\sigma^0, ..., \sigma^j, ..., \sigma^{P-1})\rangle\\
&\overset{(3.49)}{=} \mathbb{P}[\![...\,;com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;...\,;(..., \varnothing, ...)]\!\langle(..., \mathbb{S}[\![S_a^j\,;S_b^j]\!\langle\sigma^j\rangle, ...)\rangle\\
&\overset{(3.50)}{=} ... \overset{(3.50)}{=} \mathbb{P}[\![(..., \varnothing, ...)]\!\langle(..., \mathbb{S}[\![S_a^j\,;S_b^j]\!\langle\sigma^j\rangle, ...)\rangle\\
&\overset{(3.49)}{=} \mathbb{P}[\![\varnothing]\!\langle(..., \mathbb{S}[\![S_a^j\,;S_b^j]\!\langle\sigma^j\rangle, ...)\rangle
\end{aligned}$$

$$\begin{aligned}
\Rightarrow\ &\mathbb{P}[\![(..., S_a^j, ...)\,;...\,;com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;...\,;(..., S_b^j, ...)]\!\langle(\sigma^0, ..., \sigma^j, ..., \sigma^{P-1})\rangle\\
&= \mathbb{P}[\![(..., S_a^j\,;S_b^j, ...)\,;...\,;com(\,k\,;\,i\,;\,id_k\,;\,id_i\,)\,;...\,;(..., \varnothing, ...)]\!\langle(\sigma^0, ..., \sigma^j, ..., \sigma^{P-1})\rangle
\end{aligned}$$

□

Dans le cas particulier de notre succession de transformations, seule une tâche est exécutée lors d'une super-étape de calculs. Ainsi, en prenant le cas où $S_a^j = \varnothing$ et $S_b^j \neq \varnothing$ et où cette transformation peut être faite, alors $S_a^j ; S_b^j = S_b^j$ peut s'exécuter en même temps qu'une autre tâche.

4.4 Conclusion

Ce chapitre a montré que les différentes transformations, décrites dans le chapitre 3, sont correctes. Les preuves concernant les différentes phases de transformations : la préparation du code (section 4.1), l'optimisation du code (section 4.2) et la mise en parallèle du code (section 4.3) ont été réalisées. J'ai pu

prouvé que les différentes transformations successives génèrent bien un nouveau programme équivalent au précédent. Les différentes transformations effectuées et les relations d'équivalence qu'elles entraînent sont résumées dans le Tableau 4.1

Ce chapitre met en évidence la possibilité de générer un code parallèle pour des tâches sur une architecture distribuée au moyen d'une succession de transformations simples sans recourir à des *built-ins*.

Le chapitre 5 montre que ces transformations sont modulaires et peuvent être remplacées par d'autres transformations équivalentes et plus performantes. Aucune preuve n'est faite pour ces nouvelles transformations.

Transformations		**Relations d'équivalence**	
$\mathcal{T}_{\mathbb{S}_{ddd}}$	Déclaration Duplication	$\approx_0$ $\mathcal{R}_d$	Préservation du store sur Id_0 Extension du domaine des stores
$\mathcal{T}_{\mathbb{S}_{com}}$ $\mathcal{S}_{cp}$	Cohérence mémoire	$\approx_{1\to P}$	Équivalence sur les variables répliquées
$\mathcal{T}_{\mathbb{S}_{map}}$	Placement des expressions	$\approx_{1\rightsquigarrow p}$ $\approx_{new}$	Équivalence pour la substitution des variables d'origine Équivalence entre variables répliquées
$\mathcal{T}_{\mathbb{S}_{idc}}$ $\mathcal{T}_{\mathbb{S}_{eid}}$	Initial Declaration Clean Élimination des identités	$\approx$	Équivalence
$\mathcal{T}_{\mathbb{S}_{dce}}$	Dead Code Elimination	$\approx_{LIVE(s)}$	Équivalence sur l'ensemble des variables vivantes
$\mathcal{T}_{\mathbb{S}_{s2p}}$ *copy2com*	Sequential to Parallel	$\approx_{\hookrightarrow /\!/}$	Équivalence entre le domaine séquentiel et parallèle

TABLE 4.1 – Transformations et Relations d'équivalence associées

Chapitre 5

Optimisation du processus de compilation

Sommaire

Ce chapitre décrit des transformations supplémentaires qui peuvent être utilisées par notre processus de compilation. Contrairement au chapitre 3 qui décrit des transformations simples en vue de preuves de correction, présentées au chapitre 4, ce chapitre vise les performances du code parallèle que l'on souhaite obtenir. Il est donc complémentaire au chapitre 3 mais aucune preuve formelle de ces transformations n'est donnée.

La section 5.1 cible les problèmes qui peuvent être rencontrés lors de la détection et génération des tâches et de leur placement. La section 5.2 propose des transformations alternatives à celles du chapitre 3, pour la mise en cohérence mémoire, mais plus performantes. Les sections 5.3 et 5.4 proposent des optimisations supplémentaires à appliquer sur le code séquentiel avant le passage à un code parallèle. Elles permettent respectivement de réduire le nombre de communications à effectuer et de réduire l'espace mémoire utilisé. La dernière

section 5.5 discute de la phase de génération de code parallèle effectuée avec la bibliothèque MPI. Elle propose également des améliorations du passage de la syntaxe séquentielle à parallèle.

La plupart des sections de ce chapitre peuvent être lues indépendamment les unes des autres. Sauf indication contraire, toutes les améliorations et optimisations décrites dans ce chapitre ont été implémentées et sont utilisées dans le processus de compilation qui fait l'objet des expériences du chapitre 6.

5.1 Un graphe de dépendances de données étendu : *Effects Dependence Graph*

```
1  void example() {
2     int i;
3     int a[10], b[10];
4     for(i=0; i<10; i++) {
5        a[i] = i;
6        typedef int mytype;
7        mytype x;
8        x = i;
9        b[i] = x;
10    }
11    return;
12 }
```

Code 5.1 – boucle `for` en C99 avec déclaration de type et de variable interne à ma boucle

```
1  void example() {
2     int i;
3     int a[10], b[10];
4     for(i=0; i<10; i++)
5        a[i] = i;
6     for(i=0; i<10; i++)
7        typedef int mytype;
8     for(i=0; i<10; i++)
9        mytype x;
10    for(i=0; i<10; i++) {
11       x = i;
12       b[i] = x;
13    }
14    return;
15 }
```

Code 5.2 – Après une distribution de boucle incorrect du Code 5.1

Cette section résume très succinctement le contenu des articles [4] et [5], que j'ai présentés au workshop *Compilers for Parallel Computing (CPC)* et à la conférence *Source Code Analysis and Manipulation (SCAM)*. Je replace le résultat de ces travaux dans le contexte de ma thèse.

Le graphe de dépendances de données est indirectement utilisé lors de notre étape de maintien de la cohérence mémoire, réalisée par la Transformation 3.8, et plus directement par l'élimination de code mort implémentée dans PIPS. La définition classique du graphe de dépendances de données décrite dans le *Dragon Book* [9] ou dans d'autres livres de compilation [54][110] est suffisante pour l'application correct des deux transformations précédentes.

Par contre, pour la phase de détection et de création des tâches et de leur placement sur les différents processus, le graphe de dépendances de données classique n'est pas suffisant. En effet, il ne prend pas en compte les instructions de déclaration des variables, voire de types, si on considère toutes les possibilités offertes par le langage C99. Par exemple, dans le contexte de génération de code distribué, lors de la création des tâches, il peut être utile de distribuer des boucles [54] pour générer plusieurs tâches pouvant s'exécuter simultanément. Si des déclarations sont présentes, comme dans l'exemple du Code 5.1 (lignes 6

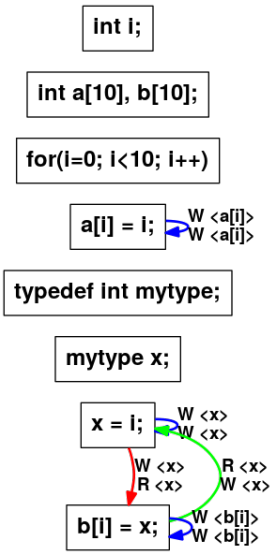


FIGURE 5.1 – Graphe de dépendance de données pour le Code 5.1

et 7), une distribution de boucles, se basant sur les informations du graphe de dépendances de données classique (Figure 5.1), donnerait le Code 5.2 qui est incorrect. Un compilateur source-à-source peut considérer les déclarations comme étant des instructions classiques et faire une distribution de celles-ci, comme le montre les lignes 7 et 9 du Code 5.2.

Dans PIPS, le calcul du graphe de dépendances de données est fondé sur l'analyse des effets. Cette analyse s'applique à chaque instruction et renvoie les variables qui sont lues et les variables qui sont écrites par l'instruction. Elle permet de réaliser une analyse de dépendances de données qui construit le graphe de dépendances de données. Les trois contraintes de dépendances de données sont ainsi calculées grâce aux effets :

Flow Dependence ou vrai dépendance, correspond à la lecture d'une variable effectuée après une écriture de celle-ci (RAW) ;

Anti-Dependence correspond à l'écriture d'une variable effectuée après une lecture de la même variable (WAR) ;

Output Dependence correspond à deux écritures successives sur une variable (WAW).

Les limitations de ce graphe de dépendances de données proviennent de trois nouvelles fonctionnalités qui ont été ajoutées dans la norme C99 :

1. Les déclarations de variables peuvent être effectuées n'importe où dans le code. Cette fonctionnalité provient de ce qui pouvait se faire en C++ à la même époque. Pour un compilateur source-à-source cela implique de conserver les informations nécessaires à la génération d'un nouveau code avec les déclarations au même endroit.
2. Les déclarations de tableaux de taille variable, *variable-length array*, sont possibles. Elles permettent la déclaration de tableaux de façon dynamique sans utiliser de `malloc` et de `free`.

3. Les déclarations de types peuvent également apparaître n'importe où dans le code avec des `struct`, `union`, `enum` ou `typedef`.

Une solution classique pour traiter les problèmes liés aux déclarations de variables et de types qui peuvent apparaître n'importe où, est de les regrouper au début de la fonction pour les variables, avec la transformation *flatten code*, ou avant la fonction les utilisant pour les types. Mais cette solution ne fonctionne pas en présence de tableaux à taille variable, car dans ce cas la taille du tableau serait initialisée après sa déclaration, ce qui conduirait à un code incorrect. De plus, dans le contexte de ma thèse, sortir les variables locales à une tâche serait contre-productif pour deux raisons. La première est que cela alourdirait inutilement le processus de génération du code parallèle décrit lors du chapitre 3. La deuxième est que cela pourrait entraîner une consommation excessive des ressources mémoire pour la majorité des processus qui exécutent le code parallèle généré, car les variables locales à une tâche sont censées devenir des variables privées à un processus, *ie.* déclarées uniquement sur ce processus.

La solution proposée dans ces articles est plutôt d'étendre l'analyse des effets des écritures et des lectures sur les variables. Au lieu d'uniquement considérer les effets qu'à une instruction sur l'état mémoire, on considère également ses effets sur son environnement, c'est-à-dire quand une variable est déclarée ou lorsqu'elle est référencée. Ce nouvel effet est appelé "effet d'environnement". Un "effet d'environnement" d'écriture correspond à la déclaration d'une nouvelle variable. Un "effet d'environnement" de lecture est associé à chaque écriture ou lecture d'une variable.

Une instruction ne modifie plus uniquement un état mémoire σ, mais un couple (environnement, état mémoire) (ρ, σ). La sémantique doit donc être modifiée pour pouvoir prendre en compte ce nouveau paramètre.

$$
\begin{aligned}
\rho \in \mathit{Env} &: \langle \mathit{identifier} \rangle \to \mathcal{L} \\
\sigma \in \mathit{Store} &: \mathcal{L} \to \mathit{Value} \\
\mathbb{L} &: \langle \mathit{identifier} \rangle \to \mathit{Env} \times \mathit{Store} \to \mathcal{L} \\
\mathbb{E} &: \langle \mathit{expression} \rangle \to \mathit{Env} \times \mathit{Store} \to \mathit{Value} \\
\mathbb{S} &: \mathit{Env} \times \mathit{Store} \to \mathit{Env} \times \mathit{Store}
\end{aligned}
$$

Cette nouvelle sémantique ne sera pas plus détaillée.

De façon similaire, des "effets de types" peuvent être introduits. Un "effet de type" d'écriture correspond à la déclaration d'un nouveau type utilisateur. Un "effet de type" de lecture correspond à l'utilisation d'un type utilisateur lors d'une déclaration de variable.

Une extension du graphe de dépendances de données traditionnel prenant en considération ces nouveaux effets peut donc être faite. Nous l'avons appelé graphe de dépendances des effets ou *Effects Dependence Graph*.

Définition 5.1 (Effects Dependence Graph)**.** Un graphe de dépendances des effets ou *Effects Dependence Graph* est une extension du graphe de dépendances de données qui tient compte de l'environnement et de la déclaration des types utilisateur.

Grâce à ce nouveau graphe de dépendances, la création de nouvelles tâches associées à une distribution de boucles est possible durant la phase préliminaire

de génération de code parallèle. Le résultat du Code 5.1 est présenté dans le Code 5.3 qui a été construit à l'aide du graphe de la Figure 5.2.

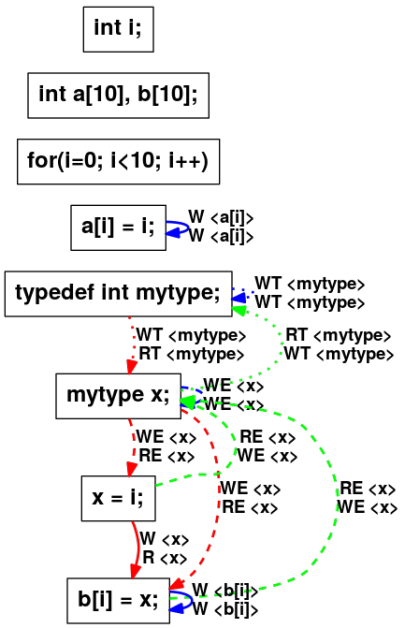


FIGURE 5.2 – Graphe de dépendance des effets pour le Code 5.1

```
void example() {                    1
   int i;                           2
   int a[10], b[10];                3
   for(i=0; i<10; i++)              4
      a[i] = i;                     5
   for(i=0; i<10; i++) {            6
      typedef int mytype;           7
      mytype x;                     8
      x = i;                        9
      b[i] = x;                     10
   }                                11
   return;                          12
}                                   13
```

Code 5.3 – Après une distribution de boucle correct du Code 5.1

De plus, lors de la création de tâche, il facilite la détection des variables qui peuvent être locales à une tâche. Cela permet à la fois d'alléger le processus de compilation permettant la génération du code parallèle, mais également d'utiliser moins d'espace mémoire lors de l'exécution du code parallèle.

5.2 L'amélioration du maintien de la cohérence mémoire

Le maintien de la cohérence mémoire est sans doute la transformation la plus importante de notre suite de transformations avec l'étape de traduction du code séquentiel en code parallèle. Il est décrit en section 3.3.2 et est rediscuté en section 5.2.1 en donnant les avantages et inconvénients de ces transformations. Deux axes d'amélioration sont envisagés. Le premier consiste à tenir compte de la continuation du code (section 5.2.2). Le deuxième permet de ne plus considérer les tableaux dans leur ensemble, mais de considérer des sous-ensembles utiles des éléments de tableaux (section 5.2.3).

5.2.1 Solutions précédentes

La section 3.3.2 a exposé deux transformations possibles de mise en cohérence mémoire.

La première est sur-conservative. Elle affecte à toutes les variables répliquées la valeur de leur variable d'origine associée. Elle est très simple à implémenter

et systématique dans son fonctionnement. Le code qui en résulte est correct. Mais il ne permet pas de mettre en évidence le parallélisme potentiel du programme car notre succession de transformations conduit dans ce cas à un code séquentiel exécuté sur plusieurs processus. Cela a été abordé en section 3.4.1.1. Cette approche sur-conservative n'est donc pas intéressante à moins d'envisager l'implémentation d'une nouvelle transformation spécifique, additionnelle à l'élimination de code mort traditionnelle.

La deuxième est également conservative, mais elle tient compte des tâches pour lesquelles le maintien de la cohérence doit être faite. Ainsi, seules les variables qui sont écrites au sein d'une tâche affecteront la valeur des variables associées correspondantes sur les différents processus. Ces variables écrites peuvent être détectées grâce à l'analyse des effets, introduite dans la section précédente. Elle reste simple à implémenter, tout en améliorant la précision des communications. Elle permet également d'éviter le problème soulevé par la première transformation. Néanmoins beaucoup d'instructions d'affectation générées restent inutiles.

5.2.2 Prise en compte de la continuation du code

Tenir compte de la tâche pour laquelle la mise en cohérence doit être effectuée permet de réduire le nombre d'instructions générées. L'étape suivante est de tenir compte de la continuation du programme. Deux solutions ont été envisagées. La première est d'utiliser le graphe de dépendances de données (section 5.2.2.1). La seconde est d'utiliser un sous-ensemble des variables faisant partie des effets d'écriture qui tient compte de la continuation du programme, et que l'on appelle effets exportés ou effets OUT (section 5.2.2.2).

5.2.2.1 Utilisation du graphe de dépendances de données

L'utilisation d'un graphe de dépendances de données permet de connaître les variables utiles dans la suite du programme, pour quelles tâches et donc pour quels processus. Ainsi, seule la valeur des variables d'origine réellement utiles au programme est transmise. De même, seuls les processus ayant besoin de la valeur de ces variables la recevront. Ainsi, on n'utilise plus la fonction de copie S_{cp} de la Définition 3.6, mais une analyse au cas par cas pour chaque variable d'origine sur chaque processus qui détermine si l'ajout d'une instruction de copie est nécessaire. L'utilisation d'un graphe de dépendances permet de s'approcher du nombre minimum de transferts nécessaires à un programme correct.

Néanmoins, cette implémentation est beaucoup plus complexe que les transformations précédentes. Non seulement, une analyse au cas par cas est nécessaire, mais une mémorisation des affectations déjà effectuées est essentielle pour éviter les redondances. Son exécution est également plus qu'exponentielle par rapport au nombre de variables, de tâches et de processus. L'utilisation par la suite d'une élimination de code mort devrait pouvoir générer un code similaire si ce n'est identique au code généré par cette transformation. Dans le cadre d'une succession de transformations "simples", le gain de cette transformation n'est pas significatif car l'élimination de code mort, qui est standard, peut potentiellement générer un code équivalent. Elle n'a donc pas été implémentée.

Pour information par rapport à la section 5.1, un graphe de dépendances de données traditionnel est suffisant pour cette transformation. Les informations sur les effets d'environnement ou de types ne modifient pas son fonctionnement car 1) pour l'environnement, soit les variables sont déclarées au niveau de la fonction, et comme aucune communication n'est faite après des déclarations, cela ne change rien, soit les variables sont déclarées localement à une tâche et aucun arc de dépendance n'exporte une variable locale ; 2) pour les déclarations de types, elles doivent toutes être connues et déclarées sur tous les processus.

5.2.2.2 Utilisation des effets exportés *OUT*

L'utilisation du graphe de dépendance complet est sans doute trop précise dans le cadre de transformations "simples". On peut se limiter à ne considérer que les variables ayant une dépendance de flot. Ainsi, toutes les variables étant écrites dans une tâche puis lues dans une tâche qui lui succède doivent être transférées. Ce transfert se fera vers toutes les variables associées à un processus indépendamment de la tâche qui doit l'utiliser. Mais, il est possible de se passer de ce graphe de dépendances comme expliquer ci-après.

Il suffit de considérer les variables vivantes, dont la définition a été donnée en Définition 4.3. Un algorithme de construction des variables vivantes est décrit par la suite. Si une variable est vivante après l'exécution d'une tâche et qu'elle a été écrite dans cette tâche, alors il est certain qu'à la fois cette tâche a affecté une valeur à cette variable, mais également que cette valeur est utile et doit donc être transmise à au moins une autre tâche. On définit ainsi un effet exporté OUT pour une instruction. Il a comme propriété d'être un sous-ensemble des effets d'écriture de l'instruction considérée et un sous-ensemble des variables vivantes après exécution de cette instruction.

Définition 5.2 (Effets exportés OUT)**.** Les effets exportés OUT correspondent à l'ensemble des variables définies par une instruction et dont leur valeur peut être utilisée dans la continuation du programme.

Propriété 5.1 (Effets exportés OUT)**.** *L'ensemble des variables faisant partie des effets exportés d'une instruction (E_{out}) correspond à l'intersection de l'ensemble des variables appartenant aux effets d'écriture de cette instruction (DEF) et des variables vivantes après exécution de cette instruction ($LIVE_{out}$).*

$$\forall s \in \langle block \rangle \cup \langle statements \rangle \cup \langle statement \rangle,$$
$$E_{out}(s) = DEF(s) \cap LIVE_{out}(s)$$

Les effets exportés diffèrent des variables vivantes car certaines variables vivantes peuvent provenir d'une écriture qui a été faite par une instruction antérieure à l'instruction courante. D'une certaine manière, on peut dire que les variables inclues dans les effets exportés correspondent aux variables qui viennent de devenir vivantes lors de l'exécution de l'instruction considérée.

La construction de l'analyse des variables vivantes est rappelée en Définition 5.3.

Définition 5.3 (Construction des variables vivantes)**.** L'analyse des variables vivantes est une analyse ascendante. Elle s'effectue au moyen des deux sous-ensembles $LIVE_{in}$ et $LIVE_{out}$, respectivement l'ensemble des variables vivantes avant et après exécution d'une instruction, comme suit :

$$\begin{aligned}
&\forall s \in \langle block \rangle \cup \langle statements \rangle \cup \langle statement \rangle, \\
&\qquad LIVE_{in}(s) = (LIVE_{out}(s) \setminus DEF(s)) \cup USED(s) \\
&\qquad LIVE_{out}(s) = \bigcup_{p \in succ(s)} LIVE_{in}(p) \\
&\qquad LIVE_{out}(f) = \varnothing
\end{aligned}$$

Les ensembles DEF et $USED$ correspondent respectivement aux variables définies, *ie.* écrites, et aux variables utilisées, *ie.* lues, par une instruction. $succ(s)$ correspond aux successeurs de l'instruction de s, c'est-à-dire les différentes instructions pouvant potentiellement s'exécuter après s. f est l'instruction finale de la fonction considérée.

Bien que non utilisés dans cette thèse, on peut également définir les effets importés *IN* correspondant à l'ensemble des variables lues par une instruction et qui ont été précédemment définies. Des propriétés entre les variables vivantes et les effets importés et exportés sont décrites en Propriété 5.2.

Propriété 5.2 (Effets importés/exportés et variables vivantes)**.** *Comme la Propriété 5.1, une relation peut être faite entre les variables vivantes en entrée d'une instruction et les effets importés (E_{in}).*

$$\begin{aligned}
&\forall s \in \langle block \rangle \cup \langle statements \rangle \cup \langle statement \rangle, \\
&\qquad E_{out}(s) = DEF(s) \cap LIVE_{out}(s) \\
&\qquad E_{in}(s) = USED(s) \cap LIVE_{in}(s)
\end{aligned}$$

On constate que les variables appartenant aux effets exportés correspondent aux variables qui doivent être ajoutées aux variables vivantes après exécution d'une instruction. Les variables appartenant aux effets importés correspondent tout ou en partie aux variables devant être retirées des variables vivantes. Les variables des effets importés ne correspondent pas forcément à toutes les variables à enlever des variables vivantes, car il n'est pas possible, de façon descendante, de prévoir si une variable pourra encore être lue ou non. Par contre, si une variable vivante n'est pas présente dans les effets importés alors elle reste vivante. La Propriété 5.3 permet d'exprimer ces constations.

Propriété 5.3 (Variables vivantes après exécution d'une instruction)**.** *Les variables vivantes après exécution d'une instruction correspondent aux variables vivantes avant exécution de l'instruction auxquelles certaines variables appartenant aux effets importés ont pu être retirées et auxquelles les variables des effets exportés on été ajoutées.*

$$\begin{aligned}
&\forall s \in \langle block \rangle \cup \langle statements \rangle \cup \langle statement \rangle, \\
&\qquad LIVE_{out}(s) = [LIVE_{in}(s) \setminus (E_{in}(s) \setminus \dots)] \cup E_{out}(s)
\end{aligned}$$

Au final, l'utilisation des effets exportés OUT au lieu des variables potentiellement écrites $\overline{WRITE}$ permet de réduire le nombre de variables dont on assure la cohérence mémoire à celles qui sont utiles dans la suite du programme. De ce fait, le nombre d'instructions de copies est réduit.

5.2.3 Amélioration concernant les tableaux

L'autre axe d'amélioration est de ne plus considérer un tableau comme étant une entité indivisible, mais de considérer les éléments qui le composent. Néanmoins, une analyse portant sur tous les éléments serait trop coûteuse en espace mémoire et en complexité. Un compromis a donc été fait : l'analyse polyédrique des références aux éléments de tableau. Elle permet de déterminer les parties de tableau qui sont utilisées aussi bien en lecture qu'en écriture. Cette analyse existe déjà dans PIPS et s'appelle l'analyse des régions convexes de tableaux.

5.2.3.1 Utilisation des régions convexes de tableaux

L'analyse des régions convexes de tableaux a été définie par Triolet [103] et Creusillet [28].

Cette analyse permet de calculer les parties convexes de tableaux, appelées régions convexes de tableaux, ou par abus de langage régions, qui sont lues ou écrites par une instruction. Une région est représentée par une abstraction qui est une fonction du *Store* vers un polyèdre des éléments du tableau référencés. Ces régions sont exactes lorsque l'analyse garantit que ses éléments sont lus ou écrits, ou approximées, lorsqu'ils sont potentiellement lus ou écrits. Il y a trois cas où une région est considérée comme approximée. Le premier vient de l'imprécision de l'abstraction des polyèdres convexes elle-même. Si par exemple une boucle travaille sur tous les éléments pairs d'un tableau, alors le polyèdre convexe qui en résulte englobe tous les éléments du tableaux de `0` à `2i`. Le deuxième cas d'imprécision provient des opérations de manipulation des polyèdres convexes. Si deux régions non adjacentes d'un unique tableau sont toutes deux en lecture ou en écriture, alors l'union convexe englobe les éléments entre ces deux régions. Le dernier cas d'imprécision vient des analyses statiques de code et des instructions de contrôle présentes dans le code. Lors d'un embranchement de test `if`, on ne peut pas toujours savoir quelle branche du test sera exécutée et si les deux branches du test ne référencent pas les mêmes régions de tableaux alors une union convexe sera effectuée.

Grâce aux analyses des régions convexes de tableaux, il est possible de réduire l'ensemble des éléments de tableaux devant être transmis entre les différentes tâches. Ainsi, une nouvelle transformation transférant uniquement les éléments de tableaux ou les scalaires ayant pu être écrits peut être conçue.

Les régions exportées OUT peuvent également être considérées afin de tenir compte de la continuité inter-procédurale du programme tout entier. Leur calcul n'utilise pas directement la notion de variables vivantes qui considèrent les tableaux comme des entités indivisibles. Mais une extension des variables vivantes en régions vivantes pourraient être envisagée. Les régions exportées actuelles sont obtenues au moyen de sous- et sur-approximations des régions écrites et lues. Elles sont présentées en détail dans la thèse de Creusillet [28].

L'utilisation des régions, aussi bien écrites qu'exportées, a néanmoins un défaut. Les bornes, qui sont utilisées pour décrire ces régions, peuvent être composées de valeurs numériques ou d'équations utilisant des variables scalaires. Mais, leur présence soulève un problème lors des transformations ultérieures, notamment pour le processus récepteur de ces variables, lors du passage de la syntaxe séquentielle à la syntaxe parallèle. Les Codes 5.4 et 5.5 illustrent le passage d'une instruction de copie d'un tableau, dont les bornes sont composées de variables, par un couple de communications faisant l'envoi et la réception. Lors de la parallélisation, en supposant que les variables des bornes sont bien traduites pour appartenir au processus de réception, il n'y a aucune garantie que le processus de réception possède les bonnes valeurs des bornes, `min_r` et `max_r` dans notre exemple. Pour résoudre ce problème, on peut systématiquement ajouter une communication des variables des bornes, ou raffiner ces bornes en utilisant des variables vivantes comme expliqué dans la section suivante.

```
1 for(int i=min_s; i<max_s; i++)
2   a_r[i] = a_s[i];
```

Code 5.4 – copie d'une région de tableau du processus `s` au processus `r`

```
1 // sur le processus s
2 for(int i=min_s; i<max_s; i++)
3   send(a_s[i], r);
4
5 // sur le processus r
6 for(int i=min_r; i<max_r; i++)
7   receive(a_r[i], s);
```

Code 5.5 – traduction de l'instruction de copie du Code 5.4 en communication

5.2.3.2 Utilisation des régions convexes de tableaux et des variables vivantes

Pour résoudre le problème des variables des bornes qui n'ont pas la bonne valeur sur le processus de réception, on propose d'exprimer les bornes des régions écrites ou exportées uniquement avec des variables vivantes. Dans la plupart des cas, c'est possible.

Mais ce n'est malheureusement pas toujours le cas, comme le montre les Codes 5.6 et 5.7.

En effet, la dernière utilisation d'une variable peut être réalisée lors de l'écriture d'un tableau. De ce fait, il est impossible d'exprimer la région écrite avec une autre variable. Le Code 5.6 en est un exemple. Un tableau est initialisé en deux fois, au moyen des boucles des lignes 6 et 11. Après l'instruction de boucle 11, la variable `n` n'est plus une variable vivante, or le tableau `a` a ses régions écrites et exportées entre `n` et `10` et elles ne peuvent pas s'exprimer autrement.

Même s'il est possible d'exprimer les régions écrites et exportées uniquement avec des variables vivantes, ce n'est pas suffisant pour que notre processus de compilation génère un code correct. Le Code 5.7 illustre ce problème. Trois tâches, représentées par les boucles aux lignes 7, 12 et 15, travaillent sur trois tableaux `a`, `b` et `c` de taille respective `m`, `n` et `p`. Ces trois tâches vont s'exécuter sur des processus différents. Les tableaux `a`, `b` et `c` ont en réalité la même taille.

```
1  int a[10];
2  n = ...    (0<n<10)
3
4  // <a[PHI1]-OUT-EXACT-
5  //  {0<=PHI1, PHI1<n}>
6  for i=[0:n]
7    a[i] = ...
8
9  // <a[PHI1]-OUT-EXACT-
10 //  {n<=PHI1, PHI1<10}>
11 for i=[n:10]
12   a[i] = ...
13
14 for i=[0:10]
15   ... = a[i]
```

Code 5.6 – une région contrainte uniquement avec des variables vivantes est impossible

```
1  m = ...
2  n = m
3  p = m
4
5  // <a[PHI1]-OUT-EXACT-
6  //  {0<=PHI1, PHI1<n, n=m=p}>
7  for i=[0:m]
8    a[i] = ...
9
10 // <b[PHI1]-OUT-EXACT-
11 //  {0<=PHI1, PHI1<p, n=m=p}>
12 for i=[0:n]
13   b[i] = ...
14
15 for i=[0:p]
16   c[i] = a[i] + b[i]
```

Code 5.7 – une région peut faire vivre une variable distante

L'analyse des régions de la première tâche, dont les bornes sont caractérisées par des variables vivantes, donne une région écrite pour `a` de `0` à `n`, de même pour sa région exportées. En effet, `n` est vivante à ce stade du code car la deuxième tâche va l'utiliser. Le problème est qu'après la transformation de placement des tâches sur les processus (Transformation 3.9 $\mathcal{T}_{\mathbb{S}_{map}}$) et l'élimination de code mort, le processus de la troisième tâche ne peut pas utiliser la valeur de `n` car elle a été éliminée. Lors du passage au domaine parallèle, un code erroné peut donc être généré. On peut toutefois noter que la première tâche qui n'avait pas besoin de `n`, en a maintenant besoin. Par l'ajout de l'utilisation de `n` dans la "tâche de copie", l'élimination de code mort ne supprimera pas l'envoi de sa valeur sur ce processus comme cela aurait pu être fait auparavant.

Nous avons montré sur des exemples que limiter l'expression des bornes présentes dans les régions aux variables vivantes n'est donc ni possible dans tous les cas, ni suffisant pour notre processus de compilation pour la génération de code parallèle. Néanmoins, le gain potentiel sur la réduction du volume des communications est non négligeable. Une nouvelle transformation, générant le code parallèle, vérifie si les variables utilisées pour les bornes des régions sont toujours vivantes sur le processus récepteur. Si elles ne le sont pas, alors elle ajoute une communication de cette variable avant d'effectuer la communication de la région de tableau. On rappelle que le processus émetteur possède forcément la valeur de cette variable étant donné qu'il l'utilise dans sa "tâche de copie", comme montré par l'exemple Code 5.7.

5.2.4 Conclusion

La transformation du maintien de la cohérence mémoire $\mathcal{T}_{\mathbb{S}_{com}}$ est une transformation importante de notre processus de compilation pour la génération correcte d'un code parallèle de tâches sur architecture distribuée. Deux améliorations ont été proposées pour réduire le nombre de variables devant être mises en cohérence et par conséquent réduire le nombre de communications potentielles.

La première consiste à ne pas mettre en cohérence une variable dont la valeur ne sera pas utilisée par la suite. Pour cela, on a utilisé la notion d'effet exporté qui correspond à l'ensemble des variables qui sont écrites dans une instruction et lues dans la continuation du programme. La deuxième consiste à réduire la mise en cohérence mémoire aux régions de tableaux le nécessitant. Cette réduction entraîne néanmoins une modification de la transformation $\mathcal{T}_{\mathbb{S}_{s2p}}$ qui permet de passer à la syntaxe parallèle. Elle vérifie que les variables des indices des bornes sur le processus récepteur sont vivantes et ajoute leur communication, si ce n'est pas le cas.

5.3 L'optimisation des bornes de boucles

La phase d'optimisation de notre processus de compilation (section 3.4) n'a décrit que l'élimination de code mort comme indispensable pour l'obtention d'un code parallèle. Mais d'autres optimisations sont applicables pour améliorer les performances du code généré.

La première optimisation consiste à réduire les itérations des boucles aux seules itérations nécessaires à l'exécution correcte du code. Cela correspond à minimiser les bornes de boucles, ou à éliminer les itérations de boucles inutiles. Le principe général est assez similaire à une élimination de code mort. Au lieu de supprimer des instructions inutiles, ce sont les itérations de boucles inutiles qui sont supprimées. Comme l'élimination de code mort, cette minimisation des bornes des boucles doit tenir compte de la continuation du programme. J'ai appelé cette optimisation **élimination d'itérations inutiles** ou *dead iteration eliminition*. Cette optimisation n'est pas propre à notre contexte de compilation et peut s'appliquer dans le cas général sur n'importe quel programme.

L'élimination d'itérations inutiles utilise la notion présentée dans la section précédente (section 5.2), à savoir les régions OUT. Elle est assez similaire à la transformation de maintien de la cohérence, mais au lieu de réduire les bornes des boucles de copie à ajouter, la réduction s'effectue sur des boucles déjà existantes. Les boucles concernées par cette optimisation portent sur des calculs de valeurs de tableaux. Les boucles contenant une réduction ou une variable d'induction utilisée en dehors de la boucle ne peuvent bénéficier de cette op-

```
1  for i=[0:10]
2    a_0[i] = ...
3  for i=[0:10]
4    a_1[i] = a_0[i]
5  for i=[0:10]
6    a_2[i] = a_0[i]
7
8  for i=[0:5]
9    ... = a_1[i]
10
11 for i=[5:10]
12   ... = a_2[i]
```

Code 5.8 – Avant la minimisation de bornes de boucle

```
1  for i=[0:10]
2    a_0[i] = ...
3  for i=[0:5]
4    a_1[i] = a_0[i]
5  for i=[5:10]
6    a_2[i] = a_0[i]
7
8  for i=[0:5]
9    ... = a_1[i]
10
11 for i=[5:10]
12   ... = a_2[i]
```

Code 5.9 – Après la minimisation de bornes de boucle

timisation. Ainsi, pour chaque boucle contenant des instructions calculant des tableaux, les régions OUT donnent les parties de tableaux qui sont modifiées par cette boucle et qui sont utiles dans la continuation du code. À partir de cette information, on réduit les bornes de la boucle aux seules itérations utiles.

Les Codes 5.8 et 5.9 donnent des exemples de résultats de la transformation de minimisation des bornes de boucles dérivant de notre processus de compilation. Une première tâche (l. 1 à 6) calcul la totalité des éléments du tableau `a`. La première moitié du tableau `a` est utilisée par la seconde tâche (l. 8 et 9) et la deuxième moitié par une troisième tâche (l. 11 et 12). Après placement des tâches sur les différents processus, il est possible de distinguer les parties de tableau utilisées par chaque processus. La minimisation de bornes de boucles permet de ne copier que la première moitié du tableau pour le processus `1` et la deuxième moitié pour le processus `2`.

5.4 L'optimisation des dimensions de la taille des tableaux en C

Le redimensionnement de la taille des tableaux, ou *array resizing*, est une autre optimisation possible. Elle permet de réduire l'espace mémoire dont un processus a besoin lors de son exécution. Comme la minimisation de bornes de boucles, ce redimensionnement n'est pas propre à notre contexte de compilation et peut s'utiliser dans le cas général. Un redimensionnement de la taille des tableaux en C diffère d'un redimensionnement pouvant être fait en Fortran [12]. En effet, en Fortran, il est permis de déclarer un tableau à partir de n'importe quel indice, alors qu'en C les tableaux doivent toujours commencer à l'indice `0`.

Le redimensionnement des tableaux en C s'effectue en trois étapes. La première étape détermine les régions de tableaux utilisées. La deuxième redimensionne les déclarations des tableaux. Enfin, un décalage approprié dans les fonctions d'accès aux éléments de tableaux doit être fait pour assurer la correction du programme résultant.

On note que ce redimensionnement de tableaux n'est pas conservateur au niveau du placement des données en mémoire. Par exemple, des données non contiguës en mémoire pourront le devenir.

Calcul des régions de tableau utilisées

Le calcul des régions de tableaux a été décrit en section 5.2.3.1. Les régions sont calculées pour chaque instruction du programme, or on souhaite avoir la région globale qui est utile pour chaque tableau pour tout le programme. La notion de région cumulée est ainsi introduite, permettant de résumer l'ensemble des régions utilisées par une suite d'instructions en une seule région. Pour cela, une union convexe de toutes les régions est faite pour obtenir cette région cumulée. La région utile pour un tableau correspond ainsi à la région cumulée de toutes les instructions qui suivent la déclaration de ce tableau.

Modification des déclarations de tableaux

La deuxième étape consiste à redimensionner les déclarations des tableaux. Contrairement au Fortran, les déclaration de tableaux en C doivent toujours

commencer à l'indice `0`. La différence entre l'indice maximal et minimal utilisé par le tableau détermine la nouvelle taille du tableau.

Si on ne souhaite pas trop modifier le placement et la contiguïté des éléments du tableau, seule la dernière dimension peut être considérée. En Fortran, c'est la première dimension qui doit être considérée.

Décalage dans la fonction d'accès aux éléments du tableau

Lors de la modification de la déclaration d'un tableau, un décalage est réalisé si l'indice minimal utilisé n'est pas `0`. Il faut ainsi reporter ce décalage pour l'ensemble des fonctions d'accès de ce tableau dans le programme. Le décalage réalisé correspond à un décalage vers la gauche des accès égal à l'indice minimal utilisé.

Les Codes 5.10 et 5.11 sont des exemples de résultats de la transformation de redimensionnement de la taille des tableaux. Le tableau `a` a une taille initiale `20`. Seuls ses éléments d'indice compris entre `5` et `14` sont référencés. Il est donc possible de réduire la taille du tableau à `10`. Comme les accès au nouveau tableau doivent commencer à `0`, un décalage des utilisations en écriture et en lecture doit être fait. Il correspond à l'indice minimal du tableau utilisé dans le code initial, soit `-5`.

```
1 int a[20];
2 int i;
3 for (i=5; i<15; i++)
4   a[i] = ...
5 for (i=5; i<15; i++)
6   ... = a[i]
```

Code 5.10 – Avant le redimensionnement de tableaux

```
1 int a[10];
2 int i;
3 for (i=5; i<15; i++)
4   a[i-5] = ...
5 for (i=5; i<15; i++)
6   ... = a[i-5]
```

Code 5.11 – Après le redimensionnement de tableaux

Dans notre contexte, pour complètement profiter de la réduction de l'espace mémoire utilisé, il faudrait également que chaque variable ne soit déclarée que par le processus qui l'utilise. Cette passe d'optimisation supplémentaire ne peut pas être réalisée sur le code séquentiel car cela introduit des problèmes lors de la génération du code parallèle, notamment pour les instructions de réception des communications. Elle peut être réalisée sur le code parallèle que l'on génère. On considère qu'une instruction com représente une instruction d'envoi à partir du processus émetteur et une instruction de réception sur le processus récepteur. La transformation consiste à regrouper les instructions s'exécutant sur un processus dans une fonction particulière et ce pour chaque processus. Dans cette fonction, les déclarations des variables des autres processus peuvent être supprimées. Ainsi, l'espace mémoire utilisé par chaque processus est minimal. Le code initial est transformé pour appeler chacune des fonctions locales à chaque processus.

Contrairement au redimensionnement de boucle, cette optimisation supplémentaire portant sur le code parallèle n'a malheureusement pas été implémentée faute de temps.

5.5 La génération de code MPI

La génération du code parallèle est une étape importante de notre processus de compilation. En effet, c'est elle qui permet l'obtention concrète d'un code parallèle distribué.

Le passage d'un programme séquentiel en un programme parallèle en section 3.5 présente deux étapes de transformation. En pratique, ces deux étapes sont souvent regroupées en une seule transformation. C'est le cas pour générer du code utilisant la bibliothèque MPI. Nous avons représenté formellement le code séquentiel en utilisant le modèle BSP (section 3.5.1). Mais avec la bibliothèque MPI, de même qu'avec la plupart des langages parallèles, des modèles plus complexes sont utilisés. De ce fait, l'étape de parallélisation est implicite et est liée au langage MPI. Le mode de communication utilisé est synchrone.

5.5.1 Génération de communications pour les tableaux et les structures

La description faite dans la syntaxe de $\langle com \rangle$ est représentée en MPI par une instruction `MPI_Send` sur le processus émetteur et `MPI_Recv` sur le processus récepteur. En plus, de la transformation décrite en section 3.5.3 permettant le passage de la syntaxe séquentielle à la syntaxe parallèle, une initialisation de l'environnement MPI doit être réalisée, de même qu'une clôture de cette environnement.

La transformation décrite en section 3.5.3 est assez rudimentaire et des améliorations peuvent y être apportées. Particulièrement, la communication de tableau est facilement améliorable. La transformation initiale réalise une communication élément par élément des tableaux. Or, ce mode de communication est inefficace et très coûteux. Ainsi, il est préférable de communiquer directement l'ensemble du tableau en une seule instruction. Pour cela, une légère modification de la syntaxe parallèle est opérée sur $\langle com \rangle$ pour ajouter le nombre d'éléments devant être communiqué. Les modifications de la syntaxe et de la sémantique sont décrites ci-dessous.

$$\langle com \rangle \quad ::= \quad \mathrm{com}(\, s\,;\, r;\, \langle variable \rangle;\, \langle variable \rangle;\, n) \qquad 0 \leq s < P, 0 \leq r < P, n \in \mathbb{N}$$

$$\begin{aligned}
\mathbb{P}[\![\mathrm{com}(s\,;r;\,var_s\,;var_r\,;n)]\!]\langle {}^{P}_{0}\,\sigma\,\rangle = (&\sigma^0,\\
&\vdots\\
&\sigma^{r-1},\\
&update(\\
&\quad update(\\
&\quad ...\\
&\qquad update(\sigma^r, \mathbb{L}[\![var_r]\!]\langle\sigma^r\rangle, \mathbb{E}[\![var_s]\!]\langle\sigma^s\rangle),\\
&\quad ...),\\
&\mathbb{L}[\![var_r + n - 1]\!]\langle\sigma^r\rangle, \mathbb{E}[\![var_s + n - 1]\!]\langle\sigma^s\rangle),\\
&\sigma^{r+1},\\
&\vdots\\
&\sigma^{P-1})
\end{aligned}$$

$$\mathbb{L}[\![var + n]\!]\langle\sigma\rangle = (var, m + n) \quad \text{avec} \quad (var, m) = \mathbb{L}[\![var]\!]\langle\sigma\rangle$$

$$\mathbb{E}[\![var + n]\!]\langle\sigma\rangle = \sigma(\mathbb{L}[\![var + n]\!]\langle\sigma\rangle)$$

La sémantique de com modifie simultanément plusieurs valeurs du tableau communiqué. Le processus r reçoit pour sa variable var_r des valeurs de la variable var_s du processus s. Le nombre de valeurs ainsi échangé correspond au nombre n, et ce sont des valeurs adjacentes en mémoire à partir de var_s qui sont envoyées et qui sont reçues à partir de var_r. Cela se traduit par l'imbrication de plusieurs appels à la fonction *update* pour l'ensemble des éléments du tableau.

La fonction *copy2com* générant les communications est modifiée comme ci-dessous :

$$copy2com(id_k \leftarrow id_p) = \mathrm{com}(\, p \,;\, k \,;\, id_p \,;\, id_k \,;\, 1\,)$$

$$copy2com(id_k.elem \leftarrow id_p.elem) = \mathrm{com}(p \,;\, k \,;\, id_p.elem \,;\, id_k.elem \,;\, 1)$$

$$copy2com\begin{pmatrix} \mathrm{for}(i_m \leftarrow start; \\ \quad i_m < end; \\ \quad i_m \leftarrow i_m + 1) \\ id_k[i_m] \leftarrow id_p[i_m] \end{pmatrix} = \mathrm{com}(p \,;\, k \,;\, id_p[start] \,;\, id_k[start] \,;\, end - start)$$

La copie d'un scalaire se traduit par la communication d'un élément. La copie d'une partie d'un tableau représentée par un nid de boucles se traduit par la communication des éléments du tableau en une seule communication lorsque c'est possible. Cette communication commence par le premier élément référencé par la boucle, et envoie la région de tableau à communiquer correspondant au nombre d'itérations de la boucle.

Cette amélioration permet de réduire grandement le nombre de communications effectuées par le programme parallèle. Or les communications sont souvent très coûteuses dans un programme parallèle. Une réduction de leur nombre augmente substantiellement les performances.

Une amélioration similaire peut être réalisée pour les structures. MPI permet de définir des types personnalisés qui servent à transférer des structures. La définition du type de la structure doit dans un premier temps être ajouté. La syntaxe de $\langle com \rangle$ doit considérer le type à envoyer : un entier, un flottant ou une structure personnalisée.

Ces deux optimisations sur la communication des tableaux et des structures peuvent être appliquées directement au cours de l'étape de transformation du code séquentiel en code parallèle, ou par une transformation d'optimisation de code parallèle MPI indépendante. Cette dernière serait chargée de détecter que plusieurs éléments contigus de tableau sont communiqués et de les regrouper en une seule communication. Pour les structures, elle devrait également ajouter une description du type de la structure en MPI avant de pouvoir l'utiliser dans les communications.

Le générateur de code MPI que j'ai implémenté dans PIPS permet de regrouper les communications pour des éléments de tableaux contigus, mais pas pour ceux qui ne le sont pas, ni pour les structures.

5.5.2 Génération de communications asynchrones

Une autre amélioration potentielle est l'utilisation de communications asynchrones. Elle est plus complexe à implémenter car les communications asynchrones nécessitent l'ajout de notifications assurant que la communication a bien été effectuée. Une variable *de notification* doit être déclarée et une vérifi-

cation de cette dernière doit être faite avant chaque premier accès aux variables communiquées.

La détermination du nombre optimal de variables *de notification* devant être ajoutées est un premier problème. Ce nombre est au pire borné par le nombre de communications effectuées. Le nombre de variables *de calculs* présentes dans le code et la possibilité de réutiliser les variables de notification doit permettre de réduire la quantité des variables *de notification* à ajouter. Ce problème est complexe, notamment si les contraintes mémoire sont fortes.

Le placement de la vérification de cette variable de notification est également très critique. Il faut la mettre au plus près de l'utilisation de la variable qui a été communiquée. Mais si cette variable est utilisée dans une boucle, il peut être plus judicieux de la mettre en dehors de la boucle pour éviter de devoir répéter le test plusieurs fois. Mais la variable communiquée peut être utilisée uniquement à la fin de la boucle, le test peut empêcher le commencement des calculs. Une transformation sortant la première itération de boucle peut donc être effectuée pour éviter ce problème.

De plus, les utilisateurs expérimentés dans les calculs haute performance savent que les communications asynchrones ne sont pas toujours plus performantes que les communications synchrones. Un exemple simple et naïf est une communication asynchrone suivie directement d'une utilisation de cette variable et donc d'une vérification de la notification, est moins efficace qu'une communication synchrone. Concernant la bibliothèque MPI, plusieurs implémentations de cette bibliothèque existent. Elles n'ont pas toutes les mêmes performances par rapport aux communications synchrones et asynchrones. Une génération ou un remplacement systématique par des communications asynchrones n'est donc pas toujours la meilleure solution en terme de performance.

Notre implémentation n'utilise pas de communications asynchrones car les résultats sur les performances dépendent fortement du code considéré, de la machine l'exécutant et de l'implémentation de MPI utilisée.

5.6 Conclusion

Ce chapitre a décrit les différentes améliorations et optimisations qui ont été réalisées ou envisagées pour la génération automatique de code parallèle distribué de tâches.

Une optimisation de la détection et de la génération des tâches à distribuer a été discutée en section 5.1. Elle permet, dans le contexte d'une compilation source-à-source, de tenir compte des instructions de déclaration des variables qui peuvent apparaître n'importe où dans un programme C et de générer un code correct et facilement compréhensible par l'utilisateur.

La transformation de conservation de la cohérence mémoire décrite en section 5.2 permet de réduire le nombre de copies qui sont générées. Toutes les copies éliminées sont des communications en moins dans le code parallèle distribué.

Deux nouvelles optimisations ont également été proposées en sections 5.3 et 5.4. Elles peuvent être utilisées dans un contexte général pour l'optimisation de code source. La première optimisation permet d'éliminer des itérations de boucles inutiles. La deuxième réduit l'espace mémoire utilisé par un programme et notamment par les tableaux qu'il référence. Ces nouvelles optimisations sont

ajoutées à notre processus de compilation. À partir de la Figure 2.3 qui décrit les différentes possibilités de génération de code distribué, la Figure 5.3 précise le processus de compilation et d'optimisation que j'ai réalisé et qui fait l'objet des expériences présentées au chapitre 6.

Enfin, la génération du code parallèle utilisant la bibliothèque MPI a été discutée en section 5.5. Elle propose une amélioration de la transformation passant du code séquentiel au code parallèle en se penchant sur la communication des tableaux et des structures. Au lieu de les communiquer élément par élément, une communication unique de leur ensemble est introduite. Une présentation des communications asynchrones a aussi été réalisée.

Le chapitre 6 présente les expériences et résultats obtenus par le processus de compilation décrit au chapitre 3, et prouvé au chapitre 4, avec les optimisations présentées dans ce chapitre.

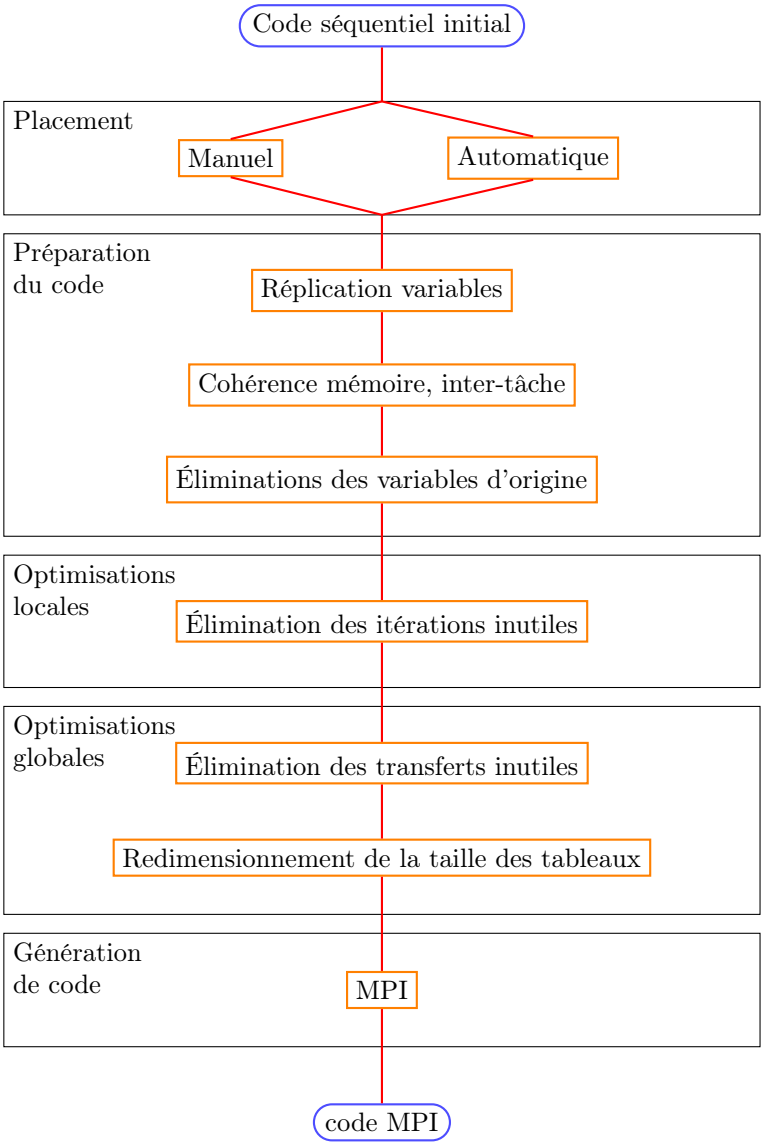


FIGURE 5.3 – Schéma de compilation permettant l'obtention d'une code parallèle distribuée pour des tâches

Chapitre 6

Expériences et interprétations

Sommaire

Ce chapitre a pour but de mesurer les temps d'exécution obtenus par le code parallèle distribué généré automatiquement par le processus de compilation décrit au chapitre 3 et optimisé au chapitre 5. Ces expériences ont été réalisées sur deux machines parallèles décrites en section 6.1. La première de ces machines est à mémoire partagée, tandis que la seconde est à mémoire distribuée. Notons que sur la machine distribuée, un seul *thread* est lancé sur chaque nœud pour réellement mesurer les temps des communications engendrées par l'exécution du code généré. Le *benchmark* Polybench, décrit en section 6.2, est utilisé comme plateforme pour réaliser nos expériences. Elles sont donc faites sur des problèmes d'algèbre linéaire (section 6.4), sur de l'exploration de données, *data-mining* (section 6.5) et sur du traitement d'image (section 6.6). Les détails des temps d'exécution et d'accélération sont présentés en Annexe B.

6.1 La configuration matérielle

Les expériences ont été réalisées sur deux machines différentes, appelées Pau et Katrina. Les caractéristiques de ces deux machines sont résumées dans le Tableau 6.1.

La première machine, Pau, est une machine à mémoire partagée. Elle est composée de deux processeurs "Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz" de huit cœurs qui peuvent être *hyper-threadés*. Jusqu'à 32 *threads* peuvent donc

	Pau	Katrina
Type de machine	mémoire partagée	mémoire distribuée
Processeurs	Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz	Intel(R) Xeon(R) CPU X5450 @ 3.00GHz
Cache L1	32kio	32kio
Cache L2	256kio	6Mio
Cache L3	20Mio	*NA*
Nombres de cœurs	2×8 cœurs	15×8 cœurs + 1
Système d'exploitation	Linux 4.4.0-57	Linux 2.6.32
Compilateur	gcc 5.4.0	gcc 4.4.7
version MPI	Open MPI 1.10.2	Open MPI 1.6.2

TABLE 6.1 – Caractéristiques matérielle et logicielle

s'exécuter sur cette machine. Ce processeur est sorti début 2012. Il possède un cache L1 de 32kio, un cache L2 de 256kio et un cache L3 de 20Mio. Pour mes expériences, je me limite à l'utilisation de 8 *threads*. La raison principale est que je souhaite mesurer les performances obtenues au sein d'un processeur sans échange possible avec l'autre processeur. L'utilisation de l'*hyper-threading* peut également entraîner des changements de contexte lors de l'exécution du programme que je souhaite minimiser. Ainsi, 8 processus s'exécuteront simultanément au maximum. Le système d'exploitation utilisé est Linux 4.4.0-57 64 bits. Cette machine compile nos expériences avec le compilateur gcc 5.4.0 et l'option d'optimisation '-O3'. L'implémentation MPI utilisée est Open MPI 1.10.2 [81].

La deuxième machine, Katrina, est une machine à mémoire distribuée. Katrina est un serveur mis à disposition en interne à l'école des Mines. Elle est composée de quinze nœuds "Intel(R) Xeon(R) CPU X5450 @ 3.00GHz" et d'un nœud d'entrée. Chaque nœud possède huit cœurs. Ce processeur est sorti fin 2007. Il possède un cache L1 de 32kio et un cache L2 de 6Mio. Il n'y a pas de gestion d'allocation du temps d'utilisation pour chacun des utilisateurs. Il revient donc à chaque utilisateur de vérifier la disponibilité des nœuds que l'on souhaite utiliser. Tous les nœuds n'étant pas toujours disponibles, nos expériences se limitent à utiliser simultanément encore une fois seulement huit nœuds au plus. De plus, étant donné que nous voulons tenir compte du temps de communication effectuée entre les tâches sur des nœuds différents, seul un *thread* est exécuté par nœud. Le système d'exploitation utilisé est Linux 2.6.32 64-bits. Cette machine compile nos expériences avec le compilateur gcc 4.4.7 et l'option d'optimisation '-O3'. L'implémentation MPI utilisée est Open MPI 1.6.2.

6.2 Polybench

Polybench [90] est un *benchmark* proposant, dans sa version 4.2, 30 programmes de calcul numérique à contrôle statique. Il permet, entre autre, de mesurer le temps d'exécution de ces programmes. Les applications de Polybench ont toutes une structure similaire à celle du Code 6.1.

```
1  int main(int argc, char** argv)
2  {
3    /* Retrieve problem size. */
4    int n = N;
5    /* Variable declaration/allocation. */
6    POLYBENCH_2D_ARRAY_DECL(C,DATA_TYPE,N,N,n,n);
7
8    /* Initialize array(s). */
9    init_array (n, POLYBENCH_ARRAY(C));
10
11   /* Start timer. */
12   polybench_start_instruments;
13
14   /* Run kernel. */
15   kernel_template (n, POLYBENCH_ARRAY(C));
16
17   /* Stop and print timer. */
18   polybench_stop_instruments;
19   polybench_print_instruments;
20
21   /* Prevent dead-code elimination on the kernel. */
22   polybench_prevent_dce(print_array(n, POLYBENCH_ARRAY(C)));
23
24   return 0;
25 }
```

Code 6.1 – Template d'une application Polybench

Un code Polybench commence par la déclaration des différentes tailles du problème et la déclaration des variables à traiter (lignes 3 à 6). Une initialisation des variables est réalisée (ligne 9). Puis, un chronomètre est lancé (ligne 12) pour pouvoir mesurer le temps d'exécution des calculs réalisés (ligne 15). Une fois cette exécution terminée, le chronomètre est arrêté et le temps d'exécution est donné (lignes 18 et 19). Le résultat du calcul est rendu, ce qui permet de vérifier qu'il est correct et d'éviter toute élimination de code mort (ligne 22).

Polybench ne mesure ainsi que le temps de calcul d'un programme et ignore ses temps de lancement, d'initialisation ou de chargement des données et d'enregistrement des données à la fin du programme.

Seule une partie des applications présentes dans Polybench est utilisée pour réaliser nos expériences, à savoir des problèmes d'algèbre linéaire (7 applications), présentés en section 6.4, et des algorithmes d'exploration de données (2 applications), présentés en section 6.5. De plus, une application de traitement d'image est codée, présentés en section 6.6, pour utiliser les fonctionnalités de Polybench.

6.3 Méthodologie d'expérimentation

Pour s'assurer de la cohérence des temps mesurés, toutes nos applications sont exécutées 5 fois. Les exécutions la plus rapide et la plus lente ne sont pas prises en compte et une moyenne des trois temps d'exécution restants est faite. C'est cette moyenne qui est présentée dans nos résultats.

Les tailles de l'ensemble des données par défaut proposées par Polybench ne sont pas utilisées. Ces tailles sont trop petites pour obtenir des temps d'exécution suffisants et pour que la distribution réalisée soit utile. Deux cas de différentes tailles de l'ensemble des données sont définis. Le cas "petit" représente une taille pouvant rentrer dans le plus grand cache présent dans chacune des deux machines d'expérimentations, à savoir 6Mio, mais ne rentrant pas dans le second plus grand cache. Ainsi, l'espace à allouer pour les expériences de cas "petit" est compris entre 256kio et 6Mio. À noter que le code généré par notre processus de compilation peut également tenir dans ce plus grand cache. Le cas "grand" représente une taille de données ne pouvant tenir dans aucun cache. L'espace à allouer par ces expériences est donc supérieur à 20Mio, et très souvent largement supérieure. Ces tailles sont définies en considérant que les calculs sont effectués en double précision, ce qui est le cas dans toutes les expérimentations réalisées.

Une exécution avec OpenMP est également réalisée sur une partie de nos expériences pour la machine à mémoire partagée Pau. Cette utilisation d'OpenMP est basique, avec une simple utilisation de la directive `#pragma omp parallel for`. Elle permet une comparaison de nos résultats par rapport à une utilisation simple d'OpenMP.

Des tentatives de génération d'un code distribué avec l'outil Pluto+ [21] et son extension [20] ont été réalisées. Mais elles n'ont pas été concluantes, car il y a eu, soit une erreur lors de la génération, soit le code généré est faux et ne peut pas être compilé. Le code généré par Pluto+ définit des fonctions permettant (1) de calculer les éléments de tableaux devant être communiqués aux différents processus et (2) d'emballer et (3) de déballer les tableaux à communiquer. Ces deux dernières fonctions générées présentent des erreurs de déclarations difficiles à corriger sans connaissance du contexte dans lequel elles doivent être générées. Cette tentative de génération d'un code distribué au moyen de Pluto+ s'est donc conclue par un échec. Aucun code pour machine distribuée, autre que le code qui est généré par notre processus de compilation, n'est donc présenté.

6.4 De l'algèbre linéaire

Les applications d'algèbre linéaire étudiées font partie de l'ensemble des fonctions présentes dans BLAS [64][32][17]. Plus précisément, elles font partie de l'ensemble des fonctions de niveau 3 [30] de BLAS. Elles correspondent à des opérations de type matrice-matrice et sont les plus complexes de BLAS. Elles sont au nombre de neuf dans l'implémentation officielle de BLAS version 3.7.0 [18]. Une partie de ces fonctions est reprise dans Polybench. Sept fonctions de BLAS sont présentes dans Polybench. Certaines d'entre elles sont décrites dans l'article [30], mais ne sont pas présentes dans l'implémentation officielle.

Les détails des temps d'exécution et d'accélération de ces expériences sont présentés en Annexe B.1. Nous décrivons ci-dessous chaque cas de test.

gemm *Generalized Matrix-Matrix Multiply* est une multiplication de matrices denses, standard de BLAS. Elle prend en entrée deux scalaires et trois tableaux :

— α, β, scalaires ;
— `A`, tableau de taille `M`×`K` ;
— `B`, tableau de taille `K`×`N` ;
— `C`, tableau de taille `M`×`N`.

Elle modifie en place le tableau `C` pour donner $\texttt{C} = \alpha\texttt{AB} + \beta\texttt{C}$. L'implémentation consiste en une boucle calculant, dans un premier temps, $\beta\texttt{C}$, puis $\alpha\texttt{AB} + \beta\texttt{C}$.

Une distribution du calcul sur la première dimension du tableau `C` est réalisée pour générer les différentes tâches. Une autre distribution est faite par bloc par rapport aux deux dimensions du tableau `C`.

Les figures 6.1 et 6.2 présentent les accélérations obtenues par rapport au nombre de processus lancés.

Sur Katrina, l'accélération est linéaire sur les petits et grands cas. La distribution par bloc est moins performante qu'une distribution simple par colonne.

Sur Pau, au delà de 4 processus utilisés, peu ou pas d'accélération est visible.

Pour le cas "petit", la version OpenMP est moins performante que notre code généré. Par contre, pour le cas "grand", elle passe à l'échelle, alors que la notre sature à partir de 4 processus.

La différence entre les résultats sur Pau avec la version OpenMP et la version générée par notre processus de compilation est due au cache. Nous avons utilisé PAPI [86], *Performance Application Programming Interface*, pour mesurer les défauts de caches, *cache miss*. Polybench permet en effet d'utiliser cette API pour les mesurer réellement. Le nombre de défauts de caches donnés par PAPI est présenté dans les figures 6.3 et 6.4 pour les versions séquentielles, OpenMP avec huit processus et notre code généré avec huit processus. Les trois niveaux de caches, L1, L2, et L3, sont distingués.

Pour le cas tenant dans le cache ("petit"), la version séquentielle ne présente pas de défaut de cache L3. Les huit processus de la version OpenMP ont des défauts de cache L3 d'environ 5000 chacun, alors que notre version présente des défauts, soit de l'ordre de la dizaine, soit autour de 1500. Les défauts de cache L3 d'OpenMP proviennent d'invalidations de lignes de cache induites lors des écritures de `C` par les différents processus, alors que, dans notre version, chaque processus a une copie différente de `C`. Il n'y a donc pas d'invalidations de ligne de cache provoquées par les différents processus, excepté lors des copies des tableaux entre les différents processus. Cela explique pourquoi notre version présente de meilleures performances que la version OpenMP dans le cas "petit".

Pour le cas ne tenant pas dans le cache ("grand"), le cache L3 doit, dans tous les cas, se recharger régulièrement pour pouvoir traiter les données. Or, dans notre version, l'espace requis est proportionnel au nombre de processus, chaque processus ayant sa propre version des données. De ce fait, le rechargement du cache est plus fréquent et notre version sature à partir de quatre processus.

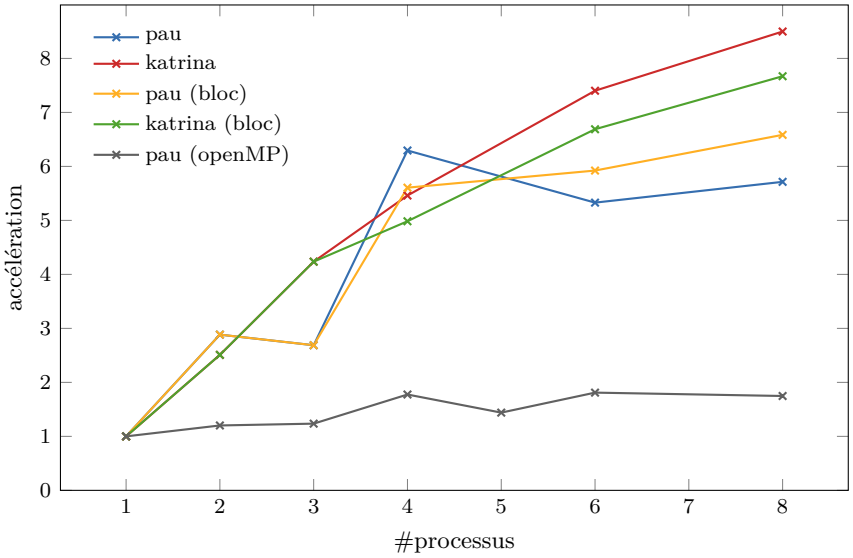


FIGURE 6.1 – Accélération de gemm (“petit”)

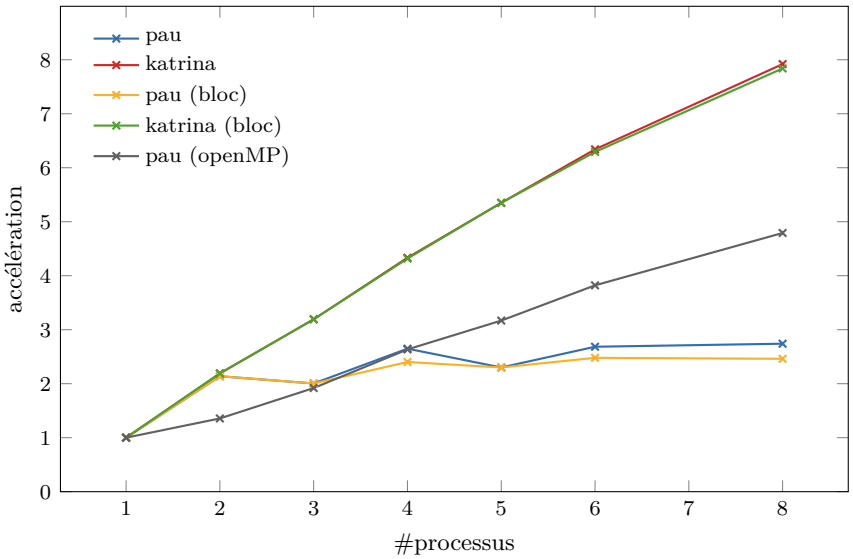


FIGURE 6.2 – Accélération de gemm (“grand”)

PAPI_L1_TCM=5788908	PAPI_L1_TCM=768287	PAPI_L1_TCM=727896
PAPI_L2_TCM=1543306	PAPI_L2_TCM=253339	PAPI_L2_TCM=279341
PAPI_L3_TCM=0	PAPI_L3_TCM=5205	PAPI_L3_TCM=9 -- 1623
version séquentielle	version OpenMP (8 proc.)	notre version (8 proc.)

FIGURE 6.3 – PAPI : défaut de caches pour gemm sur Pau ("petit")

PAPI_L1_TCM=11574253214	PAPI_L1_TCM=1451388212	PAPI_L1_TCM=1447354868
PAPI_L2_TCM=4296742222	PAPI_L2_TCM=503351387	PAPI_L2_TCM=563064254
PAPI_L3_TCM=1587447497	PAPI_L3_TCM=45404635	PAPI_L3_TCM=212562559
version séquentielle	version OpenMP (8 proc.)	notre version (8 proc.)

FIGURE 6.4 – PAPI : défaut de caches pour gemm sur Pau ("grand")

gemver `gemver` est un ensemble de multiplications de matrice-vecteur. Cette fonction fait partie d'une version étendue de BLAS [17] et n'est pas présente dans sa distribution courante. Elle prend en entrée deux scalaires, un tableau et six vecteurs :

- α, β, scalaires ;
- `A`, tableau de taille `N`×`N` ;
- `u1`, `u2`, `v1`, `v2`, `y`, `z`, vecteurs de taille `N`.

Elle retourne un nouveau tableau $\hat{\texttt{A}}$ et deux nouveaux vecteurs `x` et `w` de taille `N` tels que :

- $\hat{\texttt{A}} = \texttt{A} + \texttt{u1.v1} + \texttt{u2.v2}$;
- $\texttt{x} = \beta \hat{\texttt{A}}^T \texttt{y} + \texttt{z}$;
- $\texttt{w} = \alpha \hat{\texttt{A}} \texttt{x}$.

L'implémentation consiste en une succession de quatre tâches qui calculent respectivement $\hat{\texttt{A}} = \texttt{A} + \texttt{u1.v1} + \texttt{u2.v2}$, $\beta \hat{\texttt{A}}^T \texttt{y}$, $\texttt{x} = \beta \hat{\texttt{A}}^T \texttt{y} + \texttt{z}$ et $\texttt{w} = \alpha \hat{\texttt{A}} \texttt{x}$.

Ces quatre tâches dépendent les unes des autres et ne peuvent pas correspondre aux tâches affectées aux différents processus. Elles sont donc partitionnées en de nouvelles tâches pour correspondre au nombre de processus à exécuter. Ces partitions sont faites selon une dimension de `A`.

La Figure 6.5 représente l'accélération obtenue par rapport au nombre de processus lancés pour le cas "grand".

`gemver` ne permet pas d'obtenir de bonnes performances. Aucune accélération pour notre code généré n'est constatée ; on observe même une réduction de la vitesse d'exécution du programme. Une légère accélération est obtenue avec la version OpenMP. Cela peut s'expliquer pour deux raisons. La première est que la quantité de calculs n'est pas suffisante pour compenser les coûts des communications. En effet, les calculs s'exécutent relativement rapidement, en moins d'une seconde pour les valeurs de paramètre choisies. La deuxième raison est que le calcul de `x` dépend de la transposée de `A` et que le calcul de `w` dépend de `x` et de `A`. Donc, non seulement le calcul est forcément séquentiel entre ces différents calculs, mais la matrice `A` doit, dans tous les cas, être communiquée à plusieurs processus, voire à tous les processus, après son évaluation.

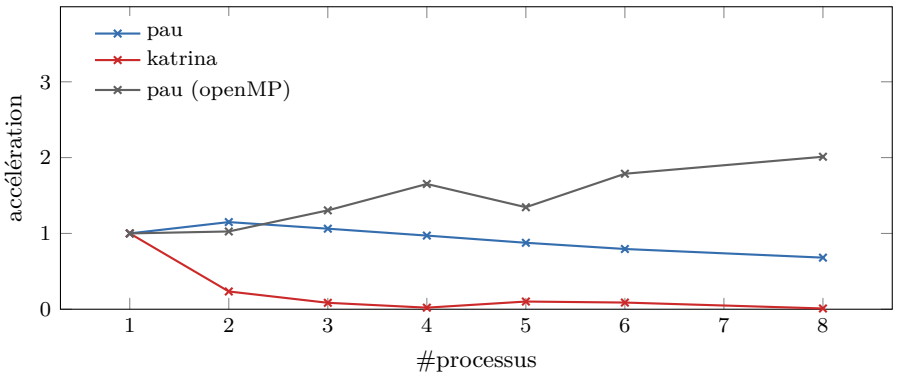


FIGURE 6.5 – Accélération de gemver (“grand”)

gesummv *Generalized Summed Matrix-Vector Multiply* est une somme de deux multiplications de matrice-vecteur. Comme `gemver`, elle fait partie d'une version étendue de BLAS [17] et n'est pas présente dans sa distribution courante. Elle prend en entrée deux scalaires, deux tableaux et un vecteur :

— α, β, scalaires ;
— `A`,`B`, tableaux de taille `N`×`N` ;
— `x`, vecteur de taille `N`.

Elle retourne un vecteur `y` de taille `N` tel que $\texttt{y} = \alpha\texttt{Ax} + \beta\texttt{Bx}$. L'implémentation consiste en une boucle calculant, dans un premier temps, `Ax` et `Bx`, puis $\alpha\texttt{Ax} + \beta\texttt{Bx}$.

Une distribution du calcul du vecteur `y` est réalisée pour générer les différentes tâches. La distribution du vecteur `y` implique également une distribution équivalente du vecteur `x`.

Les figures 6.6 et 6.7 présentent les accélérations obtenues par rapport au nombre de processus lancés.

La version OpenMP n'arrive pas a fournir de réelle accélération. Sa mise en place a même un coût particulièrement visible, avec l'utilisation de deux processus où son accélération est de 0,64 (“petit”) et 0,49 (“grand”). Avec plus de processus, son temps d'exécution est équivalent au temps séquentiel.

Pour le cas “petit”, le temps d'exécution est très court, inférieur à la milliseconde. Le code généré par notre processus arrive tout de même à améliorer ce temps d'exécution jusqu'à 4 processus avec une accélération de 2. Au delà de 4 processus, le temps d'exécution ne diffère que très peu.

Pour le cas “grand”, une accélération est présente sur les deux machines. Sur la machine distribuée, cette accélération est même super-linéaire. Relativement peu de données sont à communiquer dans cet exemple. Les temps de communications sont donc négligeables. De plus, en distribué, chaque processus a son propre cache ; de ce fait, plus le nombre de processus utilisé augmente et plus les données ont de chance de tenir en cache, d'où l'accélération super-linéaire observée.

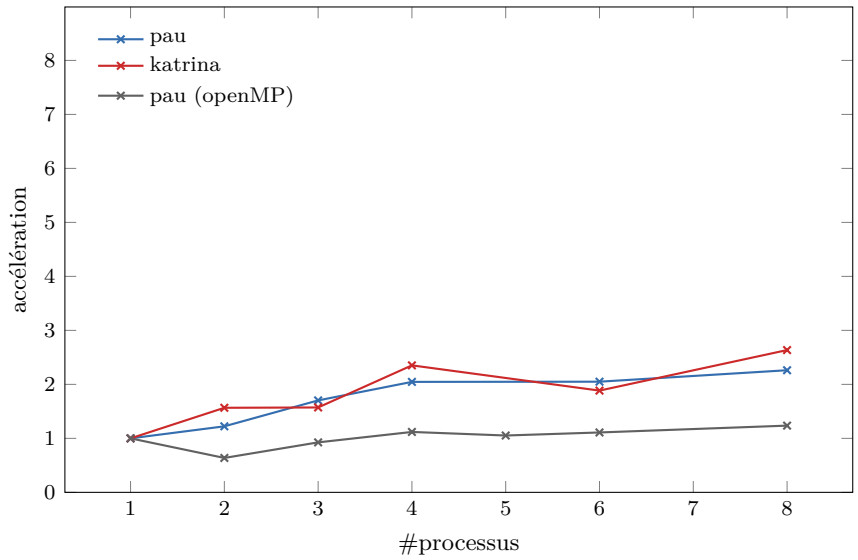


FIGURE 6.6 – Accélération de gesummv ("petit")

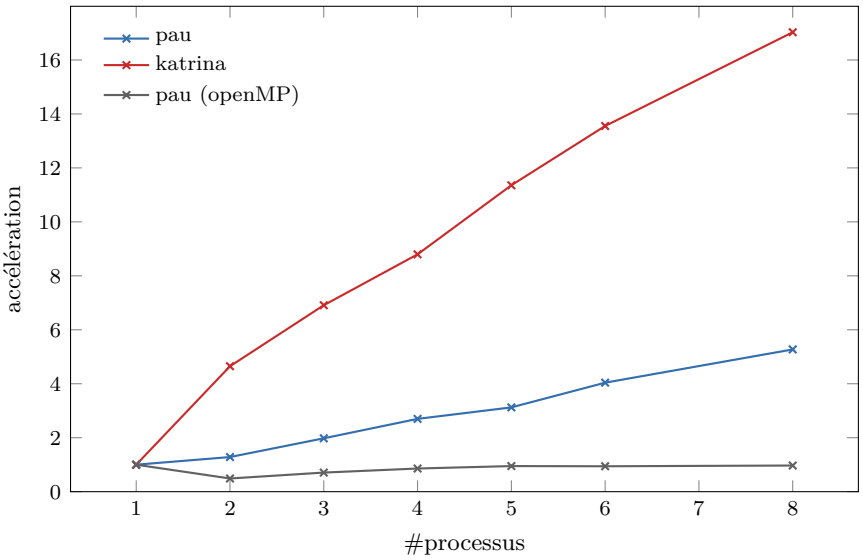


FIGURE 6.7 – Accélération de gesummv ("grand")

symm *Symmetric Matrix-Matrix Multiply* est une multiplication de matrices dont l'une des matrices est symétrique. Elle prend en entrée deux scalaires et trois tableaux :

— α, β, scalaires ;
— `A`, tableau symétrique de taille `M`×`M` ;
— `B`, `C`, tableaux de taille `M`×`N`.

Elle modifie en place le tableau `C` pour donner $\texttt{C} = \alpha\texttt{AB} + \beta\texttt{C}$. Elle se distingue par rapport à gemm par le fait que l'on suppose que seule une représentation triangulaire de `A` est connue.

Deux distributions sont réalisées pour mesurer les performances. Une distribution du calcul sur la première dimension du tableau `C` est réalisée pour générer les différentes tâches. Une autre distribution se fait par bloc sur les deux dimensions du tableau `C`.

Les figures 6.8 et 6.9 présentent les accélérations obtenues par rapport au nombre de processus lancés.

Pour le cas "petit", OpenMP a une accélération allant jusqu'à 2 pour 8 processus. Sur la même machine, Pau, la version générée par notre processus de compilation n'obtient aucune accélération dans le cas d'une distribution simple et une accélération de 1,5 avec 8 processus dans le cas d'une distribution par bloc. Sur la machine distribuée Katrina, de meilleurs résultats sont obtenus jusqu'à 6 processus exécutés pour cette version par bloc. Au delà de 6 processus, l'accélération chute fortement pour une raison inexpliquée. La distribution simple sur Katrina a une forte accélération pour 2 processus, mais elle diminue avec le nombre de processus utilisés.

Pour le cas "grand", l'accélération obtenue par OpenMP est parfaitement linéaire. Notre version où une simple distribution est utilisée ne permet pas de réelle accélération aussi bien sur Pau que sur Katrina. Pour la version distribuée par bloc, l'accélération sur Pau augmente et se stabilise à 2 pour 4 processus et plus. Sur Katrina, elle augmente et se stabilise à 3 pour 6 processus et plus.

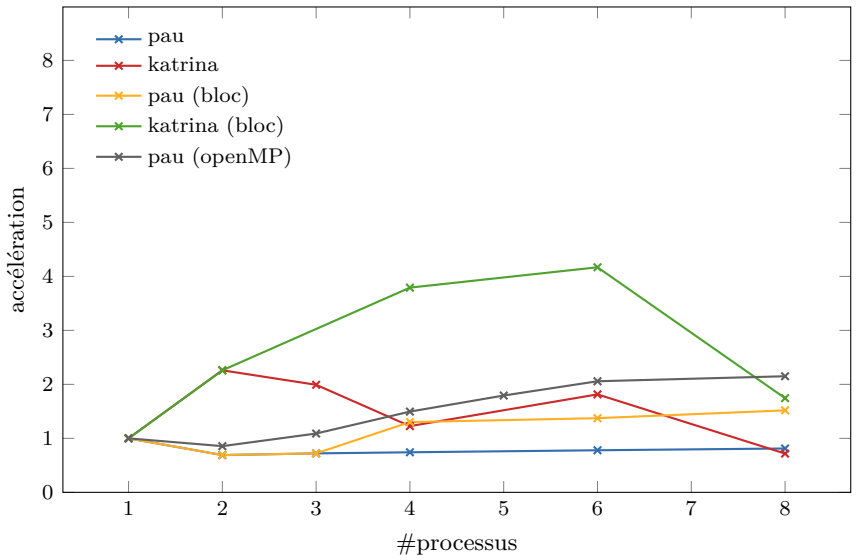


FIGURE 6.8 – Accélération de symm ("petit")

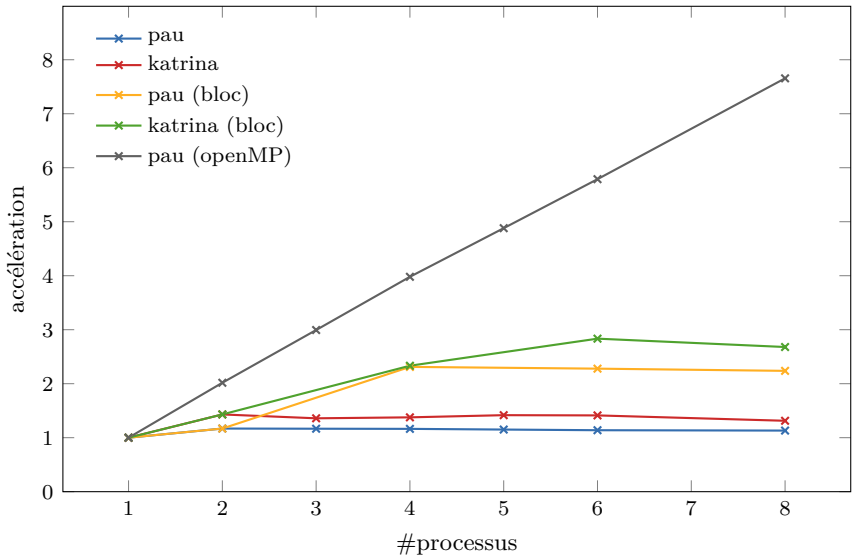


FIGURE 6.9 – Accélération de symm ("grand")

syrk *Symmetric Rank-k Update* est une multiplication de matrices de rang k. Elle prend en entrée deux scalaires et deux tableaux :

— α, β, scalaires ;
— `A`, tableau de taille `N`×`K` ;
— `C`, tableau symétrique de taille `N`×`N`.

Elle modifie en place le tableau `C` pour donner $\texttt{C} = \alpha\texttt{A}\texttt{A}^T + \beta\texttt{C}$. L'appellation matrice de rang k concerne la matrice `A` en entrée, qui est de taille `N`×`K`. Du fait que la matrice `C` est symétrique, seule une représentation triangulaire de `C` est calculée. L'implémentation est constituée d'une boucle calculant, dans un premier temps, $\beta\texttt{C}$, puis $\alpha\texttt{A}\texttt{A}^T + \beta\texttt{C}$.

Une distribution du calcul, sur la première dimension du tableau `C`, est utilisée pour générer les différentes tâches et effectuer les expériences.

Les figures 6.10 et 6.11 présentent les accélérations obtenues par rapport au nombre de processus lancés.

Dans le cas “petit”, le temps d'exécution d'OpenMP reste plus ou moins identique, quel que soit le nombre de processus utilisés. Par contre, une accélération linéaire voire super-linéaire est constatée aussi bien sur Pau que sur Katrina. Cette accélération super-linéaire sur Pau provient sans doute de l'imprécision des mesures. En effet, entre deux exécutions sur Pau, les temps sont de l'ordre de la milliseconde à la dizaine de millisecondes et vont du simple au double entre deux exécutions consécutives. Les résultats des temps d'exécution sur Katrina sont, quant à eux, plus homogènes. Cette accélération super-linéaire peut s'expliquer par une meilleure utilisation du cache. Des mesures avec PAPI ne sont malheureusement pas possibles, car PAPI n'est pas installé sur cette machine.

Dans le cas “grand”, une accélération linéaire est visible, correspondant à environ la moitié des processus utilisés. Les résultats obtenus au moyen du code généré par notre processus de compilation sont similaires aux résultats obtenus par le code OpenMP.

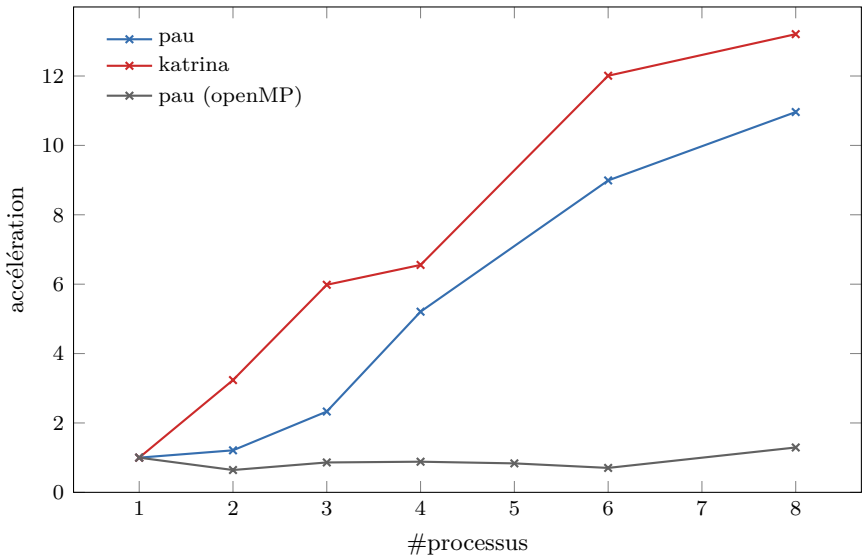


FIGURE 6.10 – Accélération de syrk ("petit")

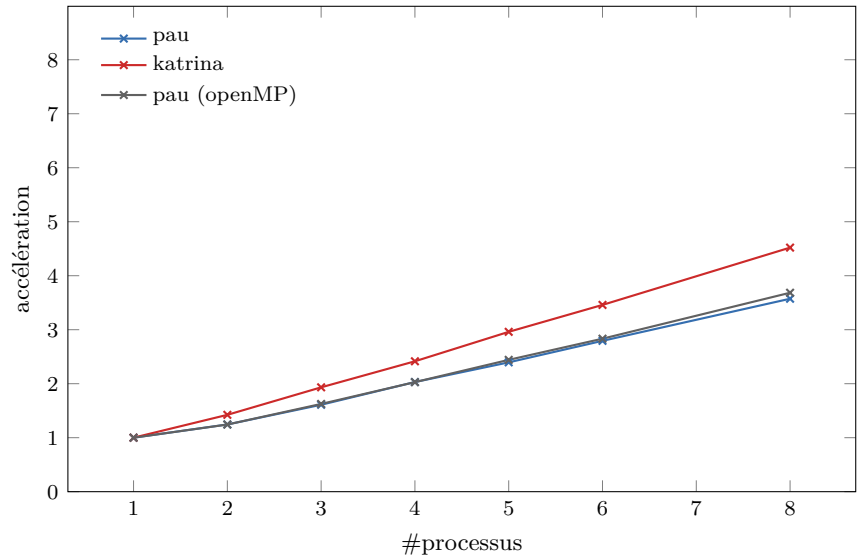


FIGURE 6.11 – Accélération de syrk ("grand")

syr2k *Symmetric Rank-2k update* est une multiplication de deux matrices de rang k. Elle prend en entrée deux scalaires et trois tableaux :

— α, β, scalaires ;
— `A`, `B`, tableau de taille `N`×`K` ;
— `C`, tableau symétrique de taille `N`×`N`.

Elle modifie en place le tableau `C` pour donner $\texttt{C} = \alpha\texttt{AB}^T + \alpha\texttt{BA}^T + \beta\texttt{C}$. L'appellation matrice de rang k concerne les matrices `A` et `B` en entrée, qui sont de taille `N`×`K`. Du fait que la matrice `C` est symétrique, seule une représentation triangulaire est calculée.

L'implémentation est constituée d'une boucle calculant, dans un premier temps, $\beta\texttt{C}$, puis $\alpha\texttt{AB}^T + \alpha\texttt{BA}^T + \beta\texttt{C}$.

Une distribution du calcul sur la première dimension du tableau `C` est réalisée pour générer les différentes tâches.

Les figures 6.12 et 6.13 présentent les accélérations obtenues par rapport au nombre de processus lancés.

Les résultats obtenus par syr2k sont très similaires à syrk.

Dans le cas “petit”, le temps d'exécution d'OpenMP est stable quel que soit le nombre de processus lancés. Aucune accélération ou décélération n'est donc à noter. L'accélération sur Pau est linéaire par rapport au nombre de processus exécutés. Une accélération super-linéaire sur Katrina est visible. Elle est encore plus remarquable que celle obtenue avec syrk ; son accélération est de 38 pour 8 processus, soit plus de 4 fois le nombre de processus présents.

Dans le cas “grand”, comme pour syrk, l'accélération est linéaire et égale à la moitié du nombre des processus exécutés. Les résultats sur Pau sont identiques, aussi bien pour la version OpenMP que pour notre version du code généré.

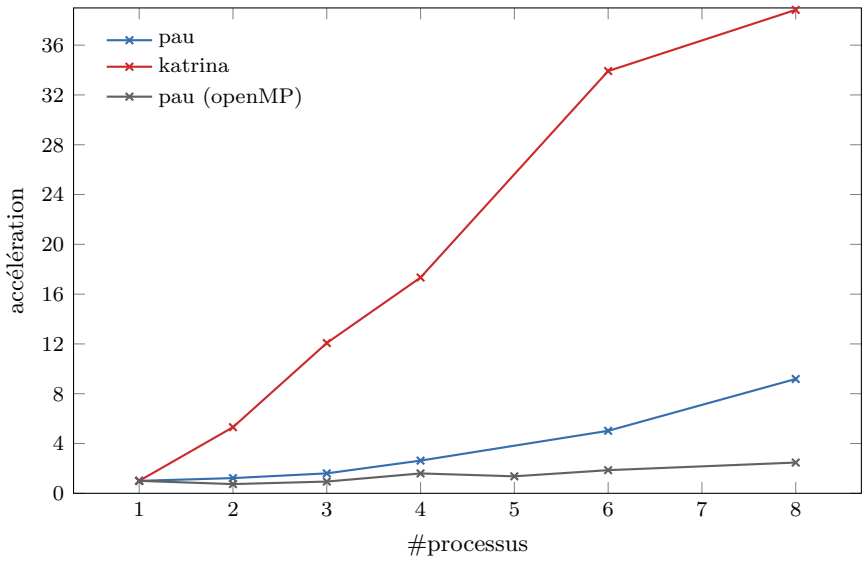


FIGURE 6.12 – Accélération de syr2k ("petit")

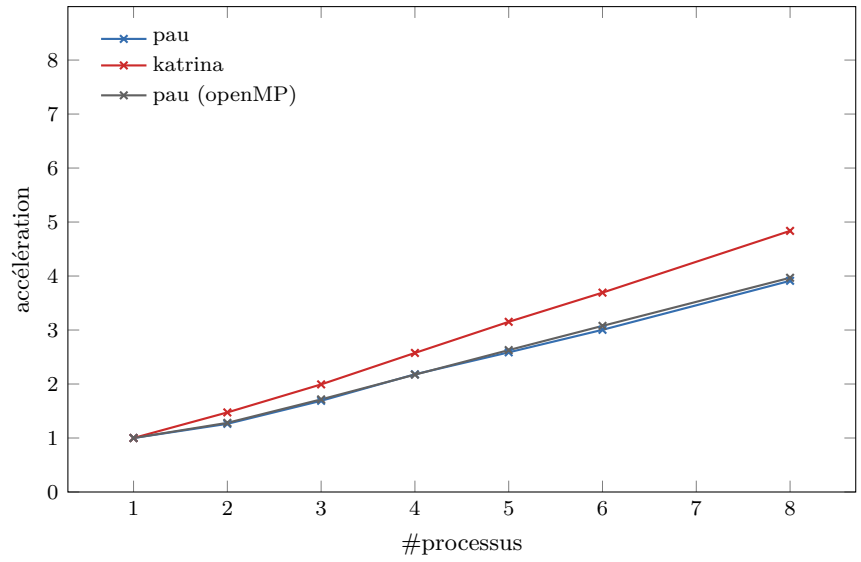


FIGURE 6.13 – Accélération de syr2k ("grand")

trmm *Triangular Matrix-Matrix Multiply* est une multiplication d'une matrice par une matrice triangulaire. Elle prend en entrée un scalaire et deux tableaux :

— α, scalaire ;
— `A`, tableau triangulaire de taille `N`×`N` ;
— `B`, tableau de taille `N`×`M`.

Elle modifie en place le tableau `B` pour donner `B` = α`AB`.

L'implémentation consiste en une boucle calculant, dans un premier temps, `AB`, puis α`AB`.

Deux distributions sont utilisées pour tester les performances. Une distribution du calcul sur la première dimension du tableau `B` est réalisée pour générer les différentes tâches. Une autre distribution se fait par bloc sur les deux dimensions du tableau `B`.

Les figures 6.14 et 6.15 présentent les accélérations obtenues par rapport au nombre de processus lancés.

Dans le cas "petit", au delà de 4 processus, l'accélération obtenue par OpenMP se stabilise entre 3 et 4. Une faible accélération est visible pour l'exécution sur la machine à mémoire partagée Pau, la distribution par bloc étant très légèrement meilleure avec le nombre de processus augmentant. Sur Katrina, l'accélération obtenue est meilleure, avec une accélération de 6 pour 8 processus. Tout comme sur Pau, la distribution par bloc obtient de meilleurs résultats. Un ralentissement avec 4 processus avec une distribution simple se produit sur Katrina. La raison est inexpliquée. L'amélioration obtenue entre une distribution simple et une distribution par bloc est d'environ 5%.

Dans le cas "grand", OpenMP obtient une accélération linéaire parfaite. La version générée exécutée sur la même machine obtient, quant à elle, une accélération correspondant à la moitié du nombre de processus présents. Une distribution par bloc permet également d'améliorer ces résultats de 5%. Sur Katrina, le temps d'exécution pour une distribution simple passe de 438s en séquentiel à 272s avec 8 processus, soit une accélération de 1,61. Par contre, la distribution par bloc permet d'obtenir de meilleurs résultats avec un temps d'exécution de 196s avec 8 processus, soit une accélération de 2,23. Ces accélérations sont néanmoins beaucoup plus faibles que celles obtenues sur Pau.

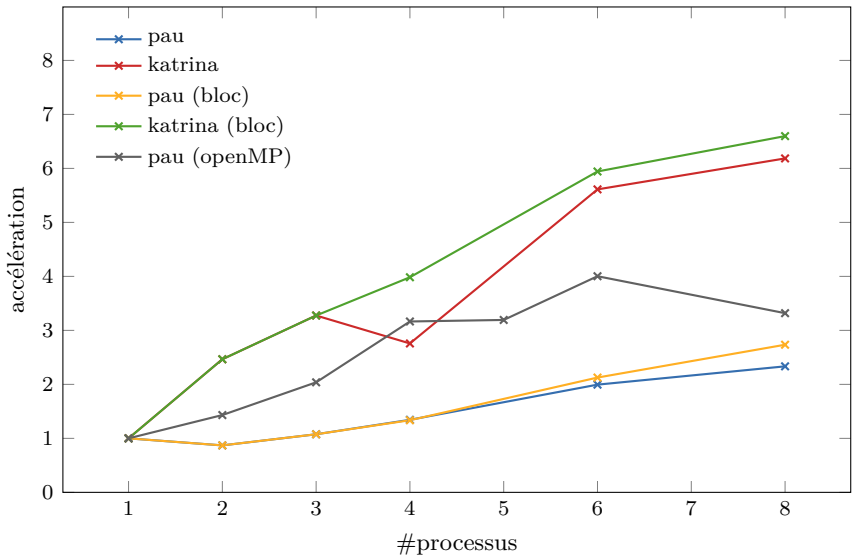


FIGURE 6.14 – Accélération de trmm ("petit")

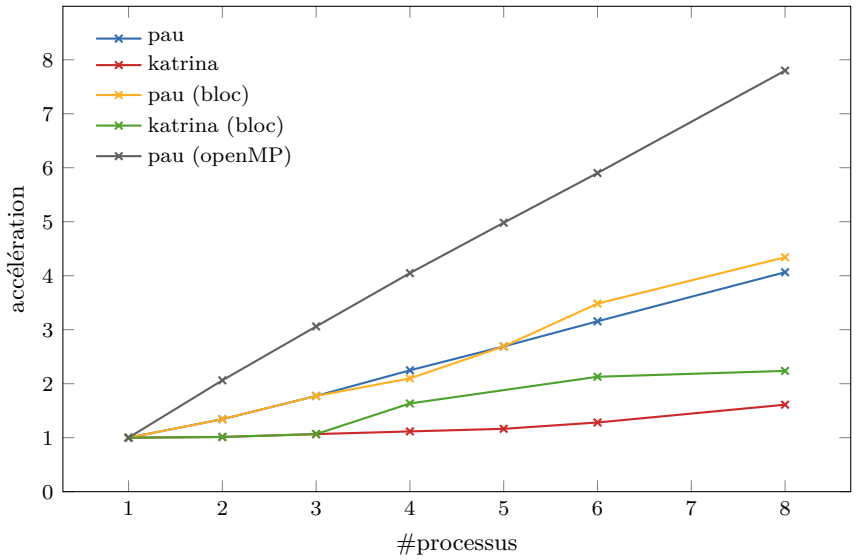


FIGURE 6.15 – Accélération de trmm ("grand")

6.5 De l'exploration de données

Deux algorithmes d'exploration de données, *datamining*, ont été étudiés. Le premier est le calcul d'une matrice de covariance. Le deuxième est le calcul du coefficient de corrélation de Pearson [87]. Ces deux algorithmes sont présents dans Polybench.

Les détails des temps d'exécution et d'accélération de ces expériences sont présentés en Annexe B.2. Ces deux algorithmes sont présentés ci-dessous.

Covariance La matrice de covariance est une mesure statistique permettant de quantifier la variation entre chacune des variables par rapport aux autres. Elle prend en entrée un tableau

— `data`, tableau de taille `N`×`M`, où `N` représente le nombre de points à étudier, les variables, et `M` représente les caractéristiques de ces points.

Elle génère un tableau `cov` de taille `M`×`M`, où le $(i,j)^{ème}$ élément est la covariance entre i et j. La matrice `cov` est une matrice symétrique et se calcule ainsi :

$$\texttt{cov}(i,j) = \frac{\sum_{k=0}^{N-1} (\texttt{data}(k,i) - \texttt{mean}(i))(\texttt{data}(k,j) - \texttt{mean}(j))}{N-1}$$

avec

$$\texttt{mean}(x) = \frac{\sum_{k=0}^{N-1} \texttt{data}(k,x)}{N}$$

L'implémentation réalisée consiste en trois tâches dépendantes successives. La première consiste à calculer les valeurs moyennes du tableau intermédiaire $\texttt{mean}(j)$. La deuxième calcule la différence entre les données étudiées et leur valeur moyenne $\texttt{data}(i,j) - \texttt{mean}(j)$. La dernière tâche calcule la matrice de covariance résultante `cov`. La matrice covariance est calculée sur 4000 points ($\texttt{N} = 4000$) possédant 3840 caractéristiques ($\texttt{M} = 3840$).

Les trois tâches précédentes étant dépendantes les une des autres, une distribution de chacune de ces tâches sur les différents processus est effectuée. Cette distribution est effectuée de manière à limiter les communications entre ces trois tâches. De plus, deux versions de ce code ont été réalisées. La première calcule la matrice de covariance dans sa globalité. La deuxième calcule uniquement une représentation triangulaire de la matrice de covariance. Étant donné que la matrice de covariance est symétrique, toutes les informations sont présentes dans cette représentation triangulaire.

La Figure 6.16 représente l'accélération obtenue par rapport au nombre de processus lancés.

La version calculant la matrice de covariance dans son ensemble ne permet pas l'obtention de bonnes performances. Cela s'explique, car la dernière des trois tâches, décrites précédemment, est la plus lente. Or, dans le cas où la matrice est calculée dans son ensemble, on ne peut pas distribuer une partie du tableau à l'un des processus et une autre partie à un autre processus, car la dernière tâche calcule uniquement une sous-matrice triangulaire, puis effectue une copie sur son symétrique. De ce fait, malgré une distribution de la dernière tâche, celle-ci s'exécute de façon séquentielle, empêchant tout gain réel de performance. En ne retournant que la représentation triangulaire de la matrice de covariance,

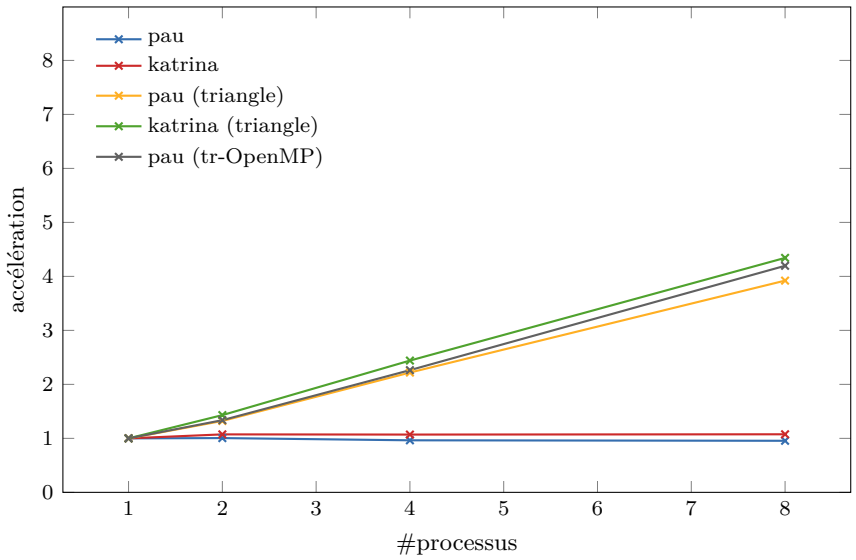


FIGURE 6.16 – Accélération de covariance ("grand")

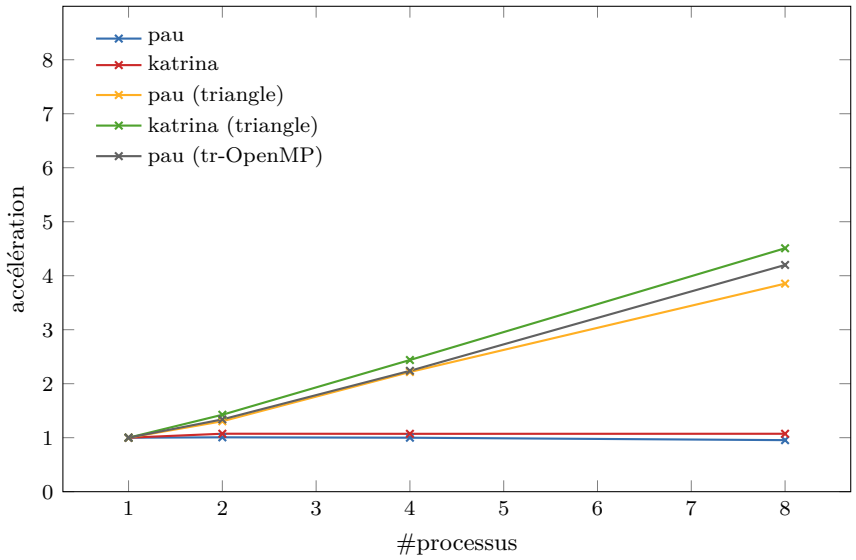


FIGURE 6.17 – Accélération de corrélation ("grand")

ce problème est évité. Une accélération linéaire par rapport à la moitié des processus exécutés peut ainsi être obtenue. Ces résultats sont équivalents à ceux obtenus par OpenMP.

Corrélation La matrice de corrélation est une forme normalisée de la matrice de covariance. Elle prend en entrée un tableau

— `data`, tableau de taille `N`×`M`, où `N` représente les points à étudier, les variables, et `M` représente les caractéristiques de ces points.

Elle génère un tableau `cor` de taille `M`×`M`, où le $(i,j)^{ème}$ élément est le coefficient de corrélation entre i et j. `cor` est une matrice symétrique et se calcule ainsi :

$$\texttt{cor}(i,j) = \frac{\texttt{cov}(i,j)}{\texttt{stddev}(i)\texttt{stddev}(j)}$$

avec

$$\texttt{stddev}(x) = \sqrt{\frac{\sum_{k=0}^{N-1}(\texttt{data}(k,x) - \texttt{mean}(x))^2}{N}}$$

L'implémentation réalisée consiste en quatre tâches dépendantes successives. La première calcule les valeurs moyennes du tableau intermédiaire $\texttt{mean}(j)$. La deuxième consiste à calculer les déviations standard du tableau intermédiaire $\texttt{stddev}(j)$. La troisième effectue une réduction sur les données $\frac{\texttt{data}(i,j)-\texttt{mean}(j)}{\sqrt[N]{\texttt{stddev}(j)}}$. La dernière tâche calcule la matrice de corrélation résultante `cor`. La matrice de corrélation est calculée sur 4000 points ($\texttt{N} = 4000$) possédant 3840 caractéristiques ($\texttt{M} = 3840$).

De même que pour le calcul de la matrice de covariance, une distribution de chacune des tâches sur tous les processus est réalisée de sorte à limiter les communications générées. Deux versions sont également étudiées. Une première calcule toute la matrice de corrélation. Une autre version calcule uniquement les informations nécessaires, à savoir une représentation triangulaire.

La Figure 6.17 représente l'accélération obtenue par rapport au nombre de processus lancés.

Des résultats similaires au problème du calcul de la matrice de covariance sont obtenus pour le calcul de la matrice de corrélation. Le résultat pour le calcul de la matrice entière ne permet pas d'obtenir d'accélération pour les mêmes raisons. Lorsque seule une représentation triangulaire est calculée, une accélération linéaire par rapport à la moitié du nombre de processus exécutés est obtenue. Ces résultats sont équivalents à ceux obtenus par OpenMP.

6.6 Du traitement d'image

Un algorithme de traitement d'image est étudié. Il s'agit d'un algorithme de détection de coins d'Harris et Stephens [44]. Cet algorithme n'est pas présent dans Polybench. J'ai donc réalisé une implémentation simple de celle-ci.

Les détails des temps d'exécution et d'accélération de ces expériences sont présentés en Annexe B.3. La présentation de cet algorithme et de ces résultats est décrite ci-après.

Harris&Stephens L'algorithme d'Harris et Stephens est un algorithme de détection de points d'intérêt dans une image, à savoir des coins. Il est réalisé par une succession de tâches de calculs. Il applique des filtres de Sobel et de Gauss, des multiplications de matrices et finit par la détection des coins. La Figure 6.18 montre les différents calculs à effectuer et les utilisations des différents résultats pour chacun de ces calculs. Elle correspond à un graphe de dépendances de tâches pour l'algorithme d'Harris&Stephens. Ces tâches sont distribuées sur des processus différents pour pouvoir être exécutés en parallèle.

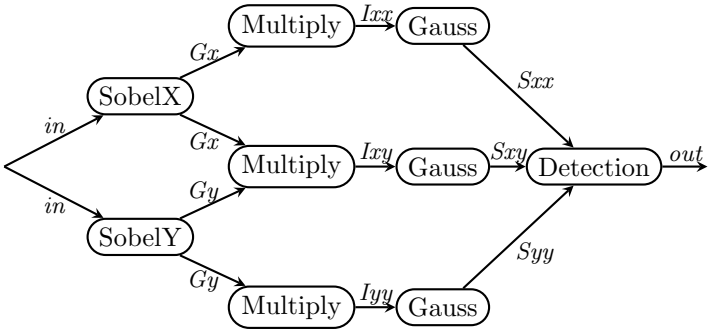


FIGURE 6.18 – Graphe de dépendances de tâches d'Harris&Stephens

D'après la Figure 6.18, on peut intuitivement distribuer les tâches sur 3 processus différents. La première partie des calculs, le filtre de Sobel, peut être exécuté sur deux processus simultanément. La multiplication de matrice et le filtre de Gauss sont exécutés sur 3 processus et le calcul de détection des coins sur un processus final. Une variante (b) consiste à redistribuer le calcul final de détection des coins sur les trois processus présents. Une autre distribution possible est de réaliser une distribution au sein des différentes tâches. Ainsi, toutes les tâches sont exécutées par deux processus au lieu d'un, portant le nombre de processus nécessaire à six.

Il faut noter que le compilateur sur Katrina a du mal à compiler certains des codes générés, au point que certaines des versions du code généré provoquent une erreur interne au compilateur alors que le même code sur Pau ou sur d'autres machines compile parfaitement. Par exemple, si un *inlining* des fonctions appelées est effectuée, sur Katrina, gcc 4.4.7 n'arrive pas à compiler le code et retourne une erreur interne au compilateur : `erreur interne du compilateur: dans change_address_1, à emit-rtl.c:1956`.

Les figures 6.19 et 6.20 présentent les accélérations obtenues par rapport au nombre de processus lancés. La petite version d'`harris` sur Katrina avec 6 processus lancés semble avoir un temps d'exécution infinie. Après plus de quatre heures d'attente, le programme tourne toujours, alors que ses versions séquentielle et sur trois processus terminent en quelques millisecondes.

Sur Pau, dans le cas "petit", le programme décélère et met environ un tiers de temps en plus pour finir. Dans le cas "grand", une légère accélération peut être constatée. Le Tableau 6.2 donne les temps d'exécution des calculs et de communications effectués dans les cas "petit" et "grand", pour 3 et 6 processus sur Pau. Dans le cas "petit", on constate que la communication, après le filtre de Gauss, récupérant les données pour le calcul de la détection des coins, correspond au temps d'exécution du programme séquentiel. Cette communication

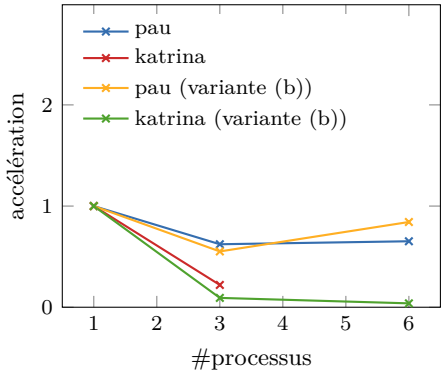


FIGURE 6.19 – Accélération d'`harris` ("petit")

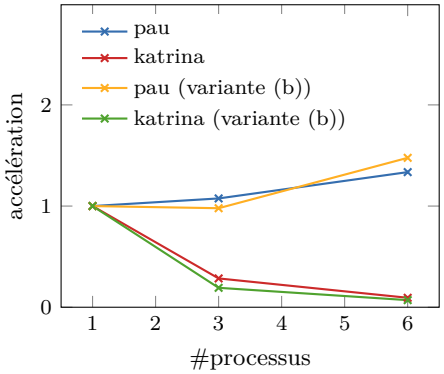


FIGURE 6.20 – Accélération d'`harris` ("grand")

est prédominante et empêche ainsi toute accélération du code parallèle généré. Le temps de calcul du code parallèle généré est d'environ 1,1 milliseconde pour 3 processus et 0,9 milliseconde pour 6 processus. Il est plus de deux fois plus court que le temps de calcul séquentiel. Mais en ajoutant, le temps de communication, qui équivaut au temps séquentiel, notre code généré s'exécute 1,5 fois plus lentement que sa version séquentielle. Dans le cas "grand", cette même communication est toujours dominante dans le temps d'exécution parallèle. Elle n'atteint néanmoins pas le temps d'exécution séquentielle. L'exécution parallèle permet de compenser ce long temps de communications, mais ne permet pas d'avoir une accélération significative, on constate au mieux une accélération de 1,47.

	"petit" (en ms)		"grand" (en ms)	
	3 procs	6 procs	3 procs	6 procs
Sobel	0,305	0,128	86,113	48,562
Communication	0,669	0,063	88,999	44,718
Multiply + Gauss	0,616	0,643	184,638	145,889
Communication	3,514	2,132	297,843	269,456
Detection (+ com sur 6 proc)	0,276	0,205	100,8	102,135

TABLE 6.2 – Temps d'exécution détaillé d'`harris` sur Pau

Sur Katrina, l'impact des temps de communications est encore plus flagrant. En effet, sur Pau, en mémoire partagée, les communications peuvent se traduire par des copies mémoire. Or, elles prennent déjà un temps significatif, alors avec de réelles communications inter-processeurs, ces communications sont d'autant plus dominantes. Ainsi, autant dans le cas "petit" que dans le "grand", l'exécution parallèle est plus lente que l'exécution séquentielle.

La variante (b), redistribuant la dernière tâche sur tous les processus, n'apporte pas d'amélioration significative.

6.7 Synthèse des résultats obtenus

Ce chapitre a permis d'évaluer les performances obtenues par le code parallèle qui est généré automatiquement par notre processus de compilation. Ces résultats sont résumés dans les figures 6.21 à 6.24, montrant l'accélération obtenue pour nos exemples sur Pau et Katrina.

Les premières expériences ont été réalisées sur sept problèmes d'algèbre linéaire classique appartenant à BLAS. Elles ont montré qu'une accélération peut facilement être obtenue dans la plupart des cas, avec une simple distribution par ligne. Une distribution par bloc, lorsqu'elle est possible, permet d'améliorer ces résultats dans deux de ces cas, grâce à une meilleure localité des données. Des expériences sur des problèmes d'exploration de données ont également été réalisées. Elles permettent de montrer que le processus de compilation décrit fonctionne sur des codes plus complexes. Pour obtenir de bonnes performances sur ces problèmes, il a été nécessaire de considérer les données utiles calculées par ces programmes et constater que de limiter au calcul d'une représentation triangulaire était suffisant. Enfin, un problème de traitement d'image a été traité. Il a montré l'importance et l'impact que des communications nombreuses peuvent avoir sur le code.

Dans les cas "petit" (figures 6.21 et 6.22), en mémoire partagée sur Pau, notre processus de compilation permet d'obtenir de bien meilleurs résultats que la version d'OpenMP sur `gemm`, `syrk`, et `syr2k` et des résultats légèrement meilleurs, équivalents ou légèrement moins bons dans les autres cas. Ces résultats s'expliquent par le fait que, avec un ensemble de données réduit et en mémoire partagée, la version OpenMP réalise beaucoup plus souvent des invalidations de lignes de caches. Ces invalidations entraînent un rechargement du cache fréquent qui diminue le gain que peut apporter la parallélisation. *A contrario*, dans notre code généré, chaque processus possède sa propre copie des tableaux à traiter ; de ce fait peu d'invalidations de lignes de cache se produisent. De plus, toutes ces données peuvent tenir dans le cache. Donc il n'y a pas ou peu besoin de le recharger, d'où de bien meilleures performances possibles. En mémoire distribuée, sur Katrina, le surcoût que peut engendrer la communication des données est souvent compensé par le gain apporté par la parallélisation. Une accélération est donc toujours présente dans nos exemples, à l'exception d'`harris`.

Dans les cas "grand" (figures 6.23 et 6.24), en mémoire partagée, sur Pau, mis à part `gemver` qui décélère, nos exemples permettent d'obtenir une accélération. Dans un exemple, `gemsummv`, l'accélération obtenue par notre code généré est supérieure à celle obtenue par OpenMP. Dans trois des exemples, `gemm`, `symm` et `trmm`, le code OpenMP obtient de meilleurs résultats. Dans les quatre exemples restants, nos performances sont équivalentes à celles d'OpenMP. En mémoire distribuée, sur Katrina, `gemver` et `harris` ont de très mauvaises performances et `symm` a une accélération faible. La majorité des autres exemples présente une accélération linaire par rapport au nombre de processus exécutés. L'impact des communications est donc beaucoup plus visible dans ce cas. `gemver` doit communiquer deux matrices entières durant son exécution, l'une bidimensionnelle et l'autre unidimensionnelle. Ces communications empêchent en partie une exécution parallèle et ne sont pas compensées par les calculs effectués. `harris` doit communiquer deux tableaux bidimensionnels entiers avant de pourvoir faire son calcul final. D'un autre coté, `gesummv` obtient une accélération super-linéaire. Seule une matrice unidimensionnelle a besoin d'être récupérée à

la fin de nos résultats, contrairement aux autres exemples qui traitent de matrices bidimensionnelles. De plus, dans cet exemple, le découpage du traitement entraîne rapidement la possibilité de faire tenir toutes les données nécessaires au calcul sur les caches de plus haut niveau des différents processus.

6.8 Conclusion

En pratique, la mise en place des expériences a été laborieuse, dû au manque d'un interfaçage adapté aux boucles. En effet, l'interface de programmation que nous avons proposée permet uniquement de considérer des tâches comme étant des séquences d'instructions. Lorsqu'une tâche se compose d'une partie des itérations d'une boucle à exécuter, il est nécessaire de réaliser des partitionnements avant de pouvoir affecter chacune de ces parties de tableaux à un processus. La génération de ces codes doit être précédée d'un prétraitement à appliquer avant de pouvoir réaliser notre processus de compilation automatique.

Malgré cela, trois classes de problèmes ont pu être traitées en considérant deux tailles d'ensemble de données. Elles ont permis de vérifier que notre processus de compilation est correct, en générant un code dont le résultat est toujours identique à celui du code initial. De plus, une accélération est visible sur la totalité des cas traités, excepté le cas de `gemver` et d'`harris`.

Avec une petite taille de l'ensemble des données, une exécution de notre code en mémoire partagée est souvent meilleure qu'en OpenMP. Notre processus de compilation permet donc d'éviter, dans de petits exemples, les problèmes d'invalidation de cache qui peuvent se produire avec une utilisation simple d'OpenMP.

Dans les exemples nécessitant un plus grand espace mémoire, si les communications ne sont pas trop nombreuses, l'augmentation du nombre de processus utilisé permet d'obtenir une accélération moyenne correspondant à la moitié du nombre de processus exécutés. En mémoire distribuée, elle permet même des accélérations super-linéaires. Cette accélération super-linéaire s'explique par le fait que chaque processus peut mieux utiliser son cache et moins de rechargements de cache sont nécessaires, vu que la quantité de données nécessaire pour calculer chaque portion des données diminue.

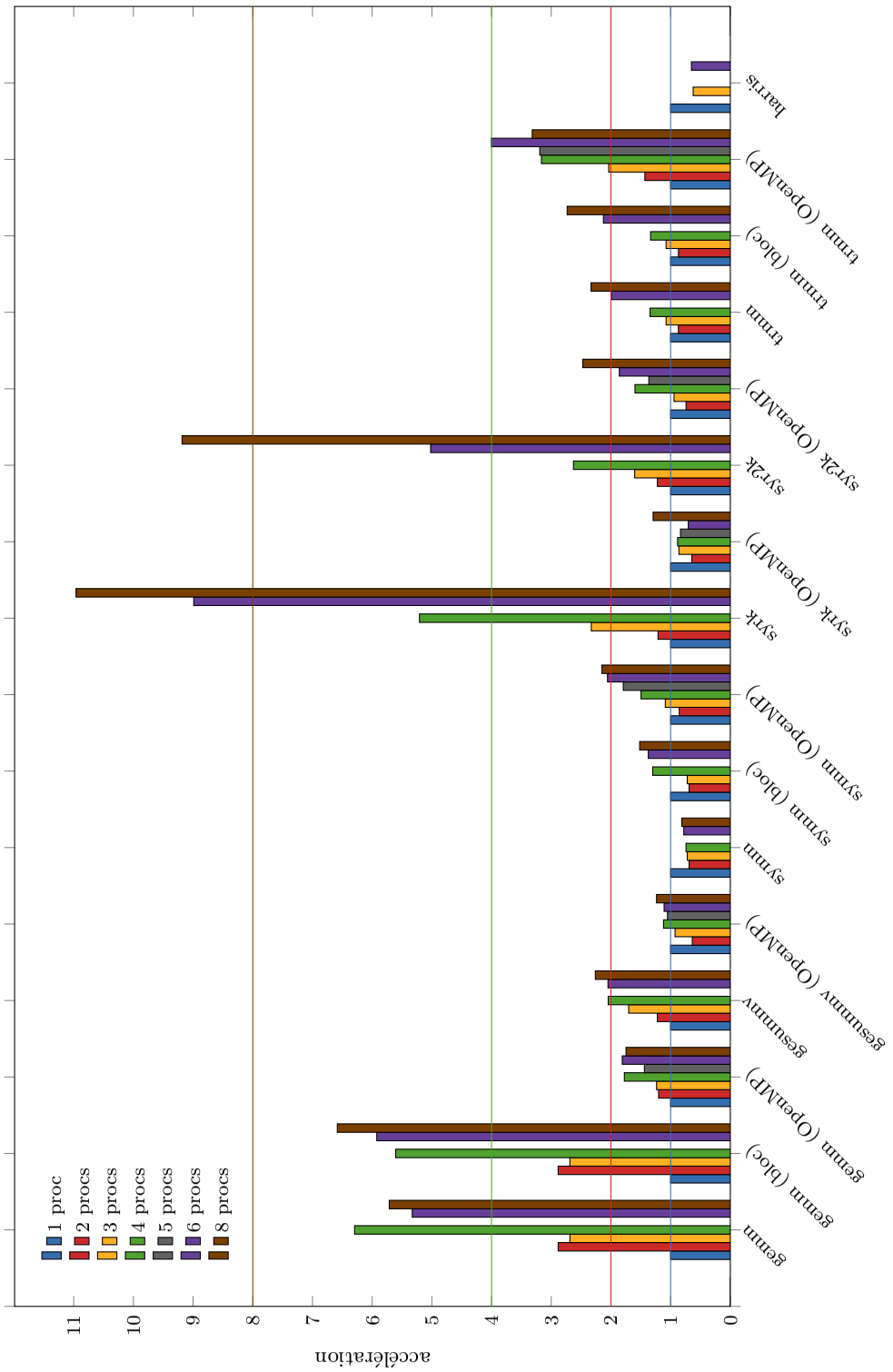


FIGURE 6.21 – Accélération sur Pau (“petit”)

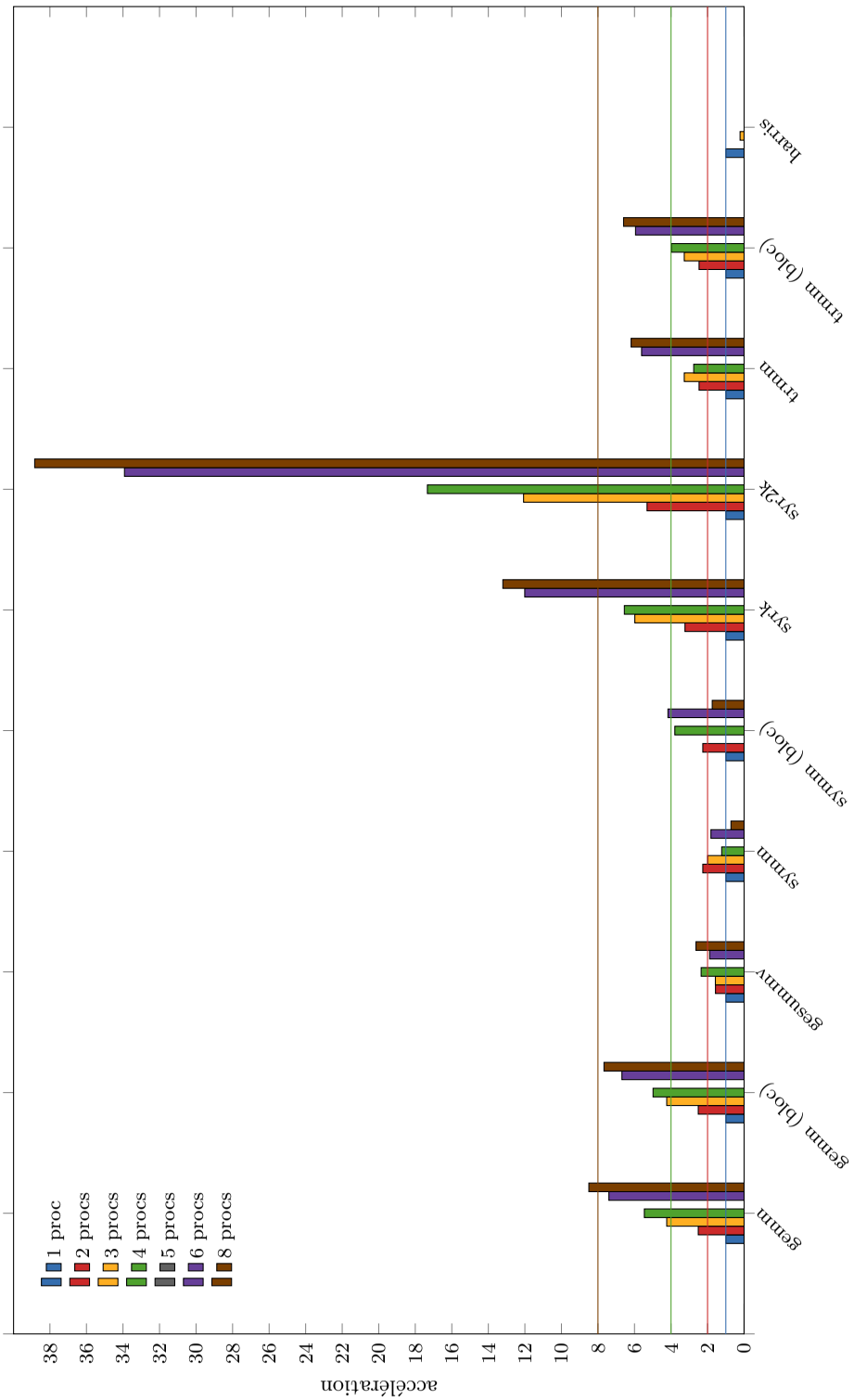


FIGURE 6.22 – Accélération sur Katrina (“petit”)

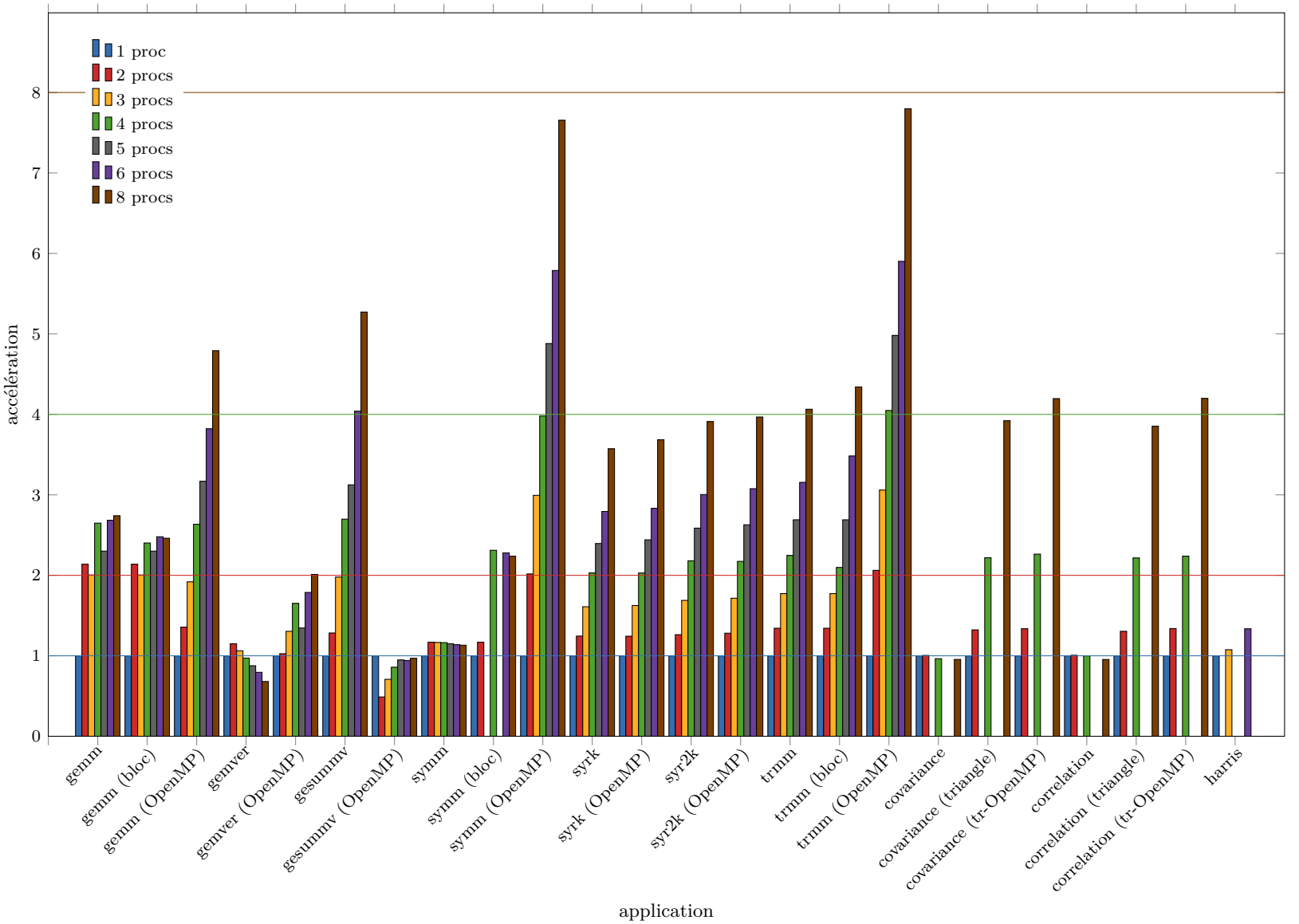


FIGURE 6.23 – Accélération sur Pau (“grand”)

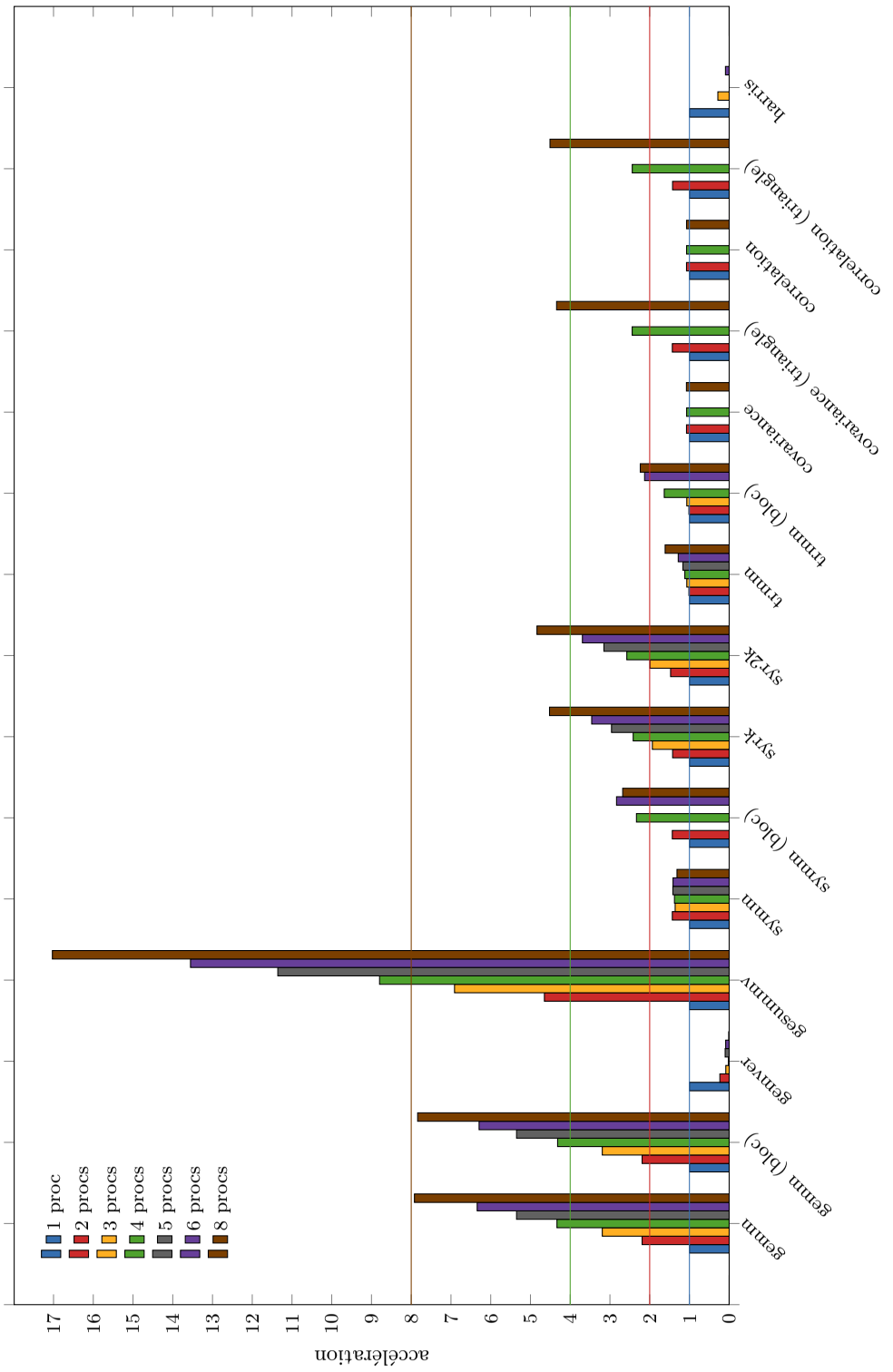


FIGURE 6.24 – Accélération sur Katrina (“grand”)

Conclusion

Synthèse

Cette thèse a montré qu'il est possible de générer automatiquement un code parallèle distribué en tâches à partir d'un code séquentiel annoté.

Le chapitre 1 a rappelé dans un premier temps les différentes architectures et modèles de programmation qui existent pour du code parallèle. Les contraintes pour la génération automatique de code distribué pour des tâches ont été exposées au chapitre 2. Elles soulèvent les différentes problématiques : 1) la gestion et la répartition des données sur les différents processus ; 2) la gestion de l'ordonnancement des tâches sur les différents processus. Des outils (dSTEP, Pluto+) permettant de répondre partiellement à ces problèmes y ont été décrits. Le cheminement menant à la solution que nous proposons y a été augmenté en mettant en avant le souci de correction du code généré, de compréhension pour l'utilisateur des transformations effectuées et de prouvabilité. Une description formelle du processus de compilation permettant la génération automatique du code désiré a été présentée au chapitre 3. La syntaxe et la sémantique du langage considéré y sont décrites, et les différentes transformations réalisées sont présentées et expliquées avec cette syntaxe. Cette description est accompagnée d'une preuve de correction des transformations (chapitre 4). Ces preuves ont consisté à montrer que le code généré par chacune des transformations est équivalent au code avant transformation. Des améliorations des transformations précédentes et de nouvelles optimisations ont été réalisées et présentées au chapitre 5. Elles permettent notamment de réduire le nombre de communications générées et de réduire l'espace mémoire nécessaire pour l'exécution du programme. Enfin, des expériences ont été réalisées pour valider l'efficacité du code généré et leurs résultats sont discutés au chapitre 6. Elles montrent des accélérations du code généré. Ces accélérations vont même dans la majorité des cas, jusqu'à la moitié du nombre de processus utilisés, *ie.* une accélération de 4 pour 8 processus exécutés.

Contributions

La première contribution de cette thèse a été la conception et l'implémentation d'une chaîne de compilation permettant de **générer automatiquement un code parallèle pour une architecture à mémoire distribuée pour des tâches** en utilisant une suite de transformations élémentaires et de prouver que cette génération automatique est **correcte** en vérifiant la correction de chacunes des transformations élémentaires.

La seconde contribution est la définition d'**une interface de programmation simple de description des tâches** à placer. Elle utilise des directives `pragma distributed`. Ces directives peuvent être ignorées par les compilateurs classiques permettant ainsi de garder le code d'origine exécutable. Pour les compilateurs ayant intégré ces directives, comme le compilateur source-à-source PIPS, elles permettent d'exécuter le processus de compilation décrit au cours de cette thèse et de générer le code parallèle distribué correspondant.

La génération automatique est faite au moyen d'**un processus de compilation composé de quatre phases** : **placement** des tâches, **préparation** du code, **optimisation** du code et **génération** du code parallèle. Ces phases se composent de plusieurs transformations qui sont individuellement prouvées correctes. Ce processus de compilation est **modulaire** : différentes versions des transformations, plus ou moins complexes, peuvent être utilisées et des optimisations ajoutées. La description de ce processus de compilation a fait l'objet d'un article [3].

Toutes ces améliorations et optimisations supplémentaires font partie des autres contributions réalisées au sein de cette thèse. Ces améliorations et optimisations ne sont pas propres à mon processus de compilation et peuvent être utilisées dans le cas général de la compilation et de l'optimisation de code. Ces contributions sont les suivantes :

- Une description du **graphe des effets prenant en compte les déclarations** dans le contexte de la **compilation source-à-source**. L'implémentation a été réalisée par Medhi Amini, et j'ai pu décrire ses avantages et les gains apportés dans les articles [4][5]. Parmi ceux-ci, il y a notamment la possibilité de garder un code lisible tel que l'utilisateur initial l'avait écrit, et de choisir le graphe de dépendances à utiliser pour une transformation donnée.
- Une optimisation de code permettant d'**éliminer les itérations de boucles inutiles**, ***dead iteration elimination***. Cette optimisation est en quelque sorte une extension de la traditionnelle élimination de code mort, *dead code elimination*. Au lieu d'éliminer une instruction inutile, elle élimine les itérations de boucle inutiles en réduisant les bornes de l'intervalle d'itérations de la boucle. Une réduction des bornes implique une réduction des calculs à effectuer lors de l'exécution et donc une exécution plus rapide.
- Une optimisation de code permettant de **réduire la taille des tableaux** en C, ***array resizing***. Cette optimisation permet de réduire l'espace mémoire utilisé par un programme lors de son exécution. Elle réduit la taille des tableaux utilisés lorsque c'est possible et modifie les références à ces tableaux pour qu'elles soient correctes.
- Une optimisation permettant d'**éliminer les instructions d'affectation représentant une identité**, ***identity elimination***. Cette optimisation permet, comme son nom l'indique, de supprimer des affectations identité. Pour le moment, elle est limitée aux identités "simples", lorsque l'on affecte une variable à elle-même. Mais, elle pourrait être étendue en évaluant la valeur de la variable à affecter et celle devant être affectée, et en vérifiant qu'elles sont différentes, sinon l'affectation peut être supprimée.
- Une amélioration de l'**analyse des Régions OUT** présente dans PIPS. Elle consiste à forcer le polyèdre des contraintes des régions OUT à ne

contenir que des variables vivantes lorsque c'est possible. Cette amélioration est utile pour toutes les transformations de code qui génèrent du code à partir des régions. Elle permet ainsi d'éviter de rendre vivante une variable censée être morte à un point du programme.

— Une nouvelle **description de la notion de *IN* et *OUT***, introduite par Béatrice Creusillet [28].

$$OUT(s) = DEF(s) \cap LIVE_{out}(s)$$
$$IN(s) = USED(s) \cap LIVE_{in}(s)$$

Cette nouvelle description met en relation les variables IN et OUT d'une instruction avec les variables vivantes et les variables lues et écrites par l'instruction. Cette nouvelle description est vérifiée lorsque l'on considère les variables en tant qu'entité indivisible. Elle doit encore être étendue pour les régions. Pour cela, une extension de la description précise d'une variables vivante pour les régions doit également être réalisée.

Perspectives

Suite à nos expériences, plusieurs améliorations sont à envisager. On peut notamment citer les suivantes :

— Une **nouvelle description de *IN* et *OUT* étendue aux régions**. Cette extension nécessite notamment de définir formellement les "régions vivantes", et comment elles sont calculées, notamment au niveau des tests et des boucles.

— La **génération de communications asynchrones** en lieu et place des communications synchrones, ou une optimisation transformant des communications synchrones en communications asynchrones. L'utilisation de communications asynchrones implique l'ajout de variables de notification et une vérification de ces variables avant l'utilisation des données envoyées ou reçues. Cette problématique a été abordée dans la section 5.5.2.

— L'**extension de notre interface** de description des tâches à des boucles. L'interface permettant d'assigner une tâche à un processeur est limitée à des tâches représentant des séquences d'instructions. Cela a particulièrement été contraignant lors de la phase d'expérimentation où il a été nécessaire de réaliser un découpage des boucles préalable au placement des instructions, *ie.* des tâches, sur des processus. Cette extension propose donc d'ajouter une nouvelle directive `pragma distributed for N` s'appliquant sur des boucles et faisant un découpage (*strip mining* + *unrolling*) de cette boucle en `N` tâches affectées à `N` processus. Plusieurs paramètres facultatifs peuvent être ajoutés pour cibler plusieurs types de distributions, par exemple par ligne, bloc ou cyclique.

— L'**extraction des instructions exécutées par un processus** dans des fonctions indépendantes. Notre processus de compilation engendre une multiplication des variables déclarées sur tous les processus exécutant le programme, alors que seul un sous-ensemble de ces variables est utilisé par chaque processus. La conception du langage C et de la bibliothèque MPI ne permettent pas d'affecter des variables uniquement à certains processus. Pour réduire l'espace mémoire alloué intempestive-

ment, chaque processus doit avoir une fonction qui lui est dédiée et qui ne déclarera que les variables qui lui sont allouées.

— La **génération de code parallèle dans et pour d'autres langages**. Une extension de mes contributions au langage Fortran77 serait facilement réalisable, étant donné que PIPS analyse déjà cette version de Fortran. Cependant, des ajustements doivent être faits au niveau des directives qui n'existent pas en Fortran et des paramètres des fonctions MPI qui diffèrent légèrement. D'autres langages de sortie peuvent également être envisagés notamment des langages PGAS tels que UPC ou X10.

Annexes

Annexe A

Description et fonctionnement de PIPS

Cette annexe présente une description très sommaire de PIPS et de son fonctionnement. Pour plus de détails concernant PIPS, il faut se référer au site `http://www.pips4u.org/` [88]. La mise en place de la programmation de PIPS au moyen d'Eclipse est présentée dans un rapport technique [2].

PIPS est un *framework* de compilation de code source-à-source pour l'analyse et la transformation de code `C` et Fortran.

Le *workspace* PIPS a besoin d'un *workspace* pour s'exécuter. Ce *workspace* correspond au fichier ou à l'ensemble des fichiers à traiter. Il permettra entre autre de stocker les ressources calculées pour les traitements à faire.

Les règles et propriétés L'exécution d'une analyse ou d'une transformation s'effectue au moyen de règles ou d'ensembles de règles, appelé passe ou phase. Par abus, on utilise indifféremment ces trois termes.

Plusieurs types de règles existent dans PIPS, elles sont principalement réunies en deux groupes : celles qui permettent de réaliser une analyse de code en générant des ressources dans le *workspace* ; et celles qui permettent de réaliser une transformation directement dans le code.

Nous pouvons citer quelques passes d'analyses comme l'analyse des effets, l'analyse des *points-to*, l'analyse des *transformers* et les *pretty-printers* correspondant. Les *pretty-printers* sont des passes spéciales qui permettent d'ajouter en commentaires les résultats obtenus par une analyse dans le code.

Ces passes sont paramétrables au moyen de propriétés.

L'interface script *tpips* On utilise l'interface *tpips* permettant d'exécuter un traitement au moyen de lignes de commandes. Elle permet également d'écrire des scripts. Dans *tpips*, la gestion du *workspace* se fait par les commandes `create`, `open`, `close` ou `delete`. L'utilisation explicite d'une règle plutôt qu'une autre nécessite la commande `activate`. Enfin la configuration d'une propriété demande la commande `setproperty`. Des règles et des propriétés par défaut sont déjà présentes lors de l'utilisation de *tpips*.

Un script *tpips* se compose principalement de quatre parties. Au début, on crée un *workspace*. Puis, on configure les propriétés. Ensuite, l'analyse et les transformations sont effectuées. Enfin, on ferme et détruit, si nécessaire, le *workspace*.

Annexe B

Résultats détaillés des expériences

B.1 Algèbre linéaire

	Temps d'exécution (en ms)					
	1	2	3	4	6	8
gemm pau	44.09	15.28	16.40	7.00	8.27	7.71
katrina	52.84	21.05	12.47	9.67	7.13	6.21
gemm pau	44.08	15.28	16.40	7.86	7.44	6.69
(bloc) katrina	52.84	21.05	12.47	10.60	7.90	6.88
gemm pau (OpenMP)	44.08	36.68	35.68	24.83	24.35	25.226
gesummv pau	0.386	0.315	0.226	0.188	0.188	0.170
katrina	0.686	0.438	0.436	0.291	0.364	0.260
gesummv pau (OpenMP)	0.386	0.605	0.417	0.345	0.348	0.312
symm pau	59.68	86.44	82.69	80.41	76.56	73.46
katrina	168.77	74.66	84.71	137.57	93.01	235.02
symm pau	59.68	86.44	82.69	45.86	43.43	39.29
(bloc) katrina	168.77	74.66	-	44.51	40.48	96.74
symm pau (OpenMP)	59.68	69.73	54.78	39.90	29.00	27.75
syrk pau	29.35	18.45	9.59	4.29	2.48	2.03
katrina	29.94	9.25	5.00	4.56	2.49	2.26
syrk pau (OpenMP)	29.35	45.60	34.07	33.23	41.68	22.69
syr2k pau	64.67	52.93	40.35	24.60	12.88	7.04
katrina	101.74	19.15	8.42	5.86	2.99	2.62
syr2k pau (OpenMP)	64.67	87.24	68.58	40.49	34.77	26.16
trmm pau	62.53	71.90	58.13	46.54	31.34	26.79
katrina	92.30	37.44	28.18	33.47	16.44	14.92
trmm pau	62.53	71.90	58.13	46.81	29.41	22.87
(bloc) katrina	92.30	37.44	28.18	23.16	15.53	13.98
trmm pau (OpenMP)	62.53	43.68	30.68	19.76	15.62	18.84

TABLE B.1 – Algèbre linéaire : temps d'exécution ("petit")

	Temps d'exécution (en s)						
	1	2	3	4	5	6	8
gemm pau	41.57	19.44	20.74	15.69	18.07	15.48	15.17
katrina	115.19	52.58	36.08	26.58	21.53	18.17	14.55
gemm pau	41.57	19.44	20.74	17.31	18.07	16.76	16.89
(bloc) katrina	115.19	52.58	36.08	26.68	21.53	18.31	14.70
gemm pau (OpenMP)	41.57	30.65	21.66	15.78	13.12	10.88	8.67
gemver pau	0.327	0.284	0.307	0.336	0.372	0.411	0.480
katrina	0.458	1.957	5.400	22.076	4.519	5.203	45.808
gemver pau (OpenMP)	0.327	0.318	0.250	0.198	0.243	0.183	0.162
gesummv pau	24.88	19.38	12.58	9.22	7.96	6.15	4.72
(en ms) katrina	161.29	34.69	23.34	18.33	14.20	11.90	9.47
gesummv pau (OpenMP)	24.88	50.91	35.16	28.98	26.23	26.42	25.68
symm pau	659.36	563.92	565.46	566.77	573.08	579.29	582.66
katrina	266.73	186.44	196.27	193.73	188.24	188.81	203.11
symm pau	659.36	563.92	-	285.30	-	289.38	294.65
(bloc) katrina	266.73	186.44	-	114.34	-	94.11	99.55
symm pau (OpenMP)	659.36	326.84	220.23	165.68	135.10	113.92	86.13
syrk pau	245.73	197.43	152.68	121.03	102.59	87.96	68.77
katrina	92.60	65.07	47.91	38.34	31.28	26.77	20.48
syrk pau (OpenMP)	245.73	197.61	151.22	121.11	100.65	86.73	66.69
syr2k pau	637.89	505.26	377.69	292.63	246.62	212.36	163.08
katrina	176.43	119.71	88.53	68.50	55.98	47.78	36.48
syr2k pau (OpenMP)	637.89	498.01	371.88	293.49	242.73	207.34	160.78
trmm pau	670.82	499.62	378.38	298.44	249.36	212.59	165.06
katrina	437.95	432.24	410.68	392.33	376.15	341.71	271.86
trmm pau	670.82	499.62	378.38	319.71	249.36	192.58	154.52
(bloc) katrina	437.95	432.24	410.68	268.07	-	205.77	195.78
trmm pau (OpenMP)	670.82	325.32	219.23	165.70	134.66	113.67	86.02

TABLE B.2 – Algèbre linéaire : temps d'exécution ("grand")

	Accélération					
	1	2	3	4	6	8
gemm pau	1	2.88	2.69	6.29	5.33	5.71
katrina	1	2.51	4.24	5.46	7.40	8.50
gemm pau	1	2.88	2.69	5.61	5.92	6.58
(bloc) katrina	1	2.51	4.24	4.98	6.69	7.67
gemm pau (OpenMP)	1	1.20	1.24	1.78	1.81	1.75
gesummv pau	1	1.22	1.70	2.05	2.05	2.26
katrina	1	1.57	1.57	2.35	1.88	2.64
gesummv pau (OpenMP)	1	0.64	0.93	1.12	1.11	1.24
symm pau	1	0.69	0.72	0.74	0.78	0.81
katrina	1	2.26	1.99	1.23	1.81	0.72
symm pau	1	0.69	0.72	1.30	1.37	1.52
(bloc) katrina	1	2.26	-	3.79	4.17	1.74
symm pau (OpenMP)	1	0.86	1.09	1.50	2.06	2.15
syrk pau	1	1.21	2.33	5.21	8.99	10.96
katrina	1	3.24	5.98	6.55	12.01	13.21
syrk pau (OpenMP)	1	0.64	0.86	0.88	0.70	1.29
syr2k pau	1	1.22	1.60	2.63	5.02	9.18
katrina	1	5.31	12.07	17.34	33.92	38.83
syr2k pau (OpenMP)	1	0.74	0.94	1.60	1.86	2.47
trmm pau	1	0.87	1.08	1.34	1.99	2.33
katrina	1	2.47	3.28	2.76	5.61	6.18
trmm pau	1	0.87	1.08	1.34	2.13	2.73
(bloc) katrina	1	2.47	3.28	3.98	5.94	6.60
trmm pau (OpenMP)	1	1.43	2.04	3.16	4.00	3.32

TABLE B.3 – Algèbre linéaire : accélération (“petit”)

	Accélération						
	1	2	3	4	5	6	8
gemm pau	1	2.14	2.00	2.65	2.30	2.68	2.74
katrina	1	2.19	3.19	4.33	5.35	6.34	7.92
gemm pau	1	2.14	2.00	2.40	2.30	2.48	2.46
(bloc) katrina	1	2.19	3.19	4.32	5.35	6.29	7.84
gemm pau (OpenMP)	1	1.36	1.92	2.63	3.17	3.82	4.79
gemver pau	1	1.15	1.06	0.97	0.88	0.79	0.68
katrina	1	0.23	0.08	0.02	0.10	0.09	0.01
gemver pau (OpenMP)	1	1.03	1.30	1.65	1.35	1.79	2.01
gesummv pau	1	1.28	1.98	2.70	3.12	4.04	5.27
katrina	1	4.65	6.91	8.80	11.36	13.55	17.03
gesummv pau (OpenMP)	1	0.49	0.71	0.86	0.95	0.94	0.97
symm pau	1	1.17	1.17	1.16	1.15	1.14	1.13
katrina	1	1.43	1.36	1.38	1.42	1.41	1.31
symm pau	1	1.17	-	2.31	-	2.28	2.24
(bloc) katrina	1	1.43	-	2.33	-	2.83	2.68
symm pau (OpenMP)	1	2.02	2.99	3.98	4.88	5.79	7.66
syrk pau	1	1.24	1.61	2.03	2.40	2.79	3.57
katrina	1	1.42	1.93	2.42	2.96	3.46	4.52
syrk pau (OpenMP)	1	1.24	1.63	2.03	2.44	2.83	3.68
syr2k pau	1	1.26	1.69	2.18	2.59	3.00	3.91
katrina	1	1.47	1.99	2.58	3.15	3.69	4.84
syr2k pau (OpenMP)	1	1.28	1.72	2.17	2.63	3.08	3.97
trmm pau	1	1.34	1.77	2.25	2.69	3.16	4.06
katrina	1	1.01	1.07	1.12	1.16	1.28	1.61
trmm pau	1	1.34	1.77	2.10	2.69	3.48	4.34
(bloc) katrina	1	1.01	1.07	1.63	-	2.13	2.24
trmm pau (OpenMP)	1	2.06	3.06	4.05	4.98	5.90	7.80

TABLE B.4 – Algèbre linéaire : accélération (“grand”)

B.2 Exploration de données

	Temps d'exécution (en s)				Accélération			
	1	2	4	8	1	2	4	8
covariance pau	632	629	656	662	1	1.01	0.96	0.96
katrina	1954	1820	1826	1817	1	1.07	1.07	1.08
covariance pau	630	476	284	161	1	1.32	2.22	3.92
(triangle) katrina	1952	1365	800	450	1	1.43	2.44	4.34
covariance pau (tr-OpenMP)	630	471	278	150	1	1.34	2.26	4.20
correlation pau	632	628	632	662	1	1.01	1.00	0.95
katrina	1950	1817	1819	1818	1	1.07	1.07	1.07
correlation pau	621	476	280	161	1	1.30	2.22	3.85
(triangle) katrina	1943	1364	797	431	1	1.42	2.44	4.51
correlation pau (tr-OpenMP)	621	464	278	148	1	1.34	2.24	4.20

TABLE B.5 – Exploration de données : temps d'exécution ("grand")

B.3 Traitement d'image

	Temps d'exécution (en ms)			Accélération		
	1	3	6	1	3	6
harris pau	2.46	3.96	3.78	1	0.62	0.65
katrina	2.14	9.75	-	1	0.22	-
harris pau	2.46	4.47	2.92	1	0.55	0.84
(variante (b)) katrina	2.14	23.26	56.02	1	0.09	0.04

TABLE B.6 – Traitement d'image : temps d'exécution et accélération ("petit")

	Temps d'exécution (en s)			Accélération		
	1	3	6	1	3	6
harris pau	0.820	0.762	0.614	1	1.08	1.34
katrina	1.984	6.969	21.248	1	0.28	0.09
harris pau	0.820	0.838	0.555	1	0.98	1.48
(variante (b)) katrina	1.984	10.340	28.244	1	0.19	0.07

TABLE B.7 – Traitement d'image : temps d'exécution et accélération ("grand")

Bibliographie personnelle

[1] Nelson Lossing. *Analyse sémantique des tableaux, des structures et des pointeurs*. Rapp. tech. http://www.cri.ensmp.fr/classement/doc/E-322.pdf. MINES ParisTech, août 2013.

[2] Nelson Lossing. *PIPS : Tutorial to use PIPS with Eclipse*. Rapp. tech. http://www.cri.ensmp.fr/classement/doc/E-334.pdf. MINES ParisTech, 2014.

[3] Nelson Lossing, Corinne Ancourt et Francois Irigoin. "Automatic Code Generation of Distributed Parallel Tasks". In : *Computational Science and Engineering (CSE), 2016 IEEE 19th International Conference on*. doi : 10.1109/CSE-EUC-DCABES.2016.190. Août 2016, p. 234–241.

[4] Nelson Lossing, Pierre Guillou, Mehdi Amini et Francois Irigoin. "From Data to Effects Dependence Graphs : Source-to-Source Transformations for C". In : *18th International Workshop on Compilers for Parallel Computing (CPC 2015)* (jan. 2015). http://www.cri.ensmp.fr/classement/doc/A-596.pdf.

[5] Nelson Lossing, Pierre Guillou et Francois Irigoin. "Effects Dependence Graph : A Key Data Concept for C Source-to-Source Compilers". In : *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. doi : 10.1109/SCAM.2016.20. Oct. 2016, p. 167–176.

Références

[6] Aravind Acharya et Uday Bondhugula. “PLUTO+ : Near-complete Modeling of Affine Transformations for Parallelism and Locality”. In : *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. San Francisco, CA, USA : ACM, 2015, p. 54–64.

[7] Laksono Adhianto, Guohua Jin, M. Krentel, John Mellor-Crummey, William N. Scherer III, C. Yang et F. Zhao. *Coarray Fortran 2.0 Prealpha Release Notes*. Rapp. tech. Rice University, avr. 2012.

[8] Vikram Adve, Guohua Jin, John Mellor-Crummey et Qing Yi. “High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes”. In : *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC ’98. San Jose, CA : IEEE Computer Society, 1998, p. 1–23.

[9] Alfred V. Aho, Ravi Sethi, Jeffrey Ullman et Monica S. Lam. *Compilers : principles, techniques, & tools*. 2nd. Boston : Pearson/Addison Wesley, 2006.

[10] AMD. *3DNow ! Technology Manual*. Mar. 2000. url : https://support.amd.com/TechDocs/21928.pdf (visité le 02/02/2017).

[11] Corinne Ancourt, Fabien Coelho, Francois Irigoin et Ronan Keryell. “A Linear Algebra Framework for Static High Performance Fortran Code Distribution”. In : *Sci. Program.* 6.1 (jan. 1997), p. 3–27.

[12] Corinne Ancourt et Thi Viet Nga Nguyen. “Array Resizing for Scientific Code Debugging, Maintenance and Reuse”. In : *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’01. Snowbird, Utah, USA : ACM, 2001, p. 32–37.

[13] J. W. Backus. “The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference”. In : *Proceedings of the International Conference on Information Processing, UNESCO*. Paris, 1960, p. 125–131.

[14] Dirk Bartz, Bengt-Olaf Schneider et Claudio Silva. “Rendering and Visualization in Parallel Environments”. In : *ACM SIGGRAPH 2000 Courses*. SIGGRAPH ’00. New Orleans, Louisiana : ACM, 2000.

[15] Gordon Bell et Jim Gray. “What’s Next in High-performance Computing ?” In : *Commun. ACM* 45.2 (fév. 2002), p. 91–95.

[16] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula et Albert Cohen. “Automatic Storage Optimization for Arrays”. In : *ACM Trans. Program. Lang. Syst.* 38.3 (avr. 2016), 11 :1–11 :23.

[17] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington et R. C. Whaley. “An Updated Set of Basic Linear Algebra Subprograms (BLAS)”. In : *ACM Trans. Math. Softw.* 28.2 (juin 2002), p. 135–151.

[18] BLAS. *Basic Linear Algebra Subprograms*. URL : http://www.netlib.org/blas/ (visité le 02/02/2017).

[19] F. Bodin, L. Kervella et T. Priol. “Fortran-S : A Fortran Interface for Shared Virtual Memory Architectures”. In : *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing ’93. Portland, Oregon, USA : ACM, 1993, p. 274–283.

[20] 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*. SC ’13. Denver, Colorado : ACM, 2013, 33 :1–33 :12.

[21] Uday Bondhugula, Aravind Acharya et Albert Cohen. “The Pluto+ Algorithm : A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests”. In : *ACM Trans. On Programming Languages and Systems (TOPLAS)* (2016).

[22] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron et Nicolas Vasilache. “Tiling and Optimizing Time-iterated Computations on Periodic Domains”. In : *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada : ACM, 2014, p. 39–50.

[23] Uday Bondhugula, Vinayaka Bandishti et Irshad Pananilath. “Diamond Tiling : Tiling Techniques to Maximize Parallelism for Stencil Computations”. In : *IEEE Transactions on Parallel and Distributed Systems* (2016).

[24] BSC Programming Models group. *The OmpSs Programming Model*. URL : https://pm.bsc.es/ompss (visité le 02/02/2017).

[25] Barbara Chapman, Piyush Mehrotra et Hans Zima. “Programming in Vienna Fortran”. In : *Sci. Program.* 1.1 (jan. 1992), p. 31–50.

[26] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao et Daniel Chavarría-Miranda. “An Evaluation of Global Address Space Languages : Co-array Fortran and Unified Parallel C”. In : *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’05. Chicago, IL, USA : ACM, 2005, p. 36–47.

[27] Fabien Coelho. “Contributions à la compilation du High Performance Fortran”. Theses. Ecole Nationale Supérieure des Mines de Paris, 1996.

[28] Beatrice Creusillet. “Analyse de régions de tableaux et applications”. Theses. Ecole Nationale Supérieure des Mines de Paris, 1996.

[29] Roshan DATHATHRI, Chandan REDDY, Thejas RAMASHEKAR et Uday BONDHUGULA. "Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-memory". In : *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*. PACT '13. Edinburgh, Scotland, UK : IEEE Press, 2013, p. 375–386.

[30] J. J. DONGARRA, Jeremy DU CROZ, Sven HAMMARLING et I. S. DUFF. "A Set of Level 3 Basic Linear Algebra Subprograms". In : *ACM Trans. Math. Softw.* 16.1 (mar. 1990), p. 1–17.

[31] J. DONGARRA, T. STERLING, H. SIMON et E. STROHMAIER. "High-performance computing : clusters, constellations, MPPs, and future directions". In : *Computing in Science Engineering* 7.2 (mar. 2005), p. 51–59.

[32] Jack J. DONGARRA, Jeremy DU CROZ, Sven HAMMARLING et Richard J. HANSON. "An Extended Set of FORTRAN Basic Linear Algebra Subprograms". In : *ACM Trans. Math. Softw.* 14.1 (mar. 1988), p. 1–17.

[33] Paul FEAUTRIER. "Parametric integer programming". eng. In : *RAIRO - Operations Research - Recherche Opérationnelle* 22.3 (1988), p. 243–268.

[34] J. A. FISHER. "Trace Scheduling : A Technique for Global Microcode Compaction". In : *IEEE Trans. Comput.* 30.7 (juil. 1981), p. 478–490.

[35] Joseph A. FISHER, Paolo FARABOSCHI et Cliff YOUNG. *Embedded Computing : A VLIW Approach to Architecture, Compilers and Tools*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005.

[36] Michael J. FLYNN. "Some Computer Organizations and Their Effectiveness". In : *IEEE Trans. Comput.* 21.9 (sept. 1972), p. 948–960.

[37] Thierry GAUTIER, Joao V. F. LIMA, Nicolas MAILLARD et Bruno RAFFIN. "XKaapi : A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures". In : *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IPDPS '13. Washington, DC, USA : IEEE Computer Society, 2013, p. 1299–1308.

[38] GCC. *Auto-vectorization in GCC*. URL : http://gcc.gnu.org/projects/tree-ssa/vectorization.html (visité le 18/04/2016).

[39] Al GEIST, Adam BEGUELIN, Jack DONGARRA, Weicheng JIANG, Robert MANCHEK et Vaidy SUNDERAM. *PVM–parallel virtual machine : a users' guide and tutorial for networked parallel computing*. Scientific and engineering computation. Cambridge, Mass : MIT Press, 1994.

[40] Andrew D. GORDON et Tom MELHAM. "Five axioms of alpha-conversion". In : *Theorem Proving in Higher Order Logics : 9th International Conference, TPHOLs'96 Turku, Finland, August 26–30, 1996 Proceedings*. Sous la dir. de Gerhard GOOS, Juris HARTMANIS, Jan van LEEUWEN, Joakim von WRIGHT, Jim GRUNDY et John HARRISON. Berlin, Heidelberg : Springer Berlin Heidelberg, 1996, p. 173–190.

[41] Michael J. C. GORDON. *The Denotational Description of Programming Languages : An Introduction*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1979.

[42] Rachid HABEL. "High Performance Programming for Hybrid Architectures". Theses. Ecole Nationale Supérieure des Mines de Paris, nov. 2014.

[43] Rachid HABEL, Frédérique SILBER-CHAUSSUMIER, François IRIGOIN, Elisabeth BRUNET et François TRAHAY. "Combining Data and Computation Distribution Directives for Hybrid Parallel Programming : A Transformation System". In : *Int. J. Parallel Program.* 44.6 (déc. 2016), p. 1268–1295.

[44] Chris HARRIS et Mike STEPHENS. "A combined corner and edge detector". In : *In Proc. of Fourth Alvey Vision Conference*. 1988, p. 147–151.

[45] John L. HENNESSY et David A. PATTERSON. *Computer Architecture : A Quantitative Approach*. 5th. Waltham, MA : Morgan Kaufmann/Elsevier, 2012.

[46] Jonathan M.D. HILL, Bill MCCOLL, Dan C. STEFANESCU, Mark W. GOUDREAU, Kevin LANG, Satish B. RAO, Torsten SUEL, Thanasis TSANTILAS et Rob H. BISSELING. "BSPlib : The BSP programming library". In : *Parallel Computing* 24.14 (1998), p. 1947–1980.

[47] Seema HIRANANDANI, Ken KENNEDY et Chau-Wen TSENG. "Compiling Fortran D for MIMD Distributed-memory Machines". In : *Commun. ACM* 35.8 (août 1992), p. 66–80.

[48] HPF. *High Performance Fortan*. URL : http://hpff.rice.edu/ (visité le 02/02/2017).

[49] INTEL. *Getting Started with Intel® Cilk™ Plus Array Notations*. URL : https://software.intel.com/sites/default/files/article/185163/introduction-to-array-notation.pdf (visité le 02/02/2017).

[50] INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 : Basic Architecture*. Sept. 2016. URL : https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf (visité le 02/02/2017).

[51] INTEL. *Intel® Architecture Instruction Set Extensions Programming Reference*. Fév. 2016. URL : https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf (visité le 02/02/2017).

[52] F. IRIGOIN et R. TRIOLET. "Supernode Partitioning". In : *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA : ACM, 1988, p. 319–329.

[53] ISO/IEC/IEEE 9945 :2009(E). *Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*. Standard. International Organization for Standardization, sept. 2009.

[54] Ken KENNEDY et Randy ALLEN. *Optimizing Compilers for Modern Architectures : A Dependence-based Approach*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001.

[55] Ken KENNEDY, Charles KOELBEL et Hans ZIMA. "The Rise and Fall of High Performance Fortran : An Historical Object Lesson". In : *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California : ACM, 2007, p. 7-1–7-22.

[56] Dounia KHALDI. "Automatic Resource-Constrained Static Task Parallelization : A Generic Approach". Theses. Ecole Nationale Supérieure des Mines de Paris, nov. 2013.

[57] Dounia KHALDI, Pierre JOUVELOT et Corinne ANCOURT. "Parallelizing with BDSC, a Resource-constrained Scheduling Algorithm for Shared and Distributed Memory Systems". In : *Parallel Comput.* 41.C (jan. 2015), p. 66–89.

[58] KHRONOS GROUP. *Khronos*. URL : https://www.khronos.org/ (visité le 02/02/2017).

[59] KHRONOS GROUP. *OpenCL, The open standard for parallel programming of heterogeneous systems*. URL : https://www.khronos.org/opencl/ (visité le 02/02/2017).

[60] KHRONOS GROUP. *The OpenCL Specification, Version : 2.2*. Mar. 2016. URL : https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf (visité le 02/02/2017).

[61] Michael KLEMM et Christian TERBOVEN. "Full Throttle : OpenMP 4.0". In : *The Parallel Universe Magazine* 16 (2013), p. 6–16.

[62] Charles H. KOELBEL, éd. *The High performance Fortran handbook*. Scientific and engineering computation. Cambridge, Mass : MIT Press, 1994.

[63] Samuel LARSEN et Saman AMARASINGHE. "Exploiting Superword Level Parallelism with Multimedia Instruction Sets". In : *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada : ACM, 2000, p. 145–156.

[64] C. L. LAWSON, R. J. HANSON, D. R. KINCAID et F. T. KROGH. "Basic Linear Algebra Subprograms for Fortran Usage". In : *ACM Trans. Math. Softw.* 5.3 (sept. 1979), p. 308–323.

[65] Grzegorz MALEWICZ, Matthew H. AUSTERN, Aart J.C BIK, James C. DEHNERT, Ilan HORN, Naty LEISER et Grzegorz CZAJKOWSKI. "Pregel : A System for Large-scale Graph Processing". In : *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA : ACM, 2010, p. 135–146.

[66] William F. MCCOLL. "Foundations of Time-Critical Scalable Computing". In : *Fundamentals - Foundations of Computer Science, IFIP World Computer Congress 1998, August 31 - September 4, 1998, Vienna/Austria and Budapest/Hungary*. Sous la dir. de Kurt MEHLHORN. T. 117. Austrian Computer Society, 1998, p. 93–107.

[67] John MELLOR-CRUMMEY, Laksono ADHIANTO, William N. SCHERER III et Guohua JIN. "A New Vision for Coarray Fortran". In : *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. PGAS '09. Ashburn, Virginia, USA : ACM, 2009, 5 :1–5 :9.

[68] Daniel MILLOT, Alain MULLER, Christian PARROT et Frédérique SILBER-CHAUSSUMIER. "STEP : A Distributed OpenMP for Coarse-grain Parallelism Tool". In : *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*. IWOMP'08. West Lafayette, IN, USA : Springer-Verlag, 2008, p. 83–99.

[69] Daniel MILLOT, Alain MULLER, Christian PARROT et Frédérique SILBER-CHAUSSUMIER. "From OpenMP to MPI : first experiments of the STEP source-to-source transformation tool." In : *Parallel Computing : From Multicores and GPU's to Petascale, Proceedings of the conference ParCo*. 2009, p. 669–676.

[70] Millind MITTAL, Alex PELEG et Uri WEISER. "MMX^{TM} Technology Architecture Overview". In : *Intel Technology Journal* 1.3 (1997).

[71] MPI. *Message Passing Interface*. URL : https://www.mpi-forum.org/ (visité le 02/02/2017).

[72] MPICH. URL : https://www.mpich.org/ (visité le 02/02/2017).

[73] Robert H. B. NETZER et Barton P. MILLER. "What Are Race Conditions ? : Some Issues and Formalizations". In : *ACM Lett. Program. Lang. Syst.* 1.1 (mar. 1992), p. 74–88.

[74] John von NEUMANN. "First Draft of a Report on the EDVAC". In : *IEEE Ann. Hist. Comput.* 15.4 (oct. 1993), p. 27–75.

[75] Robert W. NUMRICH et John REID. "Co-array Fortran for Parallel Programming". In : *SIGPLAN Fortran Forum* 17.2 (août 1998), p. 1–31.

[76] Robert W. NUMRICH et John REID. "Co-arrays in the Next Fortran Standard". In : *SIGPLAN Fortran Forum* 24.2 (août 2005), p. 4–17.

[77] NVIDIA. *CUDA*. URL : https://developer.nvidia.com/cuda-zone (visité le 02/02/2017).

[78] NVIDIA. *CUDA Toolkit Documentation v7.5*. URL : http://docs.nvidia.com/cuda/index.html (visité le 02/02/2017).

[79] NVIDIA. *CuBLAS Library, User Guide*. Sept. 2016. URL : http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf (visité le 02/02/2017).

[80] NVIDIA. *CuSPARSE Library*. Sept. 2016. URL : http://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf (visité le 02/02/2017).

[81] OPEN MPI. URL : https://www.open-mpi.org/ (visité le 02/02/2017).

[82] OPENACC. URL : http://www.openacc.org/ (visité le 02/02/2017).

[83] OPENACC. *The OpenACC® Application Programming Interface, Version 2.5*. Oct. 2015. URL : http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf (visité le 02/02/2017).

[84] OPENMP. URL : http://www.openmp.org/ (visité le 02/02/2017).

[85] OPENMP ARB. *OpenMP Application Programming Interface, Version 4.5*. Nov. 2015.

[86] PAPI. *Performance Application Programming Interface*. URL : http://icl.utk.edu/papi/ (visité le 02/02/2017).

[87] Karl Pearson. “Note on Regression and Inheritance in the Case of Two Parents”. In : *Proceedings of the Royal Society of London* 58.347-352 (1895), p. 240–242.

[88] PIPS. URL : http://pips4u.org/ (visité le 02/02/2017).

[89] PLUTO. *An automatic parallelizer and locality optimizer for affine loop nests*. URL : http://pluto-compiler.sourceforge.net/ (visité le 02/02/2017).

[90] Polybench. URL : https://sourceforge.net/projects/polybench/ (visité le 02/02/2017).

[91] PVM. *Parallel Virtual Machine*. URL : http://www.csm.ornl.gov/pvm/pvm_home.html (visité le 02/02/2017).

[92] John Regehr. *Race Condition vs. Data Race*. Mar. 2011. URL : http://blog.regehr.org/archives/490 (visité le 02/02/2017).

[93] Gilles Requilé. *PVM "Parallel Virtual Machine", Les aspects communication*. URL : http://1995.jres.org/actes/appli4/1/requile.pdf (visité le 02/02/2017).

[94] Michael S. Schlansker et B. Ramakrishna Rau. “EPIC : Explicitly Parallel Instruction Computing”. In : *Computer* 33.2 (fév. 2000), p. 37–45.

[95] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake et Charles V. Packer. “Beowulf : A Parallel Workstation For Scientific Computation”. In : *In Proceedings of the 24th International Conference on Parallel Processing*. CRC Press, 1995, p. 11–14.

[96] V. S. Sunderam. “PVM : A Framework for Parallel Distributed Computing”. In : *Concurrency : Pract. Exper.* 2.4 (nov. 1990), p. 315–339.

[97] Herb Sutter. “The Free Lunch Is Over : A Fundamental Turn Toward Concurrency in Software”. In : *Dr. Dobb’s Journal* 30.3 (2005), p. 202–210.

[98] Shreekant (Ticky) Thakkar et Tom Huff. “The Internet Streaming SIMD Extensions”. In : *Intel Technology Journal* 3.2 (1999).

[99] Thinking Machines Corporation. *CM Fortran Reference Manual, Version 1.0*. Cambridge, Massachusetts, fév. 1991.

[100] Martin Tillenius, Elisabeth Larsson, Rosa M. Badia et Xavier Martorell. “Resource-Aware Task Scheduling”. In : *ACM Trans. Embed. Comput. Syst.* 14.1 (jan. 2015), 5 :1–5 :25.

[101] Alexander Tiskin. “BSP (Bulk Synchronous Parallelism)”. In : *Encyclopedia of Parallel Computing*. Sous la dir. de David Padua. Boston, MA : Springer US, 2011, p. 192–199.

[102] top500. URL : https://www.top500.org/ (visité le 02/02/2017).

[103] Rémi Triolet. “Contribution a la parallélisation automatique de programmes Fortran comportant des appels de procédure”. Theses. Ecole Nationale Supérieure des Mines de Paris, 1984.

[104] Chau-Wen Tseng. “An optimizing Fortran D compiler for MIMD distributed-memory machines”. Theses. Rice University, 1993.

[105] UPC. *Unified Parallel C*. URL : https://upc-lang.org/ (visité le 02/02/2017).

[106] UPC Consortium. *UPC Specifications, Version 1.3*. Nov. 2013.

[107] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In : *Commun. ACM* 33.8 (août 1990), p. 103–111.

[108] Michael E. Wolf et Monica S. Lam. "A Data Locality Optimizing Algorithm". In : *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada : ACM, 1991, p. 30–44.

[109] Michael Wolfe. "More Iteration Space Tiling". In : *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. Supercomputing '89. Reno, Nevada, USA : ACM, 1989, p. 655–664.

[110] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. 1st. Redwood City, CA, USA : Benjamin/Cummings, 1996.

[111] H.P. Zima et B.M. Chapman. "Compiling for distributed-memory systems". In : *Proceedings of the IEEE* 81.2 (fév. 1993), p. 264–287.

Résumé

Les grilles de calculs sont des architectures distribuées couramment utilisées pour l'exécution de programmes scientifiques ou de simulation. Les programmeurs doivent ainsi acquérir de nouvelles compétences pour pouvoir tirer partie au mieux de toutes les ressources offertes. Ils doivent apprendre à écrire un code parallèle, et, éventuellement, à gérer une mémoire distribuée.
L'ambition de cette thèse est de proposer une chaîne de compilation permettant de générer automatiquement un code parallèle distribué en tâches à partir d'un code séquentiel. Pour cela, le compilateur source-à-source PIPS est utilisé. Notre approche a deux atouts majeurs : 1) une succession de transformations simples et modulaires est appliquée, permettant à l'utilisateur de comprendre les différentes transformations appliquées, de les modifier, de les réutiliser dans d'autres contextes, et d'en ajouter de nouvelles; 2) une preuve de correction de chacune des transformations est donnée, permettant de garantir que le code généré est équivalent au code initial.
Cette génération automatique de code parallèle distribué de tâches offre également une interface de programmation simple pour les utilisateurs. Une version parallèle du code est automatiquement générée à partir d'un code séquentiel annoté.
Les expériences effectuées sur deux machines parallèles, sur des noyaux de Polybench, montrent une accélération moyenne linéaire voire super-linéaire sur des exemples de petites tailles et une accélération moyenne égale à la moitié du nombre de processus sur des exemples de grandes tailles.

Mots Clés

Compilation, Architecture parallèle distribuée, Parallélisation de tâche, Génération automatique de code, Vérification de code

Abstract

Scientific and simulation programs often use clusters for their execution. Programmers need new programming skills to fully take advantage of all the available resources. They have to learn how to write parallel codes, and how to manage the potentially distributed memory.
This thesis aims at generating automatically a distributed parallel code for task parallelisation from a sequential code. A source-to-source compiler, PIPS, is used to achieve this goal. Our approach has two main advantages: 1) a chain of simple and modular transformations to apply, thus visible and intelligible by the users, editable and reusable, and that make new optimisations possible; 2) a proof of correctness of the parallelisation process is made, allowing to insure that the generated code is correct and has the same result as the sequential one.
This automatic generation of distributed-task program for distributed-memory machines provide a simple programming interface for the users to write a task oriented code. A parallel code can thus automatically be generated with our compilation process.
The experimental results obtained on two parallel machines, using Polybench kernels, show a linear to super-linear average speedup on small data sizes. For large ones, average speedup is equal to half the number of processes.

Keywords

Compilation, Distribued parallel architecture, Task parallelisation, Automatic code generation, Code verification