Rapport de 1ère année : Compilation et optimisation de langages parallèles

Lossing Nelson
Maître de thèse : Ancourt Corinne
Directeur de thèse : Irigoin François
Centre de Recherche en Informatique, Mines ParisTech

Pour le 26 Mai 2014

1 Introduction

1.1 Contexte

En 1965, la loi de Moore [10] a été énoncée pour la première fois. Puis elle a été modifiée en 1975 pour dire que le nombre de transistors dans un microprocesseur doublerait tous les deux ans. Ce postulat s'est révélé vrai jusque dans les années 2002 où cette augmentation s'est ralentie. Cette augmentation exponentielle du nombre de transistors au sein d'un même processeur permettait à une application, sans modification de celle-ci, d'augmenter sa vitesse d'exécution. Ainsi pour exécuter une application deux fois plus vite, il suffisait en quelque sorte d'acheter un nouvel ordinateur deux ans plus tard et il n'était pas nécessaire de réfléchir à une réimplémentation de l'application. Mais cela est de moins en moins possible depuis les années 2002. En effet, le nombre de transistors sur un processeur tend à stagner et par conséquence la fréquence d'horloge et donc le temps d'exécution également.

Pour palier cette stagnation, différentes architectures parallèles se sont développées. Ces architectures ont pour but d'exécuter une application en parallèle sur plusieurs processeurs pour augmenter leur vitesse d'exécution. On peut classer ces architectures en trois grandes catégories.

La première de ces architectures est une architecture parallèle à mémoire partagée. Elle consiste à mettre plusieurs micro-processeurs ou cœurs sur un même CPU, le plus souvent des puissances de deux. Ainsi, tous les cœurs peuvent effectuer des calculs en parallèle et travaillent sur une mémoire unique. La plupart des ordinateurs modernes ont cette architecture.

La deuxième possibilité est d'avoir une architecture parallèle à mémoire distribuée. Dans cette architecture, on a plusieurs CPU qui ont chacun une mémoire propre. Ainsi pour travailler sur une donnée, un CPU doit avoir cette donnée présente dans sa mémoire. Des communications entre les différents CPU doivent s'effectuer pour pouvoir échanger les valeurs des différentes données sur lesquelles ils travaillent. On peut se représenter cette architecture en imaginant plusieurs ordinateurs en réseaux.

La dernière catégorie d'architecture possible consiste à avoir une architecture dédiée. Cela correspond à avoir une architecture propre à l'application que l'on souhaite réaliser. On peut par exemple penser à l'utilisation d'un FPGA ou d'un GPU.

Ces différentes architectures peuvent bien évidemment se compléter. Par exemple, on parle d'architecture hybride lorsque l'on combine une architecture à mémoire partagée et une architecture à mémoire distribuée, *ie.* tous les cœurs de plusieurs ordinateurs en réseaux. De même, on parle d'architecture hétérogène lorsque l'on utilise une architecture à mémoire partagée et une architecture dédiée, *ie.* le CPU et le GPU d'une ordinateur.

Mais ces différentes architectures impliquent le développement de nouvelles façons de programmer et de nouveaux langages. Pour pouvoir les utiliser, elles nécessitent également de réimplémenter tout ou en partie les applications que l'on souhaite accélérer.

1.2 Motivations et sujet de thèse

Il est souvent difficile de passer d'un code séquentiel à un code parallèle pour un non-expert de la parallélisation. De même, il est souvent plus difficile d'avoir un raisonnement "parallèle", qu'un raisonnement "séquentiel" en terme de programmation.

Ainsi, le but de cette thèse est de pouvoir générer automatiquement une nouvelle application parallèle à partir d'une application séquentielle. On souhaite bien évidemment que le résultat de l'exécution parallèle soit le même que celui qui aurait été renvoyé par sa version séquentielle.

Comme dit précédemment, plusieurs architectures parallèles existent. Dans un premier temps, le choix a été fait de se concentrer sur les architectures distribuées. L'une des raisons est

qu'il est souvent plus difficile de bien gérer toutes les communications. En effet, en `C` et avec la librairie OpenMP[11], il peut être suffisant de juste mettre quelques directives "PRAGMA" et de savoir détecter quelles sont les variables à partager pour réaliser une parallélisation simple en mémoire partagée. Alors qu'en mémoire distribuée, il est nécessaire de s'assurer que tous les processeurs ont les mêmes informations pour chacune des variables à traiter.

De plus, pour le moment, on considère que l'on dispose d'un placement, ou *mapping*, déjà donné. Ainsi, on se concentre principalement sur la génération automatique d'un code distribué à partir d'un code séquentiel. On veut assurer qu'à partir d'un code séquentiel et un placement donné, on obtienne le meilleur code distribué possible.

Enfin, la parallélisation que l'on souhaite faire est de la parallélisation de tâches. Contrairement à la parallélisation de boucles qui implique que tous les processeurs exécuteront le même code, la parallélisation de tâches permet à plusieurs processeurs d'exécuter des codes différentes, de réaliser différentes tâches simultanément. De se fait, on ne souhaite pas déporter le même code sur tous les processeurs mais du code spécifique.

1.3 Plan

Ce rapport se décompose en sept parties.

L'introduction *Sec.* 1 a présenté le contexte et les motivations de mon sujet de thèse.

La deuxième partie *Sec.* 2 présente un état de l'art des travaux qui existent et une bibliographie qui pourra m'être utile.

La troisième partie *Sec.* 3 présente le framework PIPS qui sera utilisé pour le développement des applications au cours de cette thèse.

La quatrième partie *Sec.* 4 définit le langage de base que nous étudions dans un premier temps. Il définit également différentes propriétés que l'on utilise pour valider nos transformations de programme par la suite.

La cinquième partie *Sec.* 5 présente le travail et les réflexions concernant notre problème de génération automatique d'un code distribué.

L'avant dernière partie *Sec.* 6 présente mon portefeuille de compétences.

Enfin la dernière partie *Sec.* 7 conclut ce rapport.

2 État de l'art et bibliographie

Je présente dans cette partie quelques articles, thèses ou livres qui sont en lien avec mon sujet de thèse et vont m'être utiles.

2.1 Articles et thèses

Les articles [8, 5, 9] présentent les travaux réalisés par l'institut TELECOM SudParis sur le projet STEP, *Système de Transformation pour l'Exécution Parallèle*. Comme son nom l'indique, STEP a pour but de réaliser des transformations en vue d'une exécution parallèle du programme. Il permet également à partir d'un code s'exécutant en mémoire partagée de générer un code pour de la mémoire distribuée. Ainsi, à partir d'un code en OpenMP, il génère un code MPI. On peut distinguer trois étapes pour cette transformation du code. La première consiste à extraire la partie de code que l'on souhaite exécuter en mémoire distribuée. La deuxième étape analyse les dépendances qui sont présentes. La dernière étape transforme le code en respectant les dépendances obtenues par l'analyse précédente. Les transformations de code sont néanmoins limitées à la parallélisation de boucle.

Les articles [7, 6] réalisées par l'université de Purdue aux USA sont assez similaires à ceux du projet STEP. Ils traduisent du code OpenMP en code MPI, et se limitent à la version 2.0 de OpenMP, c'est-à-dire que seule la parallélisation de boucles est considérée et pas la parallélisation de tâche qui apparait seulement dans la version 3.0 de OpenMP. Après lecture de ces articles, je n'ai pas pu constaté de différences majeures par rapport à STEP.

La thèse de Jean-Yves Vet [13] à laquelle j'ai pu assister à la soutenance, présente comment la parallélisation pour une architecture hétérogène peut être effectuée. Ainsi, son objectif est de paralléliser un code à la fois sur tous les cœurs d'un CPU, c'est-à-dire en mémoire partagée, et sur le GPU de la machine exécutant le programme. Il détermine quelle doit être la répartition de charge entre les cœurs et le GPU pour que le programme s'exécute de façon optimale. Cette thèse a été réalisée en collaboration entre le CEA et l'université Pierre et Marie Curie.

La thèse de Béatrice Creusillet [2] définit et décrit l'analyse des régions de tableaux lues ou écrites dans une partie du programme. Cette thèse a été réalisée au CRI et est implémentée dans *PIPS*. Ces régions me permettront de savoir quelles sont les cases d'un tableau qui doivent être communiquées entre des processeurs différents.

Enfin, la thèse de Dounia Khaldi [4] étudie la parallélisation de tâches. Elle y présente une représentation interne générique pour un langage parallèle. Elle étudie également un placement possible des différentes instructions du programme sur un nombre donné de processeurs. Enfin, elle essaie de générer le code parallèle correspondant aussi bien pour de la mémoire partagée que distribuée. Cette thèse a été réalisée au CRI et le résultat de celle-ci est implémenté dans *PIPS*. Pour mon étude, j'utiliserai le placement qu'elle a trouvé pour effectuer la génération de code parallèle distribué.

2.2 Livres

The denotational description of programming languages : an introduction [3] introduit les connaissances permettant de mettre en place une syntaxe et une sémantique dénotationnelle.

Compilateurs principes, techniques et outils [1], ou encore appelé "*Dragon Book*", est le livre de référence concernant le fonctionnement d'un compilateur.

Supercompilers for parallel and vector computers [14] recense plusieurs analyses et techniques de transformation pour faire de la parallélisation.

3 Le framework PIPS

Cette partie 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/` [12].

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* place l'ensemble des fichiers à traiter dans un *workspace*. Ce dernier stocke également les ressources nécessaires et produites au cours des traitements.

Les règles et propriétés L'exécution d'une analyse ou d'une transformation s'effectue au moyen de règles ou d'ensembles de règles, appelée passe ou phase. Par abus, on utilisera indifféremment ces trois termes.

Plusieurs types de règles existent dans *PIPS*, elles sont principalement réunies en deux groupes : celles qui réalisent une analyse de code et génèrent des ressources dans le *workspace* ; et celles qui réalisent une transformation directement dans le code. Un troisième groupe existe cependant, les *pretty-printers*. Elles permettent de rajouter en commentaires les résultats obtenus par une analyse dans le code analysé.

Ces passes sont paramétrables au moyen d'options, appelées propriétés.

Parmi les passes d'analyses, on peut en citer certaines qui me seront utiles :

`sequence_dependence_graph` permet de réaliser un placement au niveau des instructions. Ainsi, cette passe permet de décrire sur quel processeur les instructions doivent être exécutées. Elle a été réalisée et décrite dans la thèse de Dounia Khaldi [4].

`proper_effects` permet de calculer les effets des variables scalaires au niveau de chaque instruction, c'est-à-dire de donner les variables scalaires qui sont lues ou écrites par une instruction. `cumulated_effects` calcul les effets cumulés au niveau d'une séquence d'instructions, par exemple pour un bloc de test ou de boucle.

`may_regions` permet de calculer les effets pour des régions de tableaux au niveau de chaque instruction. Ainsi, on peut connaitre les régions de tableaux qui ont été lues ou écrites par une instruction. Elle a été réalisée et décrite dans la thèse de Béatrice Creusilet [2].

`preconditions_inter_full` permet de calculer les préconditions pour chaque instruction d'un programme.

De même, on peut citer quelques transformations de code qui me seront utiles :

`simplify_control` permet de faire de la simplification de code à partir d'un graphe de flot de contrôle. Par exemple, lors d'un test, si le résultat de la condition est toujours le même alors `simplify_control` supprimera la branche du test qui n'est jamais exécutée.

`dead_code_elimination` permet de supprimer du code mort. Un code est considéré comme *mort* s'il n'est pas utile pour la suite de l'exécution. Par exemple, si on effectue une affectation sur une variable i, mais que cette variable n'est jamais utilisée dans la suite du programme, alors l'affection de i est considérée comme étant du code mort, et on peut supprimer cette affectation.

4 Définition d'un langage de base

Cette section définit le langage de base que l'on souhaite analyser et sur lequel on pourra réaliser nos optimisations.

Dans un premier temps, en *Sec.* 4.1, on donne la syntaxe de notre langage.

Puis, on définit différentes opérations en *Sec.* 4.2 qui pourront être faites sur ce langage. Ces opérations servent à définir les propriétés que l'on souhaite assurer lors des transformations et présentées dans la partie suivante *Sec.* 5, en particulier les définitions d'équivalence en *Sec.* 4.2.3.

4.1 Syntaxe

La grammaire de notre langage de base est le suivant :

$$
\begin{array}{lcll}
\langle \textit{function} \rangle & ::= & \langle \textit{declaration} \rangle \text{ (} \langle \textit{declaration} \rangle^* \text{) \{ } \langle \textit{statements} \rangle \text{ ; [return id ;] \}} & \\
\langle \textit{statements} \rangle & ::= & \varnothing \mid \langle \textit{statement} \rangle \text{ ; } \langle \textit{statements} \rangle & \\
\langle \textit{statement} \rangle & ::= & \langle \textit{declaration} \rangle & \\
& \mid & \langle \textit{affectation} \rangle & \\
& \mid & \text{if } \langle \textit{expression} \rangle \text{ then } \langle \textit{statements} \rangle \text{ [else } \langle \textit{statements} \rangle \text{]} & \\
& \mid & \text{while } \langle \textit{expression} \rangle \text{ do } \langle \textit{statements} \rangle & \\
& \mid & \text{for (} \langle \textit{affectation} \rangle \text{ ; } \langle \textit{expression} \rangle \text{ ; } \langle \textit{affectation} \rangle \text{) } \langle \textit{statements} \rangle & \\
& \mid & (\langle \textit{statements} \rangle, \text{ p}) & \text{p} \in \mathbb{N} \\
& \mid & \langle \textit{statements} \rangle & \\
\langle \textit{declaration} \rangle & ::= & \langle \textit{type} \rangle \text{ id} & \\
\langle \textit{affectation} \rangle & ::= & \langle \textit{variable} \rangle \leftarrow \langle \textit{expression} \rangle & \\
\langle \textit{expression} \rangle & ::= & \langle \textit{variable} \rangle & \\
& \mid & \text{constant} & \\
& \mid & \text{true} \mid \text{false} & \\
& \mid & \langle \textit{unary-op} \rangle \langle \textit{expression} \rangle & \\
& \mid & \langle \textit{expression} \rangle \langle \textit{binary-op} \rangle \langle \textit{expression} \rangle & \\
\langle \textit{variable} \rangle & ::= & \text{id} & \\
& \mid & \langle \textit{variable} \rangle \text{. id} & \\
& \mid & \langle \textit{variable} \rangle \text{[} \langle \textit{index} \rangle \text{]} & \\
 & & & \\
\langle \textit{int-type} \rangle & ::= & \text{(unsigned} \mid \text{signed) (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 & \\
& \mid & \langle \textit{type} \rangle \text{[n]} & \text{n} \in \mathbb{N} \\
& \mid & \text{struct \{ id}_1 : \langle \textit{type} \rangle, \ldots, \text{id}_n : \langle \textit{type} \rangle \text{\}} & \\
\langle \textit{unary-op} \rangle & ::= & \text{-} \mid \sim \mid (\langle \textit{int-type} \rangle) & \\
\langle \textit{binary-op} \rangle & ::= & + \mid \text{-} \mid * \mid / \mid \% \mid \& \mid \mid \mid \hat{} \mid \text{» } \mid \text{ «} &
\end{array}
$$

Remarque : les $\langle \textit{statement} \rangle$ de $\langle \textit{statements} \rangle$ d'une $\langle \textit{function} \rangle$ peuvent uniquement être $\langle \textit{declaration} \rangle$ ou $(\langle \textit{statements} \rangle, \text{p})$.

Cette syntaxe permet de travailler au niveau des fonctions et d'avoir un placement au niveau de toutes instructions de premier niveau d'une fonction.

Les instructions que l'on autorise, sont les déclarations, les affections, les tests de type if, les boucles de type while ou for.

Les types utilisés sont les types de bases du langage `C` à l'exception des pointeurs qui ne sont pas présents.

Par la suite, on utilisera souvent le terme de variable pour parler de ce qui est représenté par les identifiants dans notre syntaxe.

4.2 Opérations sur les éléments du langage

4.2.1 Accesseurs

body La fonction *body* retourne l'ensemble des instructions de premier niveau d'une fonction.

$$body(func) = stats \text{ such that } func = f([a1, a2, \dots]) \{stats; [return\ r;]\}$$

subStatements La fonction *subStatements* permet d'obtenir l'ensemble de toutes les instructions se trouvant récursivement dans une instruction (ou une suite d'instructions).

$$\begin{aligned}
subStatements(stat) = \ & \text{if } stat = while\ expression\ do\ statements \\
& \quad \{stat\} \cup subStatements(statements) \\
& \text{elseif } stat = for(affectation1; expression; affectation2) statements \\
& \quad \{stat\} \cup subStatements(statements) \\
& \text{elseif } stat = if\ expression\ then\ statements \\
& \quad \{stat\} \cup subStatements(statements) \\
& \text{elseif } stat = if\ expression\ then\ statements1\ else\ statements2 \\
& \quad \{stat\} \cup subStatements(statements1) \cup subStatements(statements2) \\
& \text{elseif } stat = (statements, p) \\
& \quad \{stat\} \cup subStatements(statements) \\
& \text{elseif } stat = statement; statements \\
& \quad \{stat\} \cup subStatements(statement) \cup subStatements(statements) \\
& \text{else} \\
& \quad \{stat\} \\
subStatements(stats) = \ & \text{if } stats = statement; statements \\
& \quad \{stats\} \cup subStatements(statement) \cup subStatements(statements)
\end{aligned}$$

bodyAll La fonction *bodyAll* retourne l'ensemble de toutes les instructions d'une fonction.

$$bodyAll(func) = subStatements(body(func))$$

decl L'ensemble des variables déclarées au premier niveau d'une fonction est obtenu grâce à la fonction *decl*.

$$decl(func) = \{id | \exists s \in body(func), s = type\ id\}$$

declAll Similairement l'ensemble de toutes variables déclarées au sein d'une fonction est obtenu grâce à la fonction *declAll*.

$$declAll(func) = \{id | \exists s \in bodyAll(func), s = type\ id\}$$

4.2.2 Opérateurs

GenId La fonction GenId permet de générer un nouvel identifiant *frais* dans une fonction donnée à partir d'un identifiant et d'un numéro de processeur. Elle ne s'applique que sur des identifiants déclarés au premier niveau d'une fonction. Elle a les propriétés suivantes :

— Identifiant différent

$$\forall f \in \langle function \rangle, \forall p \in \mathbb{N}, \forall id \in decl(f), GenId(f, p, id) \neq id$$

— Unicité/Injectivité

$$\forall f \in \langle function \rangle, \forall p_1, p_2 \in \mathbb{N}, \forall id_1, id_2 \in decl(f),$$
$$id_1 \neq id_2 \Rightarrow GenId(f, p1, id_1) \neq GenId(f, p_2, id_2)$$

— Absence de conflit de nom (y compris avec des déclarations profondes)

$$\forall f \in \langle function \rangle, \forall p \in \mathbb{N}, \forall id_1 \in decl(f), \forall id_2 \in declAll(f), GenId(f, p, id_1) \neq id_2$$

Si l'on considère que l'on peut avoir plusieurs fois le même identifiant avec des portées différentes dans le programme alors $id2 \in decl(f)$ suffit

$$\forall f \in \langle function \rangle, \forall p \in \mathbb{N}, \forall id_1, id_2 \in decl(f), GenId(f, p, id_1) \neq id_2$$

eval La fonction *eval* requiert un état mémoire σ et une fonction f. Elle retourne un nouvel état mémoire σ' après exécution de la fonction f. σ et σ' appartiennent à Σ, l'ensemble des états mémoires.

$$eval(f, \sigma) = \sigma'$$

4.2.3 Prédicats

correct On définit le prédicat *correct* pour une fonction ; il est vérifié si toute variable utilisée par la fonction a été déclarée et est bien typée. Il assure également que l'évaluation de la fonction se termine.

équivalent On définit plusieurs fonctions d'équivalence : $equiv_1$, $equiv_2$, $equiv_3$.

La première équivalence $equiv_1$ est une équivalence forte. Elle porte sur l'état mémoire après l'exécution de fonctions, f_1 et f_2. Cette propriété garantit que deux fonctions sont équivalentes si et seulement si pour tout état initial, l'exécution de ces fonctions conduit au même état final. $equiv_1$ est définie par :

$$equiv_1(f_1, f_2) \equiv \begin{cases} correct(f_1) \\ correct(f_2) \\ \forall \sigma, eval(f_1, \sigma) = eval(f_2, \sigma) \end{cases}$$

Elle permet de valider l'application de certaines optimisations telles que la réduction de force (*strength reduction*), ou la propagation de constante.

```
int main() {
  int i, r;
  i = 0;
  r = i;
  return r;
}
```

```
int main() {
  int i, r;
  i = 0;
  r = 0;
  return 0;
}
```

FIGURE 4.1 – Exemple de deux fonctions équivalentes par $equiv_1$

L'équivalence $equiv_1$ est souvent plus forte que nécessaire. En effet, les variables utiles, pour lesquelles nous cherchons à montrer des équivalences, ne sont pas forcément toutes les variables du programme. C'est le cas typiquement de la valeur de la variable de retour ou celles affichées dans une sortie standard. Ainsi, on définit une équivalence $equiv_2$ restreinte à un ensemble de variables, dans notre syntaxe des identifiants, pour lesquelles on souhaite vérifier l'état mémoire final. Notre syntaxe n'offrant pas la possibilité de gérer des entrées/sorties, on doit définir l'ensemble des identifiants considérés par une fonction *observable* avant d'appliquer la fonction d'équivalence $equiv_2$. Notre deuxième fonction d'équivalence $equiv_2$ est définie par :

$$equiv_2(f_1, f_2) \equiv \begin{cases} correct(f_1) \\ correct(f_2) \\ \forall \sigma, \forall id \in observable(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(id) \\ \forall \sigma, \forall id \in observable(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(id) \end{cases}$$

Pour notre étude, les identifiants à vérifier sont tous les identifiants déclarés au premier niveau de nos fonctions, c'est-à-dire que la fonction *observable* correspondra à la fonction *decl*. Cette nouvelle équivalence permet par exemple de faire de l'élimination de code mort (*dead code elimination*).

```
int main() {
  int i, r;
  i = 0;
  r = 0;
  return r;
}
```

```
int main() {
  int i, r;
  r = 0;
  return r;
}
```

FIGURE 4.2 – Exemple de deux fonctions équivalentes par $equiv_2$

On peut encore relâcher la fonction d'équivalence $equiv_2$ en n'imposant pas que les variables soient strictement les mêmes. En effet, l'identifiant de la variable en tant que tel n'est pas très utile en soi, et seule sa valeur a une importance. Ainsi, on définit $equiv_3$ qui à la fois restreint l'ensemble des identifiants considérés, mais également autorise un renommage des variables à considérer. Comme pour $equiv_2$, on doit avoir défini la fonction *observable* qui renvoie un ensemble d'identifiants et *permut* qui prend au moins un identifiant en argument et renvoie un identifiant. Si on considère *permut* comme étant la fonction identité, alors $equiv_3$ correspond à $equiv_2$. La dernière fonction d'équivalence $equiv_3$ est la suivante :

$$equiv_3(f_1, f_2) \equiv \begin{cases} correct(f_1) \\ correct(f_2) \\ \forall \sigma, \forall id \in observable(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(permut(id)) \\ \forall \sigma, \forall id \in observable(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(permut(id)) \end{cases}$$

Pour notre étude, comme pour $equiv_2$, *observable* correspondra à la fonction *decl*, et *permut* correspondra à la fonction GenId et sa réciproque.

```
int main() {
  int i, r1;
  i = 0;
  r1 = 0;
  return r1;
}
```

```
int main() {
  int i, r2;
  r2 = 0;
  return r2;
}
```

FIGURE 4.3 – Exemple de deux fonctions équivalentes par $equiv_3$

5 Algorithmes

Cette section présente le travail effectué au cours des premiers mois de ma thèse et une partie des différentes réflexions qui ont été menées. On reprend le langage défini dans la partie précédente *Sec.* 4 ainsi que les différents opérateurs qui y ont été définis.

Le travail qui est décrit a pour but de générer automatiquement du code dans un langage parallèle distribué à partir d'un code séquentiel.

Le principe de cette génération de code automatique est décrit au moyen de différents algorithmes. Ces algorithmes sont décrits de la façon suivante :

Input les différents paramètres nécessaires pour exécuter l'algorithme.

Output ce que renvoie l'algorithme.

Ensure ce que l'on garantit après l'exécution de l'algorithme.

Execution le pseudo-code de l'algorithme.

En *Sec.* 5.1, le principe de notre algorithme général est décrit.

Puis en *Sec.* 5.2, on décrit la première partie de notre algorithme.

Ensuite en *Sec.* 5.3 et en *Sec.* 5.4, on décrit succinctement le principe et la réflexion faites sur les deuxième et troisième parties de notre algorithme.

5.1 Algorithme général

Algorithme 1: Transformation d'un code séquentiel en un code distribué(⟨*function*⟩ f, int NbP)

Input: fonction (séquentielle) correct f
Output: ensemble de fonctions devant s'exécuter sur des processeurs différents {newFs }

```
1 lId = decl(f);
2 f1 = Generate Copy for Communication(f, lId, NbP);
  // equiv3 (f1, f)
3 f2 = f1;
  // equiv1 (f2, f1)
  /* block of statements on the same processor (group of sequential
     statements that are executed on the same processor)                  */
4 foreach (statsOnP, p) in f1 do
5 |   temp = Reduce Copy(f2, statsOnP, p, lId, NbP);
  |   // equiv1 (temp, f2)
6 |   f2 = Relocate Copy(temp, statsOnP, p, lId, NbP);
  |   // equiv1 (f2, temp)
7 end
  // equiv1 (f2, f1)
8 newF = Translate Copy into Communication(f2, lId, NbP) ;
  // equiv1 (newF, f2)
9 {newFs } = Generate Function for each Processor(newF, lId, NbP);
```

Notre algorithme *Algo.* 1 a pour but de distribuer l'exécution d'un code sur plusieurs processeurs. Pour chaque création de nouvelle fonction (func1, func2...), il est indiqué quelle équivalence on souhaite obtenir. Notre algorithme renvoie à la fin NbP+1 nouvelles fonctions, NbP fonctions pour les NbP processeurs, et 1 fonction chargée de les lancer.

L'algorithme se décompose en trois grandes étapes.

La première étape *Sec.* 5.2 consiste à préparer la fonction pour l'exécution sur les différents processeurs. Pour cela, on réplique nos variables autant de fois que le nombre de processeurs sur lesquels on veut effectuer la distribution. Chaque variable répliquée représente une variable pour un processeur particulier. Des copies entre les différentes variables répliquées sont faites pour maintenir la cohérence.

La seconde étape consiste à minimiser le nombre de copies à faire localement, pour un processeur donné. Minimiser le nombre de copies locales implique la minimisation du nombre de communications qui devront êtres faites. Lors de cette étape, on regroupe également les copies faites par un processeur sur une suite d'instructions concomitantes. Ce regroupement prépare les optimisations futures.

La dernière étape consiste à traduire les copies entre variables appartenant à des processeurs différents en communications réelles. Grâce aux regroupements de copies effectués à l'étape précédente, on pourra souvent se limiter à l'étude de ces blocs pour cette étape. On vérifie si les copies sont réellement nécessaires avant de les traduire en communications. Par exemple, si un processeur n'utilise pas une variable, il n'y a pas besoin de faire de copie pour ce processeur. De même, s'il y a 3 processeurs et si le processeur 1 écrit une variable v, et que le processeur 2 refait une écriture de v, alors il n'y a pas besoin de faire de copie de v entre les processeurs 1 et 3. On communiquera autant que possible de façon atomique, par exemple, envoyer tout le tableau d'un coup au lieu de transférer les éléments du tableau un par un. Enfin, on génère une fonction pour chaque processeur, qui ne contient que les instructions qui lui sont propres.

Cette méthodologie pourra s'appliquer de façon récursive sur les nouvelles fonctions générées.

Pour exécuter notre algorithme, on suppose qu'on a les prérequis suivants :

— Une fonction (séquentielle) correcte.
— Pas d'initialisations dans les déclarations.
— Nombre de processeurs numériquement connu.
— Un mapping déjà défini, ie on sait sur quel processeur on veut exécuter chacune des instructions du corps de la fonction.
— Pas de variables globales.
— Pas d'écriture sur les arguments.
— Absence d'aliasing.
— Pas de pointeurs.

Certains de ces prérequis sont déjà imposés par la syntaxe que l'on s'est définie. Par exemple, pas de pointeurs ou pas d'initialisations dans les déclarations.

D'autres sont présents pour le moment ou pour la description de l'algorithme, mais pourront être supprimés par la suite. Par exemple, l'absence de pointeurs ou de variables globales.

5.2 Génération des copies pour communications futures

Algorithme 2: Generate Copy for Communication(⟨*function*⟩ f, set lId, int nbP)

Input:
- — fonction séquentielle initiale f
- — ensemble d'identifiants (variables) qui sont déclarés au premier niveau de la fonction lId
- — nombre de processeurs nbP

Output: fonction séquentielle sans les variables d'origine mais des variables répliquées newf

Ensure: Soit observable = lId et $\forall$ p $\in [0; \mathsf{nbP}[$, permut(id) = rename(f, p, id), $equiv_3$ (newf, f)
$\forall$ id $\in$ lId $\forall$ p $\in [0; \mathsf{nbP}[$, GenId (f, p, id) $\in$ decl(newf)
$\forall$ id $\in$ lId $\forall$ p $\in [0; \mathsf{nbP}[$, GenId (f, p, id) $\notin$ decl(f)
$\forall$ id $\in$ lId $\forall$ p_1, $\mathsf{p}_2 \in [0; \mathsf{nbP}[$, $\forall \sigma \in \Sigma$
$eval(\mathsf{newf}, \sigma)(GenId(\mathsf{f}, \mathsf{p}_1, \mathsf{id})) = eval(\mathsf{newf}, \sigma)(GenId(\mathsf{f}, \mathsf{p}_2, \mathsf{id}))$

1 f_1 = Variable Replication(f, f, lId, nbP);
// $equiv_2$ (f_1, f)
2 f_2 = Update Copy Variables(f, f_1, lId, nbP);
// $equiv_2$ (f_2, f) $\wedge$ $equiv_3$ (f_2, f)
3 newf = Eliminate Original Variables(f, f_2, lId);
// $equiv_3$ (newf, f)

Notre première étape *Algo.* 2 traite la fonction entière. Elle retourne une fonction avec de nouvelles variables qui correspondront aux variables locales des processeurs. Elle consiste donc à mettre en place les variables qui seront utilisées par chacun des processeurs. En quelque sorte, on simule l'état mémoire de chaque processeur et on s'assure que tous ces états mémoire sont identiques. Les propriétés que l'on souhaite assurer après l'exécution de cette étape sont présentées dans la partie *Ensure*.

Cette première étape *Algo.* 2 a trois passes.

La première passe est décrite en *Sec.* 5.2.1. Elle consiste à générer les variables qui seront utilisées par les différents processeurs.

Puis, *Sec.* 5.2.2 assure que toutes les variables locales générées pour chaque processeur ont la même valeur.

Enfin, *Sec.* 5.2.3 consiste à supprimer l'utilisation des variables d'origine qui ont été répliquées sur chaque processeur. Malgré cette suppression, on assure que toutes les variables répliquées ont la même valeur et que cette valeur est bien la bonne.

5.2.1 Répliquer la déclaration des variables

Algorithme 3: Variable Replication(⟨*function*⟩ f_0, ⟨*function*⟩ f, set lId, int nbP)

Input:
- fonction séquentielle initiale f_0
- fonction séquentielle f
- ensemble d'identifiants lId
- nombre de processeurs nbP

Output: fonction séquentielle avec de nouvelles variables déclarées newFunc

Ensure: $\forall$ id $\in$ lId $\forall$ p $\in [0; nbP[$, GenId (f, p, id) $\in$ decl(newFunc)
Soit observable = lId, $equiv_2$ (newFunc, f)
$\forall$ id $\in$ lId $\forall$ p $\in [0; nbP[$, GenId (f_0, p, id) $\in$ decl(newFunc)
Soit observable = lId, $equiv_2$ (newFunc, f_0)

```
1  newFunc.arg = f.arg;
2  newFunc.body = ∅
3  foreach stat in body(f) do
4  |  newFunc.body += stat ;
5  |  if isDeclaration(stat) then
6  |  |  foreach p in [0; nbP[ do
7  |  |  |  newFunc.body += makeDeclaration(stat, p);
8  |  |  end
9  |  end
10 end
```

Cette première passe *Algo.* 3 consiste à générer de nouvelles variables *fraiches*. Chaque variable déclarée au premier niveau de la fonction en entrée f sera répliquée NbP fois, le nombre de processeurs. Chacune de ces variables répliquées correspond à la variable qui sera utilisée pour un processeur donné. Cette passe a donc besoin en entrée d'une fonction à traiter f et du nombre de processeurs disponibles NbP.

```
int i;
```

(a) avant

```
int i;
int i_0;
int i_1;
```

(b) après

```
int i;
int j;
int a[10];
j=0;
for (i=0; i<10; i++) {
  j++;
  a[i]=j;
}
```

(c) avant

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;
for (i=0; i<10; i++) {
  j++;
  a[i]=j;
}
```

(d) après

FIGURE 5.1 – Exemple avec 2 processeurs avant et après l'application de Variable Replication

5.2.2 Effectuer des copies après chaque instruction d'écriture

Algorithme 4: Update Copy Variables(⟨*function*⟩ f_0, ⟨*function*⟩ f, set lId, int nbP)

Input:
- fonction séquentielle initiale f_0
- fonction séquentielle f
- ensemble d'identifiants lId
- nombre de processeurs nbP

Output: fonction séquentielle avec toutes les variables répliquées ayant la même valeur newFunc

Ensure: Soit observable = lId, $equiv_2$ (newFunc, f)
Soit observable = lId et $\forall$ p $\in [0; \text{nbP}[$, permut(id) = GenId (f_0, p, id), $equiv_3$ (newFunc, f)

Cette deuxième passe *Algo.* 4 effectue, après chaque écriture de variable déclarée au premier niveau de la fonction, une copie permettant de préserver la cohérence des données. La détection des variables écrites ou lues est obtenue grâce au calcul des effets dans *PIPS*.

Le choix a été fait de le faire au niveau de chaque instruction. On utilisera donc les effets propres de chaque instruction. Ainsi, cette passe devra traiter récursivement toutes les sous-instructions d'une instruction donnée. De ce fait, pour la spécification, on se limite à donner les propriétés que l'on souhaite garantir pour cette passe. En effet, la détailler reviendrait à en donner une implémentation.

Une autre possibilité est de ne pas descendre récursivement dans chacune des sous-instructions mais d'utiliser l'analyse des régions écrites au niveau de l'instruction parente. Dans ce cas, on peut utiliser les effets cumulés sur une instruction. Cette option permettrait de simplifier notre algorithme. Mais on perdrait en précision quand à ce qu'il est réellement nécessaire de copier, et empêcherait des optimisations futures. Cette possibilité restera éventuellement à étudier.

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;
for (i=0; i<10; i++) {
  j++;
  a[i]=j;
}
```

(a) avant

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;
j_0=j;
j_1=j;
for (i=0; i<10; i++) {
  j++;
  j_0=j;
  j_1=j;
  a[i]=j;
  a_0[i]=a[i];
  a_1[i]=a[i];
}
i_0=i;
i_1=i;
```

(b) après

FIGURE 5.2 – Exemple avec 2 processeurs avant et après l'application d'Update Copy Variables

5.2.3 Remplacer chaque accès à une variable d'origine par un accès à la copie correspondant au processeur où l'accès doit être fait

Algorithme 5: Eliminate Original Variables($\langle function \rangle$ f_0, $\langle function \rangle$ f, set lId)

Input:
- fonction séquentielle initiale f_0
- fonction séquentielle f
- ensemble d'identifiants lId

Output: fonction séquentielle sans les variables initiales $\mathsf{newFunc}$

Ensure: $\forall\ \mathsf{id} \in \mathsf{lId}$, $\mathsf{id} \notin \text{decl}(\mathsf{newFunc})$
Soit observable $= \mathsf{lId}$ et $\forall\ \mathsf{p} \in [0; \mathsf{nbP}[$, permut(id) = rename(f, p, id), $equiv_3$ (newFunc, f)

Cette troisième passe *Algo.* 5 a pour but dans un premier temps de remplacer tous les appels ou utilisations à des variables initiales par un appel ou une utilisation à une variable répliquée. La variable répliquée est celle que l'instruction devra utiliser localement. Ce renseignement est donné dans la syntaxe par ($\langle statements \rangle$, p).

Cette passe permet également de supprimer la déclaration des variables initiales déclarées au premier niveau de la fonction. En effet ces déclarations deviennent inutiles étant donné qu'elles ne sont plus utilisées.

Notons que le remplacement des variables initiales par une variable répliquée caractéristique du processeur sur lequel l'instruction s'exécute, rend notre algorithme très rapidement dépendant de la cartographie donnée.

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;
j_0=j;
j_1=j;
for (i=0; i<10; i++) {
  j++;
  j_0=j;
  j_1=j;
  a[i]=j;
  a_0[i]=a[i];
  a_1[i]=a[i];
}
i_0=i;
i_1=i;
```

(a) avant

```
int i_0;
int i_1;
int j_0;
int j_1;
int a_0[10];
int a_1[10];
j_0=0;
j_0=j_0;
j_1=j_0;
for (i_0=0; i_0<10; i_0++) {
  j_0++;
  j_0=j_0;
  j_1=j_0;
  a_0[i_0]=j_0;
  a_0[i_0]=a_0[i_0];
  a_1[i_0]=a_0[i_0];
}
i_0=i_0;
i_1=i_0;
```

(b) après

FIGURE 5.3 – Exemple avec 2 processeurs avant et après l'application d'Eliminate Original Variables

5.3 Réduction du nombre de copie et déplacement des copies au sein d'un bloc dédié à un processeur

Cette étape travaille uniquement sur des suites d'instructions concomitantes qui s'exécutent sur un même processeur.

Cette étape sera composée elle-même de deux parties.

La première consiste à effectuer des optimisations sur le bloc d'instructions à traiter. On effectuera des optimisations locales à un processeur par rapport à la fonction entière. Trois optimisations peuvent déjà être considérées. La plus simple est de supprimer les affectations qui sont des identités. En effet, la première étape de notre algorithme génère plusieurs affections de type "$i_0 = i_0$;". Une autre optimisation est de supprimer des copies redondantes de la même variable. Une dernière optimisation est possible dans le cas d'un test. Si les deux branches d'un test font une même copie alors on peut sortir cette copie en dehors du test pour l'effectuer une seule fois.

La deuxième partie de cette étape consiste à déplacer les copies. Elle a plusieurs objectifs et devra faciliter la génération de communication groupée. Lors de l'optimisation globale, on pourra faire un graphe des copies et donc des communications inter-processeurs. La réalisation de cette deuxième partie se fera de façon récursive en partant des sous-instructions les plus profondes de la partie de code étudié. Si la suite d'instructions ne contient qu'une séquence d'affections alors le déplacement pourra se faire facilement. Par contre, dans le cas de tests ou de boucles, le déplacement pourra se faire à condition de garder les résultats des conditions des tests ou boucle en mémoire, par l'utilisation de variable temporaire par exemple.

5.4 Traduction des copies en communication et génération des différentes fonctions pour chaque processeur

Cette dernière étape travaille sur la fonction dans son ensemble, et plus particulièrement sur les blocs de copies : les copies qui ont été regroupées à l'étape précédente.

Cette étape se passe en trois temps.

Le premier consiste à effectuer une optimisation globale. Par exemple, si un processeur n'utilise pas une variable, il n'y a pas besoin de faire de copie pour ce processeur. De même, pour 3 processeurs, si le processeur 1 écrit une variable v, et si le processeur 2 refait une écriture de v avant que le processeur 3 n'ai besoin d'utiliser cette variable v, alors il n'y a pas besoin de faire de copie de v entre les processeurs 1 et 3.

La deuxième étape consistera à réaliser la traduction des copies en communications réelles. Si l'on considère que c'est le nombre de communications et non la taille du message communiqué qui prend du temps, on essayera autant que possible de faire le moins de communications possibles. Par exemple, si l'on doit envoyer un tableau entier d'un processeur à un autre, alors on fera une seule communication qui enverra tout le tableau et non plusieurs communications pour envoyer les éléments du tableau un à un.

La troisième étape a pour but de générer une fonction pour chaque processeur. Chacune de ces fonctions ne possèdera que les instructions qu'elle devra exécuter. Cette étape permettra de faire d'autres optimisations de code. Par exemple, on pourra faire une exécution parallèle à mémoire partagée. Ainsi, on s'offre la possibilité de pouvoir faire de la parallélisation hybride entre mémoire distribuée et partagée.

6 Portefeuille de compétences

La figure *Fig.* 6.1 présente mon portefeuille de compétences.

Récapitulatif de participation aux Formations
Nelson LOSSING

Doctorat : Informatique temps réel, robotique et automatique - Fontainebleau
Ecole Doctorale : SMI - Sciences des Métiers de l'Ingénieur
Etablissement : MINES ParisTech
Date de la 1ère inscription en thèse : 15 octobre 2013 (1 A en 2013)
Directeur de thèse : Francois IRIGOIN
Sujet de thèse : Compilation et optimisation de langage parallèle

Formations suivies

Catégorie : Cours professionnalisants

***** LA CONDUITE DU PROJET DOCTORAL (formation réalisée par le cabinet Adoc Talent Management) (12 février 2014) 2013 - 2014 Arts et Métiers ParisTech, Campus Paris, **(Salle Des Conseils)**.
7 heures enregistrées par : SMI - Sciences des Métiers de l'Ingénieur.

***** La publication scientifique : stratégies, outils et optimisation de sa recherche (12 décembre 2013) 2013 - 2014 MINES ParisTech, 60 boulevard Saint-Michel - Bibliothèque
13 heures

***** Lecture rapide - session mars 2014 (24 mars 2014) 2013 - 2014 MINES ParisTech - 60 Boulevard Saint-Michel - V335
21 heures

***** Point de départ - session 2013-2014 2013 - 2014 MINES ParisTech - salle V.335
14 heures

Total du nombre d'heure pour la catégorie Cours professionnalisants : *55 h*

Catégorie : Journée de rentrée école doctorale

***** Journée d'accueil des doctorants (19 novembre 2013) 2013 - 2014 MINES ParisTech - 60 boulevard Saint-Michel - salle V107
mis en place par : SMI - Sciences des Métiers de l'Ingénieur.

Total participation : 55 heures / 5 modules

FIGURE 6.1 – Récapitulatif portefeuille de compétences

J'ai eu un score de 775 au TOEIC en Mai 2011.

J'ai pu participer et faire une présentation aux *7e Rencontres de la communauté française de compilation*. J'y ai présenté des travaux réalisés pour l'analyse de pointeurs dans le cadre de mon stage. Ces travaux serviront de bases aux extensions que je prévois de ma troisième année de thèse.

J'ai également pu assister à un séminaire DIGITEO présenté par Rachid Guerraoui concernant *Adversary-Oriented Computing*. Il y présentait une méthode pour réaliser un choix automatique d'algorithmes traitant de façon parallèle plusieurs requêtes reçues.

De même, j'ai pu suivre un séminaire présenté par Alain Darte sur le thème *Understanding and manipulating multi-dimensional loops : A short tour in the world of polyhedral techniques*. Il y présentait comment fonctionnent des analyses se basant sur un modèle polyédrique.

Enfin, j'ai pu assister à différents séminaires internes au CRI, où on présentait les travaux de recherche des autres membres du laboratoire.

Je compte également participer à une école d'été cette année. Ainsi, j'ai postulé à celle organisée par KTH en Suède, *Introduction to High-Performance Computing*. Le programme de cette école d'été est de présenter différentes techniques de parallélisation sur des architectures à mémoire partagée ou distribué, ou encore sur des GPU.

7 Conclusion

Au cours de cette première année de thèse, j'ai fait un état de l'art des différentes techniques de parallélisation automatique existantes. De plus, je continue mes recherches bibliographiques avec des articles publiés en 2013. Je me suis documenté sur les différentes analyses nécessaires pour valider des transformations que je vais utiliser et plus généralement sur les analyses et transformations de code.

J'ai également pu définir un langage de base sur lequel notre analyse et nos transformations seront développées. À partir de ce langage, j'ai défini différentes opérations qui permettront de mettre en place les propriétés qu'il faudra vérifier pour valider nos transformations.

Une réflexion sur les solutions pour la génération automatique de code parallèle distribué a également été faite. Nous avons proposé des solutions présentées dans ce rapport sous forme d'algorithmes. Je me suis dans un premier temps penché sur la première partie de cette solution et ai présenté les différentes transformations successives à appliquer. J'ai également commencé à implémenter quelques passes dans *PIPS*.

J'ai suivi différents séminaires ou soutenances de thèse se rapportant à mon sujet de thèse.

J'ai prévu d'assister à d'autres séminaires ou conférences concernant mon domaine tout au long de ma thèse. Cet été, j'ai prévu d'assister à une école d'été concernant la parallélisation afin d'approfondir mes connaissances sur les différents langages parallèles. J'attends la réponse en Juin.

D'ici la fin de cette année, je compte finir la première grande étape de mon algorithme, c'est-à-dire aussi bien l'implémentation que les preuves de validité. J'espère également avoir entamé grandement la deuxième partie de l'algorithme.

Concernant l'année suivante, je prévois de terminer la deuxième partie de l'algorithme, et réfléchir sur les différents langages cibles à considérer, comme par exemple les langages de type PGAS, avant de réaliser une formalisation plus précise ainsi que l'implémentation et les preuves de celle-ci. J'espère également publier un article.

Enfin pour ma dernière année, je terminerai l'implémentation et la publication. Je pourrai également réfléchir et mettre en place la prise en compte des pointeurs pour la réalisation de la parallélisation. Durant cette dernière année, je m'attèlerai à la rédaction de mon manuscrit de thèse.

Table des matières

Table des figures

Table des algorithmes

Références

[1] Alfred Vaino Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilateurs principes, techniques et outils*. Pearson Education, Paris, 2e édition edition, 2007.

[2] Béatrice Creusillet. *Analyses de Régions de Tableaux et Applications.* PhD thesis, 1996.

[3] Michael J. C. Gordon. *The denotational description of programming languages : an introduction*. Springer-Verlag, New York, 1979.

[4] Dounia Khaldi. *Parallélisation automatique et statique de tâches sous contraintes de ressources – une approche générique –*. PhD thesis, 2013.

[5] Abdellah-Medjadji Kouadri-Mostéfaoui, Daniel Millot, Christian Parrot, and Frédérique Silber-Chaussumier. Prototyping the automatic generation of MPI code from OpenMP programs in GCC. 2009.

[6] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. A hybrid approach of OpenMP for clusters. In *ACM SIGPLAN Notices*, volume 47, page 75–84. ACM, 2012.

[7] Okwan Kwon, Fahed Jubair, Seung-Jai Min, Hansang Bae, Rudolf Eigenmann, and Samuel P. Midkiff. Automatic scaling of openmp beyond shared memory. In *Languages and Compilers for Parallel Computing*, page 1–15. Springer, 2011.

[8] Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. STEP : a distributed OpenMP for coarse-grain parallelism tool. In *OpenMP in a New Era of Parallelism*, page 83–99. Springer, 2008.

[9] Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. From OpenMP to MPI : first experiments of the STEP source-to-source transformation tool. In *PARCO*, page 669–676, 2009.

[10] Gordon E. Moore. *Cramming more components onto integrated circuits*. McGraw-Hill New York, NY, USA, 1965.

[11] OpenMP. http ://openmp.org/wp/.

[12] PIPS : Automatic Parallelizer and Code Transformation Framework — PIPS Project. http ://pips4u.org/.

[13] Jean-Yves Vet. *Parallélisme de tâches et localité de données dans n contexte multi-modèle de programmation pour supercalculateurs hiérarchiques et hétérogènes*. PhD thesis, 2013.

[14] Hans Zima. *Supercompilers for parallel and vector computers*. ACM Press frontier series. ACM Press ; Addison-Wesley, New York, N.Y. : Wokingham, England ; Reading, Mass, 1990.