PARALLÉLISATION AUTOMATIQUE DE PROGRAMMES SCIENTIFIQUES POUR SYSTÈMES DISTRIBUÉS

par

Félix-Antoine Ouellet

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 13 janvier 2016

Le 13 janvier 2016

le jury a accepté le mémoire de Monsieur Félix-Antoine Ouellet dans sa version finale.

Membres du jury

Professeur Hugo Larochelle
Directeur de recherche
Département d'informatique

Professeur Gabriel Girard
Codirecteur de recherche
Département d'informatique

Professeur Richard St-Denis
Membre interne
Département d'informatique

Professeur Shengrui Wang
Président-rapporteur
Département d'informatique

Sommaire

Avec l'avènement des processeurs multi-coeurs comme architecture standard pour ordinateurs de tout acabit, de nouveaux défis s'offrent aux programmeurs voulant mettre à profit toute cette nouvelle puissance de calcul qui leur est offerte. Malheureusement, la programmation parallèle, autant sur systèmes à mémoire partagée que sur systèmes à mémoire distribuée, demeure un défi de taille pour les développeurs de logiciels. Une solution intéressante à ce problème serait de rendre disponible un outil permettant d'automatiser le processus de parallélisation de programmes. C'est dans cette optique que s'inscrit le présent mémoire. Après deux essais infructueux, mais ayant permis d'explorer le domaine de la parallélisation automatique dirigée par le compilateur, l'outil *Clang-MPI* a été conçu pour répondre au besoin énoncé. Ainsi, cet outil prend en charge la parallélisation de programmes originellement séquentiels dans le but de produire des programmes visant les systèmes distribués. Son bon fonctionnement a été évalué en faisant appel aux bancs d'essai offerts par la suite *Polybench* et ses limites ont été explorées par une tentative de parallélisation automatique du processus d'entraînement d'un réseau de neurones.

Mots-clés: compilateur, parallélisation automatique, systèmes distribués, réseau de neurones.

Remerciements

Je tiens à remercier mes superviseurs, Hugo Larochelle et Gabriel Girard, pour m'avoir laissé autant de liberté dans la poursuite de ma maîtrise. Je veux aussi remercier mes collègues du laboratoire de recherche avancée en calcul interactif, scientifique et technique. Ils ont su rendre le quotidien de la maîtrise agréable même dans les moments où le moral n'était plus là. Finalement, je veux remercier ma famille pour tout le soutien moral qu'elle m'a donné tout au long de ma maîtrise.

Abréviations

ASA Arbre syntaxique abstrait

DSWP *Decoupled Software Pipelining*

GDP Graphe de dépendence de programme

GFC Graphe de flux de contrôle

GIL *Global Interpreter Lock*

HPCS *High Productivity Computing Systems*

IR *Intermediate Representation*

LLVM *Low Level Virtual Machine*

MPI *Message Passing Interface*

NaN *Not a Number*

MPMD *Multiple Programs, Multiple Data*

PGAS *Partitioned Global Address Space*

SCoP *Static Control Parts*

SSA *Static Single Assignment*

Table des matières

Liste des figures

Liste des figures

Liste des tableaux

Liste des programmes

Liste des programmes

Introduction

Contexte

Depuis les dix dernières années, le monde de l'informatique effectue un virage de plus en plus prononcé vers la multiprogrammation. En effet, motivés par les gains de moins en moins substantiels offerts par la loi de Moore, les concepteurs de processeurs se sont mis à offrir des architectures comprenant de plus en plus de coeurs. Face à ce changement matériel, les concepteurs de logiciels n'ont eu d'autres choix que d'adapter leurs pratiques à cette nouvelle réalité.

Cependant, le développement de systèmes parallèles demeure une discipline très ardue de la programmation. Et ce l'est d'autant plus dans le cas de l'implémentation de systèmes distribués dans lesquels non seulement les calculs, mais aussi les données, doivent être répartis de manière intelligente. Dans une optique similaire, il faut aussi reconnaître que bien des systèmes informatiques d'envergure datent de plusieurs années voire plusieurs décennies et que leur réécriture pour les faire entrer dans l'ère des processeurs multi-coeurs risque de s'avérer une trop grande entreprise pour les compagnies propriétaires de ces dits systèmes. C'est donc dans ce contexte que les outils de parallélisation automatique de code ont vite gagné en popularité.

Objectifs

Le but premier de cette maîtrise est de mettre en place un système de parallélisation automatique de programmes écrits en C/C++ agissant au travers d'annotations de programmes visant à dicter les intentions du programmeur au compilateur. Plus spécifiquement, ces annotations aident le compilateur à paralléliser un programme

traité sur un système distribué par le biais de génération de code MPI. C'est donc dans cette optique que nous avons développé une variante du compilateur *Clang*, nommée *Clang-MPI*.

La gradation du succès de ce projet passe par la parallélisation automatique de programmes dits scientifiques, en d'autres termes, des applications effectuant des calculs d'algèbre linéaire. Concrètement cela passe par la parallélisation automatique d'une partie des programmes offerts par le banc d'essai *PolyBench*. Après cette première étape, une évaluation des capacités de l'outil développé a été faite par le biais d'une tentative de parallélisation de l'algorithme d'apprentissage de base d'un réseau de neurones, c'est-à-dire la rétropropagation. Le but ici est de déterminer s'il est possible qu'un algorithme fondamentalement séquentiel puisse être optimisé par un outil de parallélisation automatique pour être exécuté plus efficacement sur un système distribué.

Résultats

Pour tester l'outil mis de l'avant dans cette maîtrise, nous avons utilisé la suite de bancs d'essai *PolyBench*. Ces tests ont révélé que *Clang-MPI* permet de paralléliser automatiquement certains programmes de telle façon que leurs temps d'exécution rivalisent de très près avec des programmes MPI codés à la main. Cependant, dans d'autres cas, les temps d'exécution des programmes transformés à l'aide de *Clang-MPI* sont bien plus importants que ceux de programmes MPI codés à la main. En fait, dans ces cas, le programme transformé peut même être plus lent que le même programme dans sa forme séquentielle. La principale raison expliquant ces faits est que *Clang-MPI* produit plus d'appels à des fonctions MPI qu'il est nécessaire.

De plus, une fois le moment venu de paralléliser automatiquement le processus d'apprentissage dans un réseau de neurones, l'outil développé s'est avéré inadéquat. En effet, il est incapable d'effectuer son travail peu importe la forme que revêt le réseau de neurones. Les principales raisons de cet échec sont liées à des choix d'architecture combinés avec le fait que l'algorithme d'apprentissage dans un réseau de neurones ne se prête pas, à la base, à l'exécution sur un système à mémoire distribuée.

Structure du mémoire

Le mémoire est présenté de la façon suivante. Tout d'abord, le chapitre 1 est consacré à la présentation des notions de base nécessaires pour comprendre ce mémoire et à la revue de littérature. Ensuite, le chapitre 2 offre une introduction à l'infrastructure de compilation LLVM utilisé pour construire l'outil *Clang-MPI*. Par après, le chapitre 3 décrit deux tentatives non fructueuses d'atteindre mon but qui était de paralléliser automatiquement un réseau de neurones et son algorithme d'apprentissage sur un système distribué comme une grappe de calcul. Le chapitre 4 traite de la parallélisation automatique aidée par des annotations destinées au compilateur. Finalement, le chapitre 5 présente et commente les résultats expérimentaux obtenus grâce à la parallélisation automatique aidée d'annotations.

Chapitre 1

Revue de littérature et notions de base

Le but de ce chapitre est de donner une vue d'ensemble des grands champs de recherche relatifs au sujet abordé dans ce mémoire. De fait, bien que le but poursuivi, c'est-à-dire produire un outil permettant de paralléliser automatiquement un réseau de neurones sur un système distribué, ait demeuré le même pendant tout ce travail de recherche, la méthode utilisée pour tenter de l'atteindre a changé à de nombreuses reprises. Suivant l'ordre dans lequel ces champs de recherche ont été abordés dans ce mémoire, ce chapitre est structuré de la façon suivante. Tout d'abord, la section 1.1 présente un survol des projets de parallélisation automatique de programmes écrits dans le langage Python. Ensuite, la section 1.2 introduit le lecteur au modèle polyédral de compilation et les possibilités qu'il offre pour exploiter le parallélisme intrinsèque de certains programmes scientifiques. Finalement, la section 1.3 conclut le chapitre en s'attardant sur la parallélisation automatique effectuée à partir d'annotations de code.

```
# Creation d'un tenseur M x N x K contenant des valeurs
# aleatoires
x = numpy.random.rand(M, N, K)

# Creation d'une matrice min(M, N) x K contenant des zeros
y = numpy.zeros((numpy.min((M, N)), K))

# Calcul des valeurs singulieres
for i in xrange(K):
  y[..., i] = numpy.linalg.svd(x[..., i])[1]
```

Programme 1.1 – Décomposition en valeurs singulières d'une matrice $M \times N$ avec *Numpy*

1.1 Parallélisation automatique de programmes Python

Cette section offre un regard historique sur les divers projets entrepris pour paralléliser automatiquement des programmes écrits dans le langage de programmation Python. Dans le but de souligner la difficulté d'un tel projet, cette section débute avec les approches qui furent abandonnées avant leur mise au point finale. Par après, des approches plus récentes, et à un certain égard plus modestes, sont abordées.

1.1.1 Projets abandonnés

Il existe plusieurs façons de paralléliser implicitement et automatiquement des programmes écrits en Python. Une de ces façons est d'avoir recours à une bibliothèque qui offre des fonctions en apparence séquentielles mais qui en réalité s'exécutent de manière parallèle. *Star-P* [Cho04] se base sur cette idée pour offrir une interface très similaire à *Numpy* [Oli07], une bibliothèque pour le calcul scientifique en Python, qui dissimule ce qui est en fait une architecture client-serveur. Les Programmes 1.1 et 1.2 illustrent cette différence minime présente entre un programme utilisant *Numpy* et un programme utilisant *Star-P*.

Une critique que l'on peut faire de ce projet, et même de l'approche qu'il utilise en général, est qu'il contraint l'utilisateur dans les programmes qu'il peut écrire: tout

```
# Creation d'un tenseur M × N × K contenant des valeurs
# aleatoires
x = numpy.random.rand(M, N, K)

# Creation d'une matrice min(M, N) × K contenant des zeros
y = numpy.zeros((numpy.min((M, N)), K))

# Calcul des valeurs singulieres
Y = starp.ppeval(numpy.linalg.svd, x)[1]
```

Programme 1.2 – Décomposition en valeurs singulières d'une matrice $M \times N$ avec *Star-P*

doit pouvoir se faire avec l'interface offerte aux développeurs. Ce projet fut abandonné suite à l'achat par Microsoft de la compagnie qui le développait. Le code ne fut jamais ouvert au public.

Une autre façon de paralléliser automatiquement un programme écrit en Python est de compiler ce dit programme pour ensuite utiliser des techniques propres à la compilation statique et à la compilation dynamique pour tenter de le paralléliser. L'exemple le plus ambitieux à ce jour d'un projet empruntant cette voie est le projet *Unladen-Swallow* réalisé par Google [Goo09]. Il avait pour but de transformer un script écrit en Python en un programme écrit dans la représentation intermédiaire de LLVM [Lat04] ce qui permettrait ensuite de l'optimiser, notamment en exploitant le parallélisme au niveau des instructions. Ce projet se solda par un échec pour plusieurs raisons qui ont miné et continue de miner tout essai de compiler des programmes écrits en Python.

Tout d'abord, le système de types de Python obéit au style de typage appelé *duck typing*. En d'autres termes, une variable se voit assigner un type basé sur son comportement dans le programme jusqu'au point courant. Ainsi, une variable à laquelle on affecte un tableau, et utilisée comme telle, est de type tableau. Toutefois, Python permet, par exemple, d'affecter un entier à cette variable plus loin dans le programme. Cette caractéristique rend la compilation de programmes Python relativement ardue. Puisque la représentation intermédiaire de nombreux compilateurs, en particulier LLVM mis en cause dans le cadre de *Unlanden-Swallow*, est typée et

donc il faut déduire les types utilisés implicitement dans son programme Python. En fait, comme pour tout autre langage de programmation typé dynamiquement, le problème de l'inférence statique de types pour Python est un problème indécidable [Wel96]. Bien que certains algorithmes, telle que l'interprétation abstraite [Cou77], permettent d'obtenir de bonnes estimations quant aux types de chaque variable utilisées à tout point d'un programme donné, le problème ne peut être entièrement résolu.

Dans un même ordre d'idée, la déduction de la dimension des objets en Python, également requise pour produire une représentation intermédiaire valide du programme, demande un travail d'inférence non négligeable. En effet, cette tâche est cruciale dans le cadre de la parallélisation automatique de programmes qui s'exécutent sur des systèmes à mémoire distribuée, car la distribution des structures de données est incontournable. Une façon de faire serait, encore une fois, d'avoir recours à des techniques telles que l'interprétation abstraite, mais ceci constitue un projet à part entière.

Finalement, il faut savoir que l'exploitation du parallélisme en Python est sévèrement limitée par le *Global Interpreter Lock* (GIL). Le GIL est un verrou utilisé par un interpréteur Python pour s'assurer qu'un seul fil d'exécution s'exécute à la fois. Ceci a pour effet d'éliminer beaucoup d'opportunités de parallélisme étant donné que les fils d'exécution Python se retrouvent à être des fils d'exécution en espace utilisateur. Par conséquent, un programme Python ayant été parallélisé automatiquement en ajoutant des appels à des bibliothèques d'outils parallèles Python ne s'exécuterait tout de même pas de manière parallèle. Pour contourner ce problème, deux solutions existent. D'un côté, il existe des implémentations de l'interpréteur qui ne possède pas de GIL. Cependant, elles sont encore quelque peu expérimentales compte tenu que pour arriver à leurs fins, elles doivent réécrire la gestion de la mémoire dans l'interpréteur qui n'était pas *thread safe* à la base, d'où la nécessité du GIL. D'un autre côté, on peut accéder au parallélisme en Python en mettant à profit des appels à des fonctions parallèles écrites dans un autre langage de programmation tel C/C++. Ainsi, bien qu'un programme Python soit limité à un fil d'exécution, les fonctions externes d'autres langages n'ont pas ces restrictions.

```
x = numpy.random.rand(M, N, K)
@jit
def conv_3x3_trim (image, weights):
  result = np.zeros_like(image)
  for i in xrange(1, image.shape[0]-2):
    for j in xrange(1, image.shape[1]-2):
      window = image[i-1:i+2, j-1:j+2]
      result[i, j] = np.sum(window*weights)
  return result
```

Programme 1.3 – Convolution avec *Parakeet*

1.1.2 Projets en cours

Une troisième façon de faire est d'utiliser des décorateurs, c'est-à-dire un attribut de fonction, pour modifier le comportement d'une fonction et la rendre parallèle. C'est dans cette optique que s'inscrit le projet *Parakeet* [Rub12]. Avec cet outil, on tente donc de permettre à l'utilisateur de paralléliser automatiquement et facilement les parties de code qu'il aura choisies de façon explicite. Concrètement, un programme écrit avec *Parakeet* ressemble à ce qui est présenté dans le Programme 1.3.

À l'interne, le décorateur (*@jit* dans ce cas-ci) va indique d'intercepter le premier appel à la fonction et la compiler pour les appels subséquents. Le processus de compilation utilisé consiste à transformer la fonction en LLVM IR et ensuite, à l'aide d'heuristiques, à décider le module d'exécution, soit sur un GPU, soit sur les multiples CPUs d'une machine ou soit en utilisant l'interpréteur Python si rien ne peut être fait pour paralléliser la fonction. Bien que connaissant beaucoup de succès pour un certain ensemble de calcul algébrique, cet outil demeure limité dans dans ses optimisations. Les problèmes rencontrés sont similaires à ceux identifiés dans le projet *Unladen-Swallow*. De plus, *Parakeet* n'effectue pas une analyse très poussée du code qu'il doit paralléliser, ce qui limite l'étendue de ses capacités. Il n'a pas une bonne connaissance des bibliothèques de calcul scientifique tel que *Numpy*, ce qui rend son utilisation délicate dans des programmes faisant appel à ces bibliothèques. Par conséquent, outre des programmes présentant des boucles bien explicites ou utilisant des fonctions *Numpy* bien précises, il ne peut rien optimiser de manière avantageuse.

```
import bohrium as numpy

def solve(grid, epsilon):
  center = grid[1:-1,1:-1]
  north = grid[-2:,1:-1]
  south = grid[2:,1:-1]
  east = grid[1:-1,:2]
  west = grid[1:-1,2:]
  delta = epsilon+1
  while delta > epsilon:
    tmp = 0.2 * (center + north + south + east + west)
    delta = numpy.sum(numpy.abs(tmp - center))
    center[:] = tmp
```

Programme 1.4 – Solution à l'équation de la chaleur avec *Bohrium*

Figurant parmi les projets tentant d'effectuer une parallélisation automatique par l'entremise d'une bibliothèque, *Bohrium* [Kri13] tente d'offrir une plateforme sur laquelle un programme pourra être exécuté de façon parallèle autant sur processeur multi-coeurs que sur un processeur graphique. Dans le cadre de l'exécution parallèle automatique de code Python, *Bohrium* offre une nouvelle version de la bibliothèque *Numpy*. Le Programme 1.4 offre un exemple d'un programme Python écrit en se servant de *Bohrium*.

Derrière ces appels de fonctions, *Bohrium* produit du code dans une représentation intermédiaire mettant en évidence les calculs vectoriels effectués dans un programme donné. Ce faisant, *Bohrium* tente d'extraire le plus d'information possible au niveau de l'utilisation des tableaux *Numpy* afin de déterminer la façon la plus avantageuse de paralléliser le noyau de calcul analysé. Bien que prometteur, ce projet est encore en début de développement. Il démontre de bonnes accélérations face à des applications écrites en Python, mais ne se compare pas encore avec des programmes écrits dans des langages permettant une meilleure exploitation du parallélisme natif des architectures modernes tel C/C++.

Si l'on étend nos horizons en ce qui concerne la parallélisation de programmes écrits en Python pour inclure des approches un peu plus explicite, le projet *Trickle* [Ben07] offre un outil intéressant pour paralléliser des programmes sur un système

distribué. En effet, cet outil offre une réelle approche *Multiple Programs, Multiple Data* (MPMD) ; le programmeur écrit les fonctions nécessaires à l'exécution de son programme et les distribue ensuite avec les données qu'elles requièrent sur un ensemble de machines. Le Programme 1.5 montre un exemple de son utilisation.

Comme on peut le constater en observant la fin de l'exemple, *Trickle* fonctionne en injectant du code dans des interpréteurs distants. Ce code est ensuite exécuté sur chaque machine avec la partie des données qui lui a été assignée. Le résultat peut ensuite être collecté sur un noeud qu'on peut qualifier de maître (celui démarrant le calcul) tel que dans le Programme 1.5 ou il peut être utilisé directement pour des calculs subséquents sans nécessairement respecter le modèle maître-esclave. Bien qu'intéressant, cet outil demande un certain effort de configuration de la part de ses utilisateurs. Ainsi, chaque machine dans un système distribué doit être configuré de sorte qu'un interpréteur Python y soit présent et apte à recevoir des fonctions et des données par l'entremise de *Trickle*. Autrement dit, cet outil demande au programmeur de planifier l'exécution de ses programmes dans un contexte distribué, ce qui peut être plus demandant que ce à quoi le programmeur peut s'attendre. La différence entre cette approche et celle utilisant MPI est relativement faible. Les avantages de Trickle sont plutôt au niveau de la facilité et de la rapidité d'écriture de programmes puisque son API est plus concise que celle d MPI. De plus, comme Trickle est une bibliothèque écrite en Python, elle tire avantage de la vitesse de prototypage de Python.

Tentant de reprendre le flambeau laissé par *Unladen-Swallow*, *Numba* [Con12] est un projet mis de l'avant par le créateur de *Numpy*. Ce qui le différencie de *Unladen-Swallow* est qu'il est conscient des objets *Numpy* présent dans le code. En effet, *Numba* profite du fait que les objets *Numpy*, soit des tableaux construits par l'entremise de l'interface de *Numpy*, sont tous typés dès leur création. Ceci aide donc grandement le processus de traduction de code Python en LLVM IR qui, comme il a déjà été mentionné, est typé.

Le Programme 1.6 illustre bien ce qui constitue à la fois la force et la faiblesse du projet *Numba*. D'un côté, la parallélisation du calcul requiert peu d'effort. Ici, cela se résume à l'ajout d'un décorateur à la fonction *sum2d*. D'un autre côté, l'exemple du Programme 1.6 présente le seul type de code que *Numba* est capable d'optimiser et de paralléliser efficacement. Effectivement, le domaine d'application de *Numba* se

```
import sys, glob

# Fonction comptant les mots dans une liste de fichiers
def wordcount(files):
  m = {}
  for filename in files:
    f = open(filename)
    for l in f:
      for t in l.split():
        m[t] = m.get(t,0) + 1
  f.close()
  return m

# Fonction regroupant les comptes de mots
def mergecounts(dlist):
  m = {}
  for d in dlist:
    for w in d:
      m[w] = m.get(w,0) + d[w]
  return m

# Acquisition des donnees
if len(sys.argv) != 4:
  print "usage: %s <vmcount> <chunksize> <pattern>" \
        %sys.argv[0]
  sys.exit(1)
n = int(sys.argv[1]); cs = int(sys.argv[2])
files = glob.glob(sys.argv[3] + "*")

# Detection des interpreteurs distants a notre disposition
vmlist = connect(n)

# Injection de la fonction wordcount dans les interpreteurs
# distants
inject(vmlist, wordcount)

# Execution de wordcount sur les interpreteurs distants. On
# envoie un nombre de fichiers cs a chaque interpreteur.
rlist = forwork(vmlist, wordcount, files, chunksize=cs)

# Regroupe les comptes de mots
final = mergecounts(rlist)
```

Programme 1.5 – Comptage des mots dans une série de documents avec *Trickle*

```
from numba import jit
from numpy import arange

@jit
def sum2d(arr):
  M, N = arr.shape
   result = 0.0
   for i in range(M):
     for j in range(N):
       result += arr[i,j]
   return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

Programme 1.6 – Sommation des éléments d'une matrice avec *Numba*

limite aux programmes manipulant des tableaux *Numpy* et où les calculs sont faits au moyen de boucles explicites.

Finalement, il existe encore une autre approche à la compilation, l'optimisation et l'éventuel parallélisation de code écrit en Python. Cette approche consiste à développer un langage dédié interne à Python qui est en mesure d'optimiser ce dont un programmeur dans un domaine d'application spécifique a besoin. *Theano* [Ber10] est un outil qui tente de mettre à profit cette approche. Il permet donc d'écrire ce qui est en fait des graphes de calcul décrivant des opérations mathématiques que le programmeur souhaite réaliser. Le Programme 1.7 offre un exemple d'utilisation de *Theano*. Il met en relation deux variables de type *dmatrix* au travers une addition. La compilation, lors de la construction de la fonction *f*, s'effectue en deux grandes étapes. Dans un premier temps, le graphe exprimé par l'expression *z* est construit et optimisé pour produire l'équivalent C de la fonction. Par la suite, le code C produit sera compilé avec les options d'optimisation maximale. Il est possible que la fonction soit compilée pour utiliser *OpenMP* ou *Cuda* selon les paramètres du fichier de configuration. Subséquemment, tout appel à la fonction *f* sera redirigé vers la fonction compilée. *Theano* permet de produire du code parallèle pour un processeur multicoeurs ou pour un processeur graphique, mais pas pour de multiples processeurs dans

```
import theano.tensor as T
from theano import function

# Creation d'un noeud de donnees
x = T.dmatrix('x')

# Creation d'un noeud de donnees
y = T.dmatrix('y')

# Creation d'un noeud de calcul
z = x + y

# Le calcul est compile et optimise dans une fonction Theano
f = function([x, y], z)

# La fonction est executee. Cette execution peut etre faite
# sur GPU ou sur de multiples CPU
f([[1, 2], [3, 4]], [[10, 20], [30, 40]])
```

Programme 1.7 – Multiplication de matrices avec *Theano*

un contexte distribué.

1.2 Parallélisation automatique dans le modèle polyédral

Dans cette section, nous introduisons le modèle polyédral de compilation. Une revue des essais qui ont été tentées pour utiliser ce modèle à des fins de parallélisation automatique, autant pour des systèmes à mèmoire partagée que des systèmes à mémoire distribuée est également présentée.

1.2.1 Introduction au modèle polyédral

De nos jours, l'approche la plus courante pour optimiser des programmes lors de la compilation est d'enchaîner des passes d'optimisations basées sur le flot d'informa-

```
for (int i = 1; i <= N; ++i)
  for (int j = 1; j <= N; ++j)
    if (i <= N - j + 2)
          S[i] = ...
```

Programme 1.8 – Boucles traitées par le modèle polyédral

tion d'une représentation intermédiaire. Elle est pensée de sorte qu'elle expose mieux, du point de vue du compilateur, le flot d'information d'un programme. Cette représentation intermédiaire a été obtenue à partir d'un parcours de l'arbre syntaxique abstrait. GCC et LLVM sont deux des exemples les plus connus de cette manière de procéder. Une critique de cette technique d'optimisation repose sur l'impossibilité appliquer systématiquement des transformations complexes qui exploitent la localité et le parallélisme présents dans des structures répétitives. C'est dans cet esprit que la compilation polyédrale a vu le jour [Bas04, Mel02].

L'idée derrière la compilation polyédrale est de représenter des boucles dans un programme compilé à l'aide de polytopes. La dimension des polytopes correspond à la profondeur de l'imbrication du regroupement de boucles que l'on tente d'optimiser. Ainsi, on peut appliquer une série d'opérations algébriques sur ces polytopes dans le but d'optimiser le programme autant du point de vue du temps de calcul que de la mémoire utilisée. La figure 1.1 présente un exemple d'une représentation polyédral construite à partir du Programme 1.8, tous les deux empruntés de [Pou10].

1.2.2 Extraction de la représentation polyédrale

Dans le processus de compilation polyédrale, l'unité de base se nomme *partie à contrôle statique* (abrégé SCoP d'après l'expression originale *static control part*) [Bas04]. Un SCoP est défini comme l'ensemble des énoncés compris dans une boucle structurée d'une façon qui expose des informations-clés au compilateur. Ainsi, les bornes de la boucle et les accès mémoire effectués dans cette dernière doivent être des fonctions affines des itérateurs et des paramètres avoisinants. En d'autres termes, elles doivent pouvoir être représentées par addition et multiplication des itérateurs et des paramètres avoisinants tels que des constantes du programme ciblé. Les Programmes

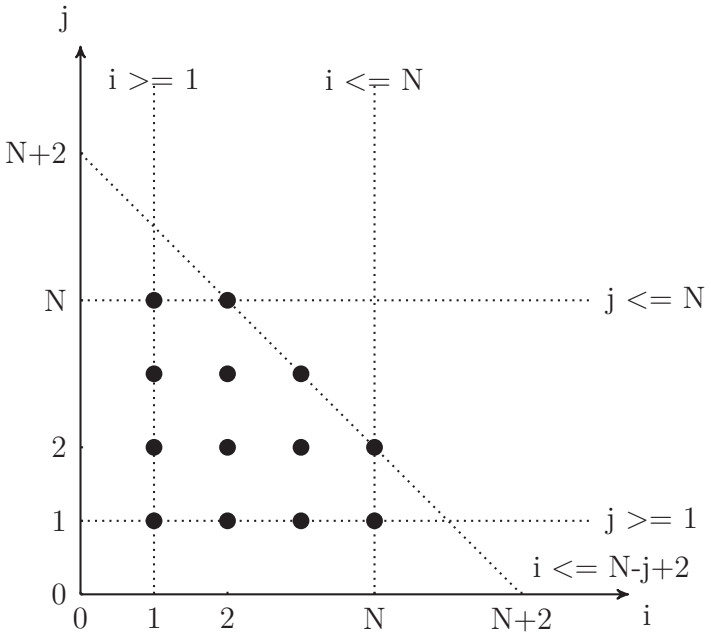

Figure 1.1 – Espace d'itérations du Programme 1.8 défini par un polytope à deux dimensions

```
for (int i = 0; i < 32; ++i)
  for (int j = 0; j < 1000; ++j)
    A[i][j] += 10;
```

Programme 1.9 – Boucles imbriquées formant un SCoP

1.9 et 1.10 illustrent à haut niveau ce qui est considéré comme un SCoP et ce qui ne l'est pas.

Plus explicitement, le Programe 1.9 est un exemple de SCoP étant donné qu'il vérifie tous les critères d'un SCoP. Ainsi, les bornes des boucles impliquées peuvent être connues statiquement. De plus, le pas de la variable d'induction est une fonction affine, une simple addition de 1. Finalement, l'accès fait au tableau multidimensionnel A est fait à partir de fonctions affines des variables d'induction des boucles, soit l'identité de ces variables. Le Programme 1.10, quant à lui, n'est pas un exemple de SCoP compte tenu que le pas de la variable d'induction de la boucle n'est pas une fonction affine.

```
for (int i = 1024; i > 0; i/=2)
  A[i] += 10;
```

Programme 1.10 – Boucle ne formant pas un SCoP

Il est important de préciser qu'il s'agit d'une vue de haut niveau. Bien souvent, le compilateur n'agit pas directement les boucles telles que codées dans le programme original, il agit plutôt des parties du graphe de flot de contrôle (abrégé CFG d'après l'expression originale *control flow graph*), une représentation graphique de la représentation intermédiaire d'un programme, qui représentent ces boucles. À titre d'exemple, la figure 1.2 illustre le CFG représentant la boucle présente dans le Programme 1.9 qui serait présente dans une fonction qui n'effectuerait aucun autre travail que l'exécution de cette boucle. On précise ici que le CFG présenté est celui produit par LLVM. Pour plus de détail sur le travail effectué par LLVM et les structures qu'il utilise, notamment sa représentation intermédiaire, on réfère le lecteur au chapitre 2.

Lorsqu'un SCoP est détecté, le compilateur extrait l'information importante de la boucle pour en bâtir une représentation polyédrale. Cette représentation est composée de trois éléments: le domaine d'exécution des énoncés D_{Stmt}, l'ordre d'exécution de chaque instance de chaque énoncé S_{Stmt} et les accès mémoire effectués par chaque énoncé A_{Stmt}. Un exemple de représentation polyédrale est présenté à la Figure 1.3.

Pour mieux comprendre la représentation polyédrale, précisons les informations que l'on a extrait du code présent à la Figure 1.3. Tout d'abord, les divers ensembles contenant les informations servant à la compilation polyédrale sont tous composés d'un seul élément $Stmt[i, j]$, car la boucle ne contient qu'un seul énoncé à exécuter et ce dernier utilise les variables d'induction i et j. L'ensemble du domaine d'exécution D_{Stmt} contient les bornes des variables d'induction. Dans le cas étudié, il s'agit de 0 et 32 pour la variable i et 0 et 1000 pour la variable j. L'ensemble de l'ordre d'exécution S_{Stmt} indique que les énoncés concrets ($Stmt[0, 0]$, $Stmt[0, 1]$ et ainsi de suite) sont exécutés dans l'ordre usuel, c'est-à-dire, en exécutant d'abord complètement une boucle de la structure répétitive interne avant de faire progresser l'exécution d'une structure répétitive externe. L'ensemble des accès mémoire A_{Stmt} offre des informations sur la façon dont les structures présentes dans le SCoP sont accédées. Dans le cas présent, chaque énoncé concret $Stmt[i, j]$, c'est-à-dire chaque énoncé exécuté lors

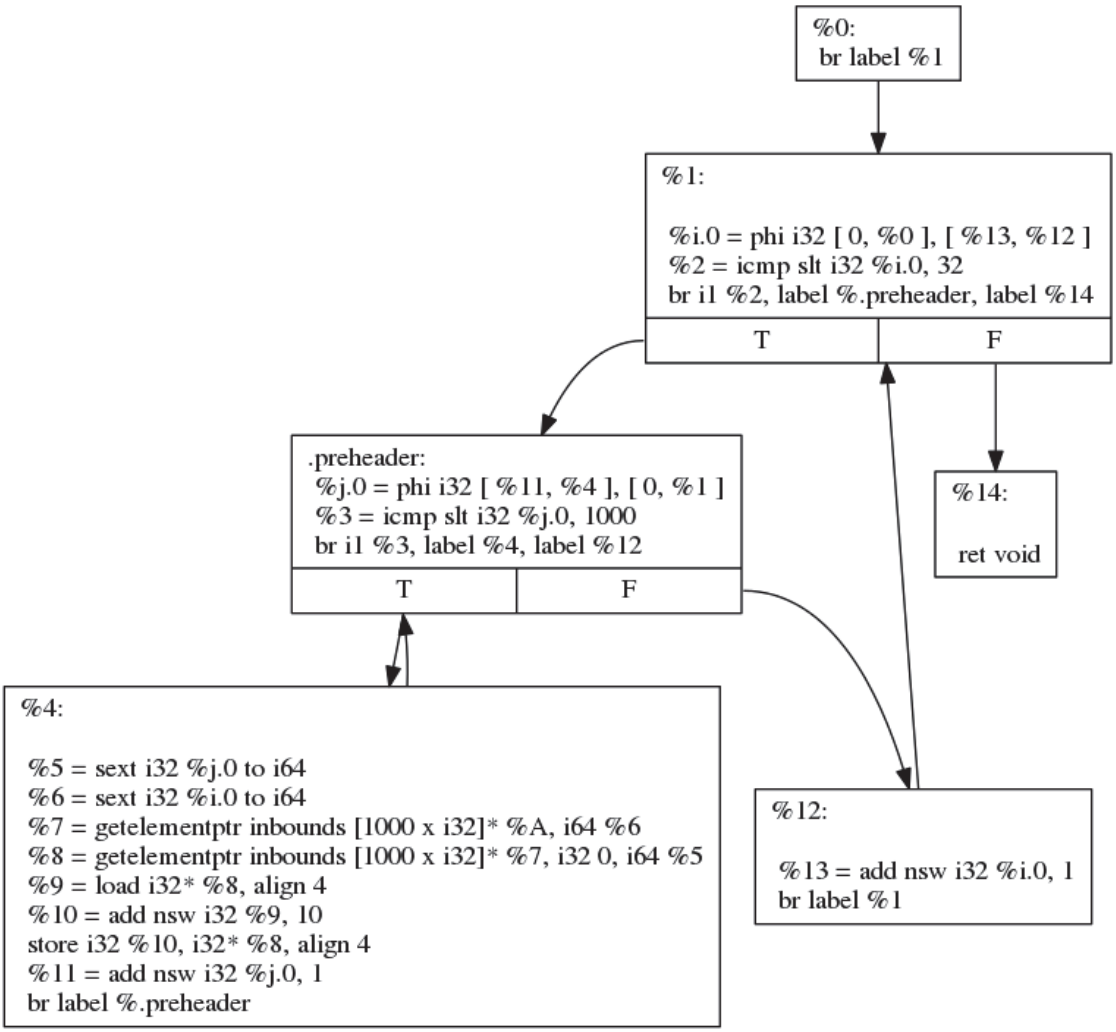

Figure 1.2 – Exemple de graphe de flot de contrôle (CFG)

`for (int i = 0; i < 32; ++i)`
`  for (int j = 0; j < 1000; ++j)`
`  A[i][j] += 10; // Stmt`
	Code

$D_{Stmt} = \{Stmt[i,j] : 0 \leq i < 32 \wedge 0 \leq j < 1000\}$ $S_{Stmt} = \{Stmt[i,j] \rightarrow [i,j]\}$ $A_{Stmt} = \{Stmt[i,j] \rightarrow A[i,j]\}$	Représentation polyhédrale

Figure 1.3 – Exemple d'une représentation polyhédrale

d'une itération $I_{i,j}$, accède à la structure A en se servant strictement des valeurs des variables d'induction de l'itération courante. Cet ensemble d'information constitue la représentation polyédrale de la boucle du Programme 1.9.

1.2.3 Optimisation polyédrale

Une fois la représentation polyédrale extraite, il est maintenant possible d'appliquer des transformations sur le code dans le but de l'optimiser en termes de vitesse d'exécution et de localité d'accès mémoire. Pour expliciter ce processus d'optimisation dans le modèle polyédral, l'exemple précédent est repris et optimisé.

Tout d'abord, la transformation appelée échange de boucles [All02, Mid12] est appliquée au SCoP. Cette transformation est normalement appliquée dans les cas où la disposition des structures de données en mémoire fait en sorte que l'inversion de l'imbriquement de boucles améliore la performance du programme par une meilleure utilisation de l'antémémoire. Ceci est généralement déterminé par une analyse des accès mémoire effectués combinée avec une analyse de l'architecture pour laquelle un programme est compilé. La Figure 1.4 illustre cette transformation.

$T_{Interchange} = \{[i,j] \rightarrow [j,i]\}$ $S'_{Stmt} = S_{Stmt} \circ T_{Interchange}$	Transformation

`for (int j = 0; j < 1000; ++j)`
`  for (int i = 0; i < 32; ++i)`
`  A[i][j] += 10; // Stmt`
	Code transformé

Figure 1.4 – Échange de boucles

Par la suite, une transformation nommée *strip mining* [All02, Mid12] est appliquée sur le SCoP. Cette transformation consiste à sectionner une boucle en une série de blocs qui sont par la suite parcourus à l'aide d'une nouvelle boucle interne. Cette transformation est souvent utilisée comme première étape d'une autre transformation appelée *loop tiling* ou pour préparer le code à utiliser des opérations vectorielles [All02]. La Figure 1.5 montre l'effet de cette transformation.

Cet exemple, illustré par les Figures 1.4 et 1.5, permet de mettre en relief la principale, et la plus intéressante, caractéristique de l'optimisation dans le modèle polyédral.

$T_{Interchange} = \{[i,j] \rightarrow [j,i]\}$ $T_{StripMine} = \{[i,j] \rightarrow [i,jj,j] : jj \mod 4 = 0 \wedge jj \leq j < jj + 4\}$ $S'_{Stmt} = S_{Stmt} \circ T_{Interchange} \circ T_{StripMine}$	Transformation

`for (int j = 0; j < 1000; ++j)`
`  for (int ii = 0; ii < 32; ii += 4)`
`  for (int i = ii; i < ii+4; ++i)`
`  A[i][j] += 10; // Stmt`
	Code transformé

Figure 1.5 – *String mining*

En effet, comme il a pu être constaté, appliquer une transformation dans le modèle polyédral se résume à appliquer une opération algébrique sur l'ordre d'exécution du SCoP traité. Ceci permet potentiellement d'appliquer, en une seule passe, une optimisation constituée de plusieurs transformations et de plusieurs opérations algébriques. Cette caractéristique est un très grand avantage de l'optimisation polyédrale, car cela permet de mieux optimiser des programmes à l'aide de transformations plus complexes tant que celles-ci peuvent être décomposées en une série de transformations simples. De plus, l'application simultanée de plusieurs transformations permet de réduire le temps passé par le compilateur à optimiser un programme donné. C'est une très bonne chose étant donné que la création de la représentation intermédiaire d'un programme est un processus relativement coûteux en terme de temps.

De plus, la nature mathématique des changements à apporter dans le cadre de l'optimisation de boucles potentiellement imbriquées fait en sorte que le modèle polyédral est un choix plus naturel que l'est celui d'une représentation intermédiaire sous la forme *Single Static Assignment* (SSA) comme celle de LLVM (voir le chapitre 2 pour plus de détail sur la représentation intermédiaire de LLVM et la forme SSA). De fait, les transformations intéressantes à appliquer sont celles qui mettent en jeu les accès mémoire à des structures de données et les variables d'induction. En d'autres termes, on tente dans ce contexte de modifier des expressions mathématiques indiquant la progression des accès mémoire et des calculs sur ces derniers. Il va donc de soi que l'application de transformations algébriques sur ces expressions est plus naturel que de tenter de modifier une représentation intermédiaire ressemblant plutôt à un langage d'assemblage.

```
for (int i = 0; i < 100; ++ i) {
  A[i] = B[i] + C[i]; // Stmt1
  D[i] = A[i] + 10;   // Stmt2
}
```

Programme 1.11 – Boucle à paralléliser

1.2.4 Parallélisation automatique

L'intérêt grandissant pour cette technologie qu'est la compilation polyédrale est principalement motivé par le désir d'exploiter les architectures modernes qui offrent de plus en plus de processeurs et donc de plus en plus de ressources pour le parallélisme aux développeurs. L'intuition dans ce contexte consiste tout d'abord à prendre conscience que la représentation polyédrale possède des informations sur toutes les exécutions des énoncés dans un SCoP. Par la suite, il faut réaliser que les transformations effectuées dans le modèle polyédral ne sont en fait que des réordonnancements de l'ordre d'exécution des instances des énoncés dans un SCoP. Par conséquent, il ne suffit que de trouver un ordre d'exécution parallèle respectant les dépendances présentes dans la ou les boucles composant un SCoP.

Pour donner un exemple sur la façon de procéder, considérons la parallélisation de la boucle du Programme 1.11.

La première étape de ce processus, illustrée par la Figure 1.6, consiste à extraire une représentation polyédrale de la boucle. Plus précisément, on veut s'attarder aux ensembles d'accès mémoire, autant en lecture qu'en écriture, des énoncés contenus dans la boucle. Pour ce faire, on raffine l'ensemble des accès mémoire A_{Stmt} en quatre ensembles qui représentent les accès mémoire en écriture et en lecture associés aux énoncés $Stmt1$ et $Stmt2$ (les ensembles A_{Stmt1_W}, A_{Stmt1_R}, A_{Stmt2_W} et A_{Stmt2_R}).

Dans la boucle présentement traitée, on peut constater que la structure A est accédée en écriture au niveau de $Stmt1$ (présence dans A_{Stmt1_W}) et en lecture au niveau de $Stmt2$ (présence dans A_{Stmt2_R}).

La seconde étape du processus consiste à trouver les dépendances potentielles entre les énoncés de la boucle. De fait, si une dépendance portée par la boucle, c'est-à-dire une lecture dans une itération dépend d'une écriture faite lors d'une itération précédente [All02], existe entre deux énoncés traités, le processus de parallélisation

```
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i]; // Stmt1
  D[i] = A[i] + 10;   // Stmt2
}
```

$A_{Stmt1_W} = \{Stmt1[i] \to A[i]\}$
$A_{Stmt1_R} = \{Stmt1[i] \to B[i], Stmt1[i] \to C[i]\}$
$A_{Stmt2_W} = \{Stmt2[i] \to D[i]\}$
$A_{Stmt2_R} = \{Stmt2[i] \to A[i]\}$

Figure 1.6 – Représentation polyédrale des accès mémoire

automatique est arrêté, car aucune boucle contenant une dépendance portée par la boucle ne peut être parallélisée. Dans le cas qui nous concerne, on constate qu'il existe une dépendance entre *Stmt1* et *Stmt2*, car les deux énoncés effectuent un accès mémoire, l'un en écriture et l'autre en lecture. La Figure 1.7 illustre ce résultat.

```
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i]; // Stmt1
  D[i] = A[i] + 10;   // Stmt2
}
```

$A_{Stmt1_W} \cap A_{Stmt1_R} = \emptyset$
$A_{Stmt1_W} \cap A_{Stmt2_R} = A[i]$
$A_{Stmt2_W} \cap A_{Stmt1_R} = \emptyset$
$A_{Stmt2_W} \cap A_{Stmt2_R} = \emptyset$

Figure 1.7 – Intersection des ensembles d'accès

La troisième étape, maintenant que l'on sait qu'il y a une dépendance entre les énoncés de la boucle, consiste à vérifier la distance entre les accès mémoires. En d'autres termes, on veut vérifier le nombre d'itérations qui sépare l'écriture d'un élément $A[i]$ de sa lecture. Ceci permet de déterminer si la dépendance est portée par la boucle, c'est-à-dire qu'elle implique deux itérations différentes, ou non. Pour ce faire, il faut observer les expressions affines utilisées lors des accès mémoire à chaque itération. Plus précisément, il faut vérifier la différence des éléments qui constituent

```
#pragma omp parallel
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i]; // Stmt1
  D[i] = A[i] + 10;   // Stmt2
}
```

Programme 1.12 – Boucle parallélisée à l'aide de OpenMP

ces expressions affines. Dans le cas qui nous intéresse, trouver la distance est trivial, car les deux accès mémoire se font lors de la même itération i. La Figure 1.8 permet de comprendre ce cas.

```
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i]; // Stmt1
  D[i] = A[i] + 10;   // Stmt2
}
```

$$\left.\begin{array}{l} A_{Stmt1_W} = \{Stmt1[i] \rightarrow A[i]\} \\ A_{Stmt2_R} = \{Stmt2[i] \rightarrow A[i]\} \end{array}\right\} \rightarrow i - i = 0$$

Figure 1.8 – Distance entre les accès mémoire

L'étape précédente s'est conclue avec le constat qu'il y a une dépendance entre les énoncés de la boucle, mais cette dépendance n'est pas portée par la boucle, elle en est indépendante. Par conséquent, chaque itération de la boucle peut être parallélisée indépendamment l'une de l'autre. Le Programme 1.12 est le résultat d'une telle parallélisation.

Ce principe général à la parallélisation automatique dans le modèle polyédral a déjà été utilisé dans quelques situations notamment dans le cadre de la génération de code *OpenMP* à partir de code séquentiel [Rag11]. La génération de code parallèle pour processeurs multi-coeurs n'est cependant pas la seule facette de la parallélisation automatique dans laquelle on a tenté de mettre à profit la compilation polyédrale. Ainsi, des travaux comme [Bon13], [Bon11], [Tom13] se sont penchés sur la question de génération de code parallèle distribué au travers des transformations effectuées dans le modèle polyédral de programmes qui sont à la base séquentiels. De plus, les processeurs standards ne sont pas les seuls processeurs ayant été considérés par

des travaux basés sur la compilation dans le modèle polyédral. En effet, on peut recenser quelques efforts, [Ram13] et [Bag10] par exemple, qui ont tenté de proposer des techniques pour paralléliser automatiquement du code en mettant à contribution des processeurs graphiques.

1.2.5 Limites du modèle polyédral

Bien que la compilation polyédrale offre plusieurs nouvelles opportunités au niveau de l'optimisation et de la parallélisation automatique, il reste néanmoins que son champ d'application demeure encore un peu restreint. Ainsi, toute boucle ou imbriquement de boucles ne respectant pas les conditions pour être considéré comme un SCoP ne peut pas profiter d'optimisations polyédrales. De plus, l'analyse de dépendances, comme dans le modèle plus traditionnel de la parallélisation automatique, peut être sévèrement limitée par la présence de pointeurs. En effet, si les accès mémoire se font au travers de pointeurs, le compilateur doit déterminer si les pointeurs réfèrent à la même structure de données ou non. Cette tâche est très difficile, car le problème de l'analyse de pointeurs est un problème indécidable.

Aussi, la compilation polyédrale reste, pour le moment, cantonnée dans la compilation statique. À l'exception des travaux de *PolyJIT* [Sim14], aucun effort n'a même été tenté pour amener la technologie de compilation polyédrale dans le monde de la compilation juste à temps. Ceci peut être expliqué par le fait que les approches courantes ne sont pas conçues à la base pour le permettre. Ainsi, les approches basées sur la modification de code source ne peuvent tout simplement pas être utilisées dans le contexte de la compilation juste à temps. Sinon, les approches basées sur des représentations intermédiaires pré-existantes (on peut penser à celle de LLVM) sont très loin d'être assez performantes pour être utilisées. La cause du problème dans ce cas-ci est que le passage de la représentation intermédiaire de base à la représentation polyédrale, et vice-versa, est très coûteuse en temps, une ressource limitée dans le contexte de la compilation juste à temps.

1.3 Approches historiques à la parallélisation automatique

Dans cette section, nous abordons les approches traditionnelles à la parallélisation automatique. On se penche surtout sur les succès connus en ce qui concerne la parallélisation au niveau des instructions et les difficultés rencontrées lors de tentatives de parallélisation automatique à plus haut niveau (processeurs multi-coeurs, processeurs graphiques, systèmes distribués). Nous introduison aussi la parallélisation/optimisation/transformation de code à l'aide d'outils basés sur les directives au compilateur (*pragmas*).

1.3.1 Parallélisation automatique basée sur la dépendance de données

La recherche sur les façons de paralléliser automatiquement des programmes originalement séquentiels a une longue histoire ne comptant que peu de succès. L'approche ayant connu le plus de succès est celle basée sur l'analyse des dépendances de données présentes dans un programme telle que mise de l'avant par les créateurs du langage de programmation parallèle *High Performance Fortran* (HPF).

Avant d'aborder les transformations à appliquer à un programme pour le paralléliser, il faut d'abord présenter l'analyse du programme qui doit être faite pour orienter par la suite le travail de parallélisation. La première notion à présenter est celle d'une dépendance mémoire et de ses différentes variantes.

Ainsi, on dit qu'il y a une dépendance mémoire entre deux énoncés d'un programme lorsque ceux-ci accèdent à la même donnée que ce soit en lecture ou en écriture. Selon le type d'accès qui est fait par le premier et le second énoncé, en ordre lexicographique, on compte quatre types de dépendances mémoire. Le Tableau 1.1 énumère ces types de dépendances mémoire.

Le Programme 1.13 donne des exemples de plusieurs de ces types de dépendance mémoire. Comme on peut le constater, il existe une vraie dépendance entre la variable a et la variable b dans les deux premiers énoncés. Il y a aussi présence d'une antidépendance entre la variable b et la variable a au niveau des deux derniers énoncés.

Accès 1er énoncé	Accès 2e énoncé	Type de dépendance
Écriture	Écriture	Dépendance en écriture
Écriture	Lecture	Vraie dépendance
Lecture	Écriture	Anti-dépendance
Lecture	Lecture	Dépendance en lecture

Tableau 1.1 – Types de dépendances mémoire

```
void fct() {
  int a = 20;
  int b = a;
  a = 10;
}
```

Programme 1.13 – Exemples de dépendances mémoire

La notion de dépendance devient vraiment intéressante lorsqu'on l'applique dans le contexte des boucles. En effet, une analyse de dépendances dans une boucle donne une bonne indication du niveau de parallélisation de cette dernière. Cependant, il manque une notion avant de pouvoir appliquer ce genre d'analyse aux boucles. Cette notion est celle de distance entre dépendances. En outre, dans une boucle qui fait des accès mémoire à une structure par l'entremise de la variable d'induction de la boucle, il se peut qu'un élément de la structure accédé lors d'une itération de la boucle soit accédé à nouveau lors d'une itération subséquente. Le Programme 1.15 présente un exemple de cette situation. En effet, l'élément *A[i+1]* est accédé en écriture lors d'une itération donnée et est accédé ensuite en lecture via l'expression *A[i]* lors de l'itération suivante. On dira de cette situation qu'il y a présence d'une dépendance portée par la boucle (une dépendance impliquant deux itérations d'une même boucle) dont la distance est de 1.

Cette situation en est une dans laquelle le parallélisme potentiel de la boucle est très faible, voire carrément nul. À l'inverse, une situation telle que montrée dans le Programme 1.14, où il y a présence de dépendances qui sont indépendantes de la boucle, offre un plein potentiel d'exécution parallèle pour la boucle traitée. Effectivement, rien ne lie les diverses itérations de la boucle ensemble. elles peuvent donc être

```
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i];
  D[i] = A[i] + 10;
}
```

Programme 1.14 – Exemples de dépendances mémoire indépendantes de la boucle

```
for (int i = 0; i < 100; ++i) {
  A[i+1] = A[i] + B[i];
}
```

Programme 1.15 – Exemples de dépendance mémoire portée par la boucle

exécutées en parallèle.

Une fois l'analyse des dépendances mémoire effectuée, on peut procéder à l'application de diverses transformations pour générer du code parallèle. Par exemple, si la boucle est semblable à celle de l'exemple du Programme 1.14, on peut directement générer le code parallèle pour les multiples coeurs d'un processeur moderne ou les extensions vectorielles des dernières versions des jeux d'instructions des architectures les plus courantes comme *ARM* ou *x86*. Si, par contre, on rencontre une situation qui s'apparente plus à celle du Programme 1.15, il se peut que quelques transformations, tel le *loop-skewing* permettent d'exposer le parallélisme de la boucle. Aussi, et ce peu importe la situation, des transformations pour modifier la granularité du parallélisme de la boucle, tel que le *loop fusion* ou le *loop splitting*, peuvent être appliquées. L'éventail de choix des actions suite à l'analyse de dépendance est vraiment large et dépend de facteurs tel que le résultat de l'analyse et l'architecture pour laquelle le code est compilé. Pour une description plus approfondie des diverses analyses et transformations visant à optimiser les boucles d'un programme, on invite le lecteur à consulter [All02, Mid12, Mor98, Aho06].

D'un point de vue historique, cette approche à la parallélisation automatique basée sur les dépendances a vu son plein essor avec le développement du langage de programmation *High Performance Fortran* (HPF). Comme son nom l'indique, ce langage est un dérivé de Fortran. Il facilite la programmation sur systèmes à haute performance, principalement les grappes de calculs. C'est donc dans le but d'alléger

le fardeau du programmeur que bien des techniques d'analyse de dépendances et de transformation de code ont été développées, ou du moins formalisées. Ainsi, même si le développeur est conscient qu'il programme une application parallèle, il peut laisser au compilateur le soin d'effectuer une bonne partie du travail de parallélisation notamment au niveau de la distribution des données. Dans ce contexte, le programmeur n'a qu'à spécifier que quelques instructions au compilateur, c'est-à-dire déclarer quels tableaux doivent être distribués et spécifier quelles boucles devaient s'exécuter de manière parallèle, pour que ce dernier complète le travail de parallélisation.

Malheureusement, HPF ne fut pas un succès [Ken07] dans la communauté du calcul de haute performance. Ceci a fait en sorte que plusieurs de ces idées n'ont pas sorti du milieu académique pendant plusieurs années. Cette tendance a cependant commencé à changer dans la dernière décennie avec la place qu'ont pris les processeurs multi-coeurs et les appareils mobiles sur le marché. Compte tenu des opportunités et des besoins de performance requis par ces nouvelles architectures, bien des compilateurs modernes, tels GCC, Clang et MSVC, ont commencé à implémenter quelques-unes des transformations mises de l'avant par HPF pour au moins exploiter le parallélisme au niveau des instructions. Aussi, les techniques d'analyse mises à profit par HPF sont aussi utilisées dans le cadre d'outils d'aide à la parallélisation automatique pour déterminer si le compilateur peut effectivement paralléliser une partie du code annotée par un programmeur.

Finalement, même si HPF ne fut pas le succès auquel s'attendait son auteur, le travail fait pour mettre sur pied ce langage a permis d'ouvrir la voie à de nouveaux langages qui tentent de réussir là où HPF a échoué. Ainsi, dans la dernière décennie, un nouveau paradigme de programmation pour le calcul de haute performance, principalement sur des systèmes à mémoire distribuée, connu sous le nom de *Partitioned Global Address Space* (PGAS) a vu le jour. Les langages de programmation Chapel [Cha07] et X10 [Sar12] sont des exemples de tentatives d'implémenter ce paradigme. Suivant les traces de HPF, ces langages tentent d'offrir un parallélisme multi-résolution. En d'autres termes, ils présentent aux programmeurs la possibilité de choisir entre la parallélisation automatique ou manuelle d'un programme grâce aux divers éléments linguistiques qui les composent. Ils permettent aussi de créer aisément des applications pour systèmes à mémoire distribuée en offrant la possibilité de créer

```
#pragma omp parallel shared(A,B,C) private(i,j,k)
{
  #pragma omp for schedule (static)
  for(int i = 0; i < N; ++i)
    for(int j = 0; j < M; ++j)
      for(int k = 0; k < P; ++k)
        C[i][j] += A[i][k] * B[k][j];
}
```

Programme 1.16 – Multiplication de matrices à l'aide de OpenMP

des structures de données distribuées dont la partition en mémoire peut être effectuée automatiquement pour le programmeur. L'avenir nous dira si ces nouveaux langages sauront s'imposer dans le domaine du calcul de haute performance.

1.3.2 Outils pour aider la parallélisation automatique

Étant donné que la recherche sur la parallélisation automatique n'a pas encore produit de percée significative, bien des approches, autant commerciales qu'expérimentales, utilisent des directives au compilateur pour le guider dans ses tâches. On cherche par ces approches à déléguer la responsabilité de choisir la partie du programme à paralléliser au programmeur. Dans ce paradigme, le compilateur n'a qu'à effectuer les transformations qui lui sont demandées.

L'outil sans doute le plus connu pour paralléliser automatiquement un programme est OpenMP [Ope13]. OpenMP permet la parallélisation de programmes à l'aide d'annotations destinées au compilateur pour générer du code qui met en oeuvre plusieurs fils d'exécution.

Le Programme 1.16, qui effectue une multiplication de matrices, permet de mettre en relief plusieurs fonctionnalités de OpenMP. Tout d'abord, on constate qu'il est possible de créer des portées qui seront annotées par une *pragma* de OpenMP dictant leurs exécutions. De plus, il est possible avec OpenMP de décider quelles structures de données et quelles variables seront partagées entre les fils d'exécution et lesquelles seront transformées de sorte que chaque fils d'exécution aura sa propre copie. Une autre fonctionnalité qu'on peut observer est qu'il est possible de choisir l'ordonnancement

des fils d'exécution. Dans l'exemple présent, les fils d'exécution seront ordonnancés de façon statique, c'est-à-dire le compilateur divisera le travail à faire en blocs les plus égaux possibles et assignera ces blocs à chaque fil d'exécution. Il est possible d'ajouter des arguments (*dynamic*, *guided*, *auto*, *runtime*) pour modifier la façon dont le compilateur gèrera la répartition et l'ordonnancement des blocs de travail. Finalement, bien que cela ne soit pas illustré dans l'exemple du Programme 1.12, OpenMP ne se limite pas au parallélisme orienté données, il peut aussi générer du code parallèle orienté tâche avec l'annotation *task*. Le seul, et majeur, désavantage d'OpenMP dans le cadre de ce projet de maîtrise est qu'il est totalement orienté vers la parallélisation automatique pour systèmes à mémoire partagée et qu'aucune initiative pour le porter vers la génération de code pour systèmes à mémoire distribuée n'a porté fruit à ce jour.

Dans un contexte plus académique, le projet *Noise* [Kar13] vise à développer un outil permettant de contrôler quelles optimisations le compilateur devrait appliquer aux énoncés présents dans une portée qualifiée par une annotation *Noise*. Par exemple, dans le Programme 1.17, on spécifie au compilateur d'appliquer les transformations nommées *loop-fusion*, *inlining*, *loop invariant code motion*, *vectorization* et finalement *loop-unrolling* aux deux boucles comprises dans la portée définie avec la macro *Noise*. L'intérêt de ce projet est qu'au travers de tests de performance et des connaissances d'experts, le binaire généré par *Noise* a de fortes chances d'être plus optimal, en termes de vitesse ou de taille, que pourrait l'être un binaire généré en spécifiant uniquement des options standards à la ligne de commande lors de la compilation.

Le problème avec cette approche est qu'elle requiert potentiellement l'intervention d'un expert en compilation, plus particulièrement la compilation avec LLVM, pour l'utiliser. Ainsi, un développeur ne connaissant pas ou peu les optimisations qu'un compilateur peut appliquer à un programme ne pourra utiliser cet outil à son plein potentiel. De plus, cet outil est intrinsèquement lié à LLVM. Par conséquent, il ne pourrait être utilisé avec *GCC* ou *MSVC* compte tenu que ces deux compilateurs ne se servent pas de LLVM dans leur processus de compilation.

```
void foo(float x, float* in, float* out, int N) {
  NOISE("loop-fusion inline(bar) licm vectorize(8)unroll(4)")
  {
    for (int i=0; i < N; ++i) {
      float lic = x * bar(x);
      out[i] = in[i] + lic;
    }
    for (int i=0; i < N; ++i) {
      out[i] *= x;
    }
  }
}
```

Programme 1.17 – Code utilisant *Noise*

Chapitre 2

L'infrastructure de compilation LLVM

Le but de ce chapitre est de faire connaître l'infrastructure de compilation LLVM. Pour ce faire, on présente d'abord son architecture en prenant bien soin de souligner les raisons derrière ses plus importants choix de conception. Par la suite, on discute de la composante possiblement la plus importante de l'infrastructure de LLVM, soit la représentation intermédiaire utilisée pour décrire les programmes à compiler et les rendre plus facile à optimiser.

2.1 Architecture

Le but derrière la création de LLVM est d'offrir une plateforme facilitant le développement rapide de nouvelles passes d'optimisation de code dans un compilateur. Il se veut aussi une plateforme rendant beaucoup plus accessible le développement de compilateurs en fournissant des outils tels que des optimisateurs de code et des générateurs de code machine. Pour atteindre ces nombreux objectifs, LLVM présente une architecture un peu différente de ce qui était la norme dans les compilateurs au moment de son invention. La figure 2.1 illustre ce fait.

On remarque donc que LLVM n'inclut aucun *frontend*, analyseurs lexicale, syntaxique et sémantique pour un langage donné, tel que C++ ou Java, et concentre

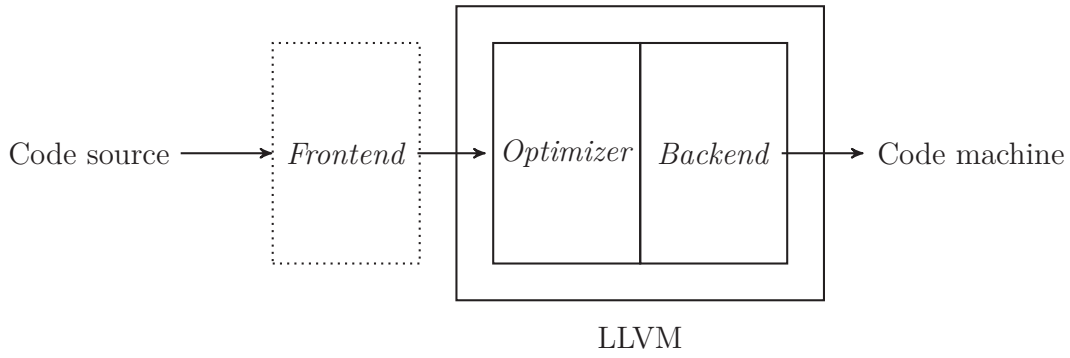

Figure 2.1 – Architecture de haut niveau de LLVM

tout son travail sur l'optimisation de code (*optimizer*) et la génération de code pour une architecture donnée (*backend*).

La façon dont fonctionne le processus de compilation avec une telle architecture est la suivante. Tout d'abord, un *frontend* pour un langage donné traite le programme à compiler de façon à produire son équivalent dans la représentation intermédiaire de LLVM. En d'autres termes, il va faire passer le code reçu au travers de phases d'analyse lexicale, syntaxique et sémantique pour produire un arbre syntaxique abstrait (ASA). Cet ASA servira ensuite à générer du code LLVM IR. Par la suite, ce code nouvellement produit est passé à l'optimisateur. Celui-ci est alors chargé d'analyser le code et, en fonction des résultats des analyses effectuées, d'exécuter des passes de transformations de code dans le but de l'optimiser de manière indépendante de l'architecture pour laquelle le code est compilé. Par après, le code sous représentation intermédiaire, maintenant optimisé, sera passé au *backend*. Ce dernier aura pour tâche d'effectuer des optimisations spécifiques à l'architecture pour laquelle le code est compilé et finalement de générer le code machine.

Comme on peut le constater, LLVM ne reçoit aucune indication quant au langage dans lequel le programme à compiler a été écrit. De plus, bien que LLVM soit distribué avec des modules de génération de code (*backend*), rien n'empêche d'en ajouter d'autres. Son travail se fait sur la base de sa représentation intermédiaire. Par conséquent, cette approche modulaire lui permet d'être une infrastructure réellement générique pour la compilation. Il ne suffit que de produire un *frontend* capable de

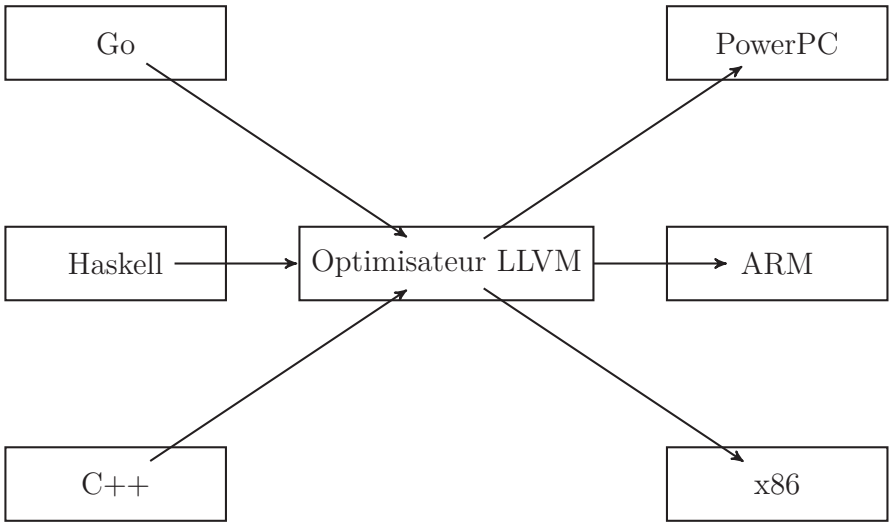

Figure 2.2 – Architecture de LLVM

prendre en charge le langage original et un *backend* pour une architecture spécifique, si LLVM n'en fournit pas déjà un, pour créer un compilateur complet avec peu d'effort. À la vue de ces faits, une façon plus juste d'illustrer l'architecture de LLVM correspond à ce qui est présenté à la figure 2.2.

2.2 Représentation intermédiaire

Cette courte introduction à LLVM permet de constater comment le choix d'une bonne représentation intermédiaire est important. À titre d'exemple, le Programme 2.1, écrit en C++, a comme représentation intermédiaire le Programme 2.2.

Comme on peut l'observer, la représentation intermédiaire du Programme 2.2, s'apparente à un langage d'assemblage. Ceci était un des buts lors de sa conception. En effet, une représentation intermédiaire sous une telle forme réduit le travail à effectuer par un générateur de code machine.

De façon générale, le format des instructions est le suivant. Tout d'abord, si l'instruction produit un résultat, une variable nommée apparaît à la gauche de l'instruction pour recevoir la valeur produite par celle-ci. Ensuite, on retrouve la mnémonique de l'instruction, suivie optionnellement de modificateurs, et une liste d'arguments,

```
int main() {
  const int N = 50;
  double tab[N];

  for (int i = 0; i < N; ++i)
    tab[i] = i * i;
}
```

Programme 2.1 – Programme en C++

```
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %N = alloca i32, align 4
  %tab = alloca [50 x double], align 16
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 50, i32* %N, align 4
  store i32 0, i32* %i, align 4
  br label %2

; <label>:2                                       ; preds = %13, %0
  %3 = load i32* %i, align 4
  %4 = icmp slt i32 %3, 50
  br i1 %4, label %5, label %16

; <label>:5                                       ; preds = %2
  %6 = load i32* %i, align 4
  %7 = load i32* %i, align 4
  %8 = mul nsw i32 %6, %7
  %9 = sitofp i32 %8 to double
  %10 = load i32* %i, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds [50 x double]* %tab, i32 0, i64 %11
  store double %9, double* %12, align 8
  br label %13

; <label>:13                                      ; preds = %5
  %14 = load i32* %i, align 4
  %15 = add nsw i32 %14, 1
  store i32 %15, i32* %i, align 4
  br label %2

; <label>:16                                      ; preds = %2
  %17 = load i32* %1
  ret i32 %17
}
```

Programme 2.2 – Programme en LLVM IR

chacun précédé d'un type. Dans le cas des opérations arithmétiques, il n'y a qu'un seul type précisé, car la représentation intermédiaire de LLVM n'opère pas sur des instructions arithmétiques aux arguments de différents types. Pour éviter d'avoir de telles situations, LLVM insère des instructions de conversion de types avant des instructions arithmétiques au besoin.

Par souci de clarté, il faut apporter quelques précisions sur les instructions nommées, c'est-à-dire, les instructions produisant une valeur que l'on affecte à une variable nommée. Ainsi, le nom d'une variable est généralement une valeur entière générée par LLVM. C'est d'ailleurs le cas de la majorité les variables de la Figure 2.3. Dans le reste des cas, le nom attribué aux variables est tiré du programme qui est compilé. Il est par contre possible pour un développeur de compilateurs de spécifier des noms particuliers pour ces variables. À titre d'exemple, l'instruction *%MulRes = mul nsw i32 %6, %7* est valide et peut remplacer l'instruction *%8 = mul nsw i32 %6, %7* de la Figure 2.3 si l'on remplace les occurrences de *%8* par *%MulRes*.

Au niveau des instructions offertes par la représentation intermédiaire, on y retrouve donc des instructions qui ont un équivalent direct sur plusieurs architectures tel que *load* et *store* qui effectuent respectivement les opérations les opérations de chargement et de stockage de données en mémoire. Les instructions arithmétiques de base, comme *add* et *mul*, font également partie de l'ensemble des instructions. La représentation intermédiaire introduit aussi de nouvelles instructions pour simplifier le travail des passes d'optimisation de LLVM. Dans le programme présent, il y en a deux: *alloca* et *getelementptr*. Ces dernières sont utilisées pour déclarer une variable sur la pile et accéder à un élément donné d'une structure (tableau, classe, etc.). Finalement, le lecteur peut remarquer que la représentation intermédiaire est typée. Ceci donne d'ailleurs lieu dans le Programme 2.2 à la nécessité d'utiliser une instruction *sext* pour étendre un type de 32 bits à 64 bits et une instruction *sitofp* pour obtenir une valeur à virgule flottante en double précision à partir d'une valeur entière. Le nombre d'instructions de la représentation intermédiaire de LLVM étant relativement élevé, on invite le lecteur à consulter [Fou14] pour une liste exhaustive.

Une autre caractéristique de la représentation intermédiaire est qu'un programme écrit dans cette représentation est divisée en blocs de base. Ainsi, au Programme 2.2, on constate que les instructions sont regroupées d'une telle façon qu'une instruction

de branchement *br* termine toujours un groupe. Ces blocs ont aussi tous une étiquette qui est utilisée comme destination par les instructions de branchement. En réalité, la représentation intermédiaire de LLVM est autant un langage d'assemblage de haut niveau typé qu'une vue sur le CFG d'un programme donnée. En d'autres termes, le Programme 2.2 est en tout point équivalent au graphe de la Figure 2.3.

Comme il a déjà été mentionné, l'utilisation d'une telle représentation intermédiaire permet de rendre modulaire le processus de compilation aidant ainsi à offrir une plateforme de compilation réellement générique. Outre ces avantages de haut niveau, la représentation intermédiaire offre aussi de nombreux autres bénéfices aux auteurs de compilateurs. Cependant, avant d'entrer dans les détails de ces bénéfices, il faut introduire la forme SSA, forme de la représentation intermédiaire de LLVM.

2.2.1 Forme SSA

La *Single Static Assignment Form* (ci-après abrégé par forme SSA) est une forme que peut prendre la représentation intermédiaire d'un programme lors de sa compilation. Comme son nom l'indique, cette forme est caractérisée par le fait qu'une variable ne peut se voir affecter une valeur qu'une et une seule fois. De plus, chaque variable dans cette représentation doit être définie avant de pouvoir être utilisée. D'une certaine façon, une représentation intermédiaire sous forme SSA est comparable à la programmation fonctionnelle.

À la base, la représentation intermédiaire utilisée par LLVM est sous forme SSA pour toutes les valeurs scalaires contenues dans des registres. Elle ne l'est donc pas pour tout ce qui touche à la mémoire. Le Programme 2.2 présentée précédemment est un exemple de programme en LLVM IR qui n'est pas totalement sous forme SSA. Pour qu'il le soit, il faut déplacer les valeurs contenues en mémoire pour qu'elles soient dorénavant contenues dans des registres. Le résultat d'une telle transformation est donné par le Programme 2.3.

Le point à souligner dans la Figure 2.3 est l'apparition d'une nouvelle instruction: *phi*. Cette dernière a pour fonction de régler le problème des variables dont la valeur est dépendante d'un choix (par exemple, une variable dont la valeur déterminé par l'utilisation d'un *if*). Ainsi, selon le bloc duquel le flux du programme arrive, l'ins-

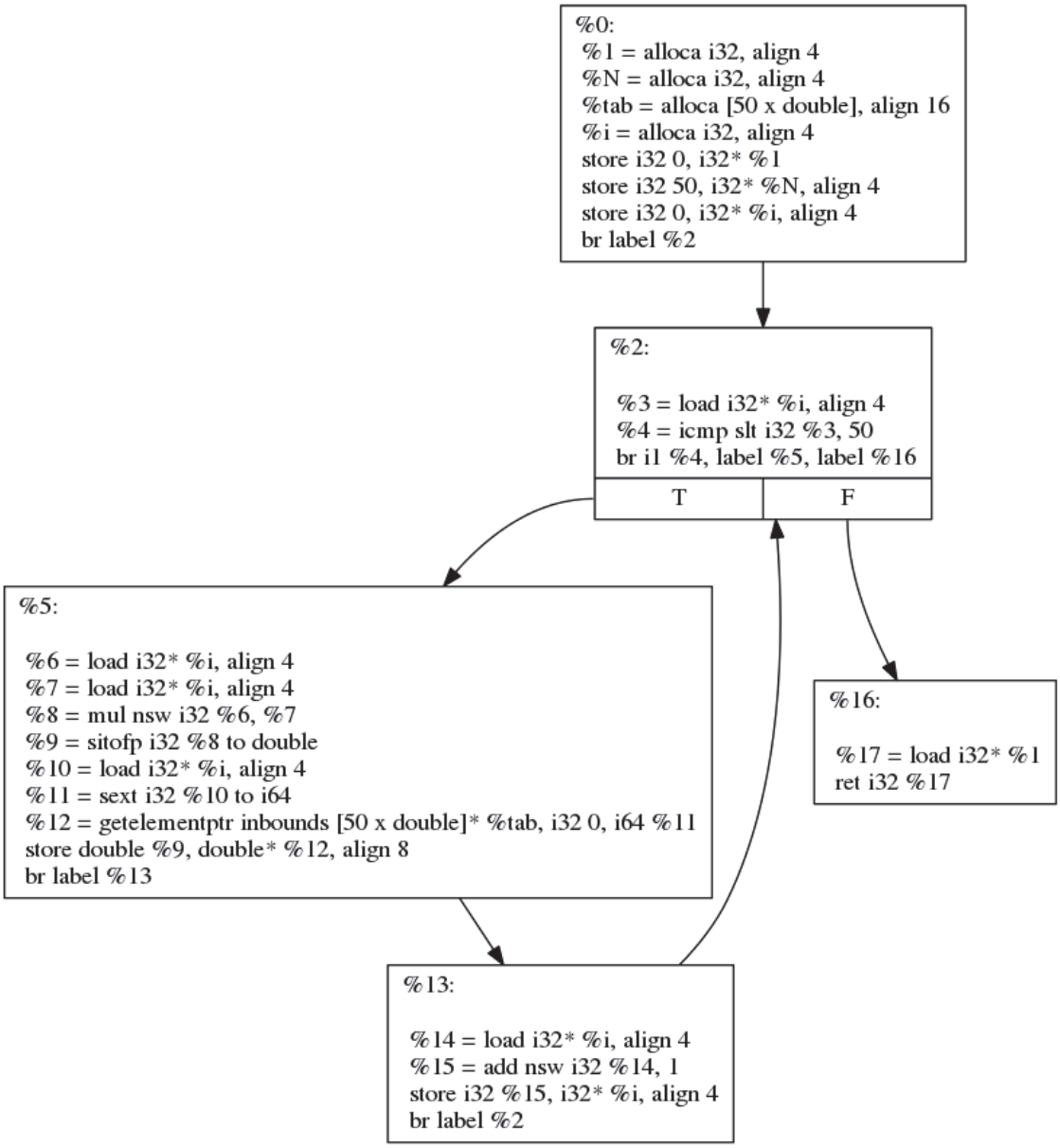

Figure 2.3 – Graphe de flot de contrôle

truction *phi* produira la valeur correspondante qui pourra ensuite être affectée à une variable.

```
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %tab = alloca [50 x double], align 16
  br label %1

; <label>:1                                       ; preds = %8, %0
  %i.0 = phi i32 [ 0, %0 ], [ %9, %8 ]
  %2 = icmp slt i32 %i.0, 50
  br i1 %2, label %3, label %10

; <label>:3                                       ; preds = %1
  %4 = mul nsw i32 %i.0, %i.0
  %5 = sitofp i32 %4 to double
  %6 = sext i32 %i.0 to i64
  %7 = getelementptr inbounds [50 x double]* %tab, i32 0, i64 %6
  store double %5, double* %7, align 8
  br label %8

; <label>:8                                       ; preds = %3
  %9 = add nsw i32 %i.0, 1
  br label %1

; <label>:10                                      ; preds = %1
  ret i32 0
}
```

Programme 2.3 – Programme en LLVM IR avec noeuds PHI exposés

2.2.2 Bénéfices

Utiliser une représentation intermédiaire de forme SSA a gagné en popularité dans les dernières années à cause des nombreux bénéfices qu'elle offre. Tout d'abord, la forme SSA simplifie grandement plusieurs passes d'analyse que le compilateur a à effectuer. Par exemple, le problème d'analyse d'activité des variables devient trivial lorsque si une variable ne peut avoir qu'une valeur lors de son existence. De plus, la forme SSA améliore aussi des passes d'optimisation en simplifiant autant leur écriture que le travail qu'elles ont à effectuer. Par exemple, la propagation de constantes [All02], une passe dans laquelle les constantes d'un programme sont identifiées et évaluées dans le but de les éliminer, est une passe bien plus simple dans le cadre de la forme SSA, car l'utilisation des constantes se trouve a y être bien plus évidente. Bref, cette forme de représentation intermédiaire offre bien des avantages au niveau de l'analyse et de l'optimisation du code.

Chapitre 3

Tentatives de solution

Comme il a déjà été mentionné, le but de de travail de recherche était de produire un outil permettant de paralléliser automatiquement des programmes scientifiques écrits de façon séquentielle. Plus précisément, il consistait à explorer la possibilité de paralléliser automatiquement un réseau de neurones sur une grappe de calcul. Avant de présenter une solution acceptable dans le prochain chapitre, deux essais infructueux font l'objet du présent chapitre. La section 3.1 décrit une première expérience avec la plateforme de compilation polyédrale *Polly* pour paralléliser automatiquement des programmes sur un système distribué. Ensuite, la section 3.2, fait était d'une deuxième expérience qui consistait à étendre l'approche à la parallélisation automatique, dénommée *Decoupled Software Pipelining*, à des systèmes distribués.

3.1 Modèle polyédral

Cette section contient une description de la plateforme de compilation polyédrale Polly, donne des détails sur une extension possible de cette plateforme lorsque l'architecture cible est un système distribué et ainsi que les raisons qui ont conduit à un échec.

3.1.1 Polly

Polly [Gro11] est une plateforme de compilation polyédrale intégrée à l'infrastructure de compilation LLVM. Comme l'illustre la Figure 3.1, le processus de compilation de *Polly* débute par la découverte de SCoP dans un programme exprimé en LLVM IR. Une fois les SCoPs identifiés, *Polly* extrait d'eux une représentation polyédrale, une représentation constituée du domaine, de l'ordre d'exécution et des accès mémoire des énoncés compris dans le SCoP traité. Une fois cette représentation polyédrale extraite, *Polly* est libre d'appliquer diverses transformations et optimisations sur le SCoP traité. Lorsque ce processus d'optimisation est terminé, *Polly* retransforme le SCoP traité en sa représentation intermédiaire LLVM. Cette façon de faire permet à *Polly* d'effectuer des transformations et optimisations du code indépendantes de l'architecture pour laquelle le programme final est destiné. À cette étape, il ne génère pas de code machine. Aussi, en opérant sur une représentation intermédiaire, il devient applicable à de nombreux programmes écrits dans une variété de langages de programmation en autant qu'il existe pour chacun de ces langages un outil qui traduit tout programme en un programme LLVM IR.

3.1.2 Approche retenue

L'approche retenue dans cette partie expérimentale consistait à permettre la parallélisation automatique, dans la plateforme Polly, de programmes scientifiques pour exécution sur un système distribué. Comme il a déjà été mentionné lors de la présentation de la compilation polyédrale dans le chapitre précédent, *Polly* permet de paralléliser automatiquement des programmes sur des systèmes à mémoire partagée en générant du code mettant à profit *OpenMP* [Rag11]. En effet, *Polly* est capable de déterminer si un SCoP donné est parallélisable. L'attention a donc été portée sur la répartition équitable des calculs à effectuer sur les processeurs disponibles. La transformation à appliquer devrait regrouper les instances concrètes des énoncés d'une boucle de façon à créer des tâches dont la granularité fait en sorte qu'elles méritent d'être exécutées sur un système à mémoire partagée. Une solution envisageable consistait à partitionner le SCoP en utilisant une transformation nommée *loop tiling* [Mid12, All02]. Cette transformation a pour effet de partitionner les itérations dans

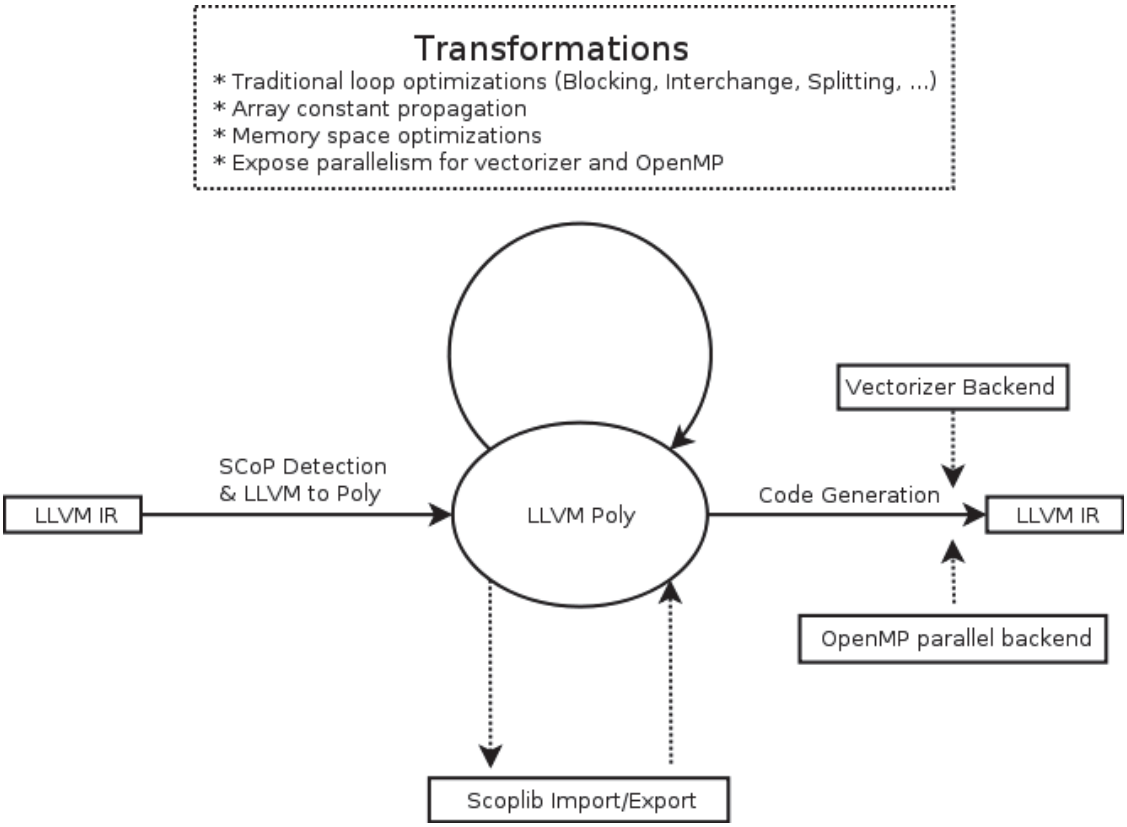

Figure 3.1 – Architecture de *Polly* [Gro11]

un imbriquement de boucles en blocs dans le but d'améliorer la localité des accès mémoires. Par exemple, appliquer la transformation *loop tiling* au Programme 3.1 produit le Programme 3.2.

Une représentation visuelle du programme transformé est aussi offerte à la Figure 3.2. On remarque que les itérations, représentées par des points noirs, sont maintenant divisées en des blocs de taille $M \times M$.

L'intuition qu'il faut avoir ici est que les blocs produits lors de la transformation feront office de tâches à exécuter dans le programme parallèle qui à la fin du processus de parallélisation automatique. Cette transformation permet donc d'atteindre un des buts fixés, soit créer des tâches à forte granularité. En effet, sans cette transformation, il est envisageable que *Polly* produise une version parallèle du programme dans laquelle chaque itération d'une boucle soit exécutée sur un processus différent. Cette

```
for (int i = 1; i < N; ++i)
  for (int j = 1; j < N; ++j)
    A[i][j] = A[i+1][j] + A[i+1][j+1];
```

Programme 3.1 – Imbriquement de boucles avant le *loop tiling*

```
for (int i = 1; i < N; i+=M)
  for (int j = 1; j < N; j+=M)
    for (int ii = i; ii < i+M; ++ii)
      for(int jj = j; jj < j+M; ++jj)
        A[ii][jj] = A[ii+1][jj] + A[ii+1][jj+1];
```

Programme 3.2 – Imbriquement de boucles après le *loop tiling*

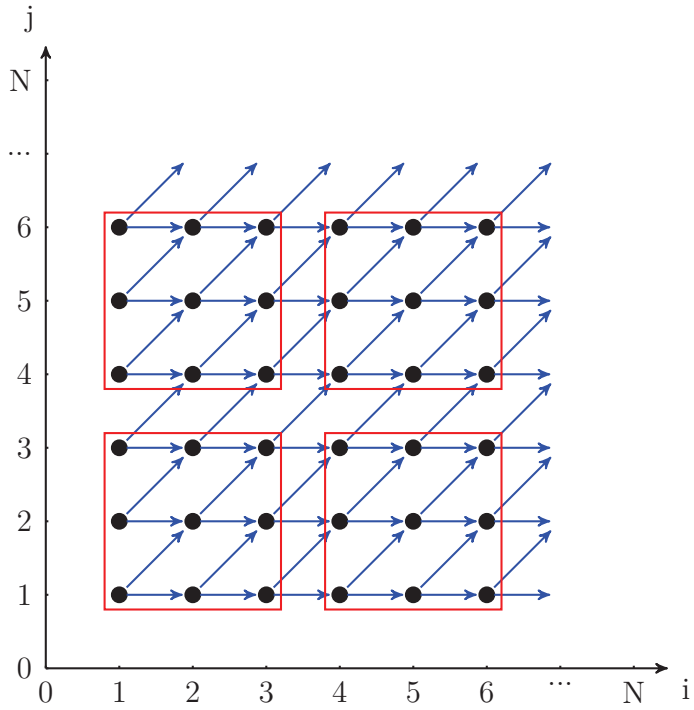

Figure 3.2 – Espace d'itérations après l'application de la transformation *loop tiling*

situation possible n'est pas souhaitable, car la quantité de calcul exéctué par chacune des itérations de la boucle n'est pas suffisamment grand pour justifier l'utilisation d'un processus dédié uniquement à cette fin.

Une fois cette transformation appliquée, la prochaine étape consiste à introduire des appels de fonctions MPI pour assurer la distribution des données. C'est ici qu'une autre des raisons derrière le choix d'appliquer la transformation *loop tiling* devient apparente. En effet, en observant la Figure 3.2, on peut remarquer que les seules données qui doivent réellement être transmises entre les processus sont celles impliquées dans une dépendance mémoire qui traverse les frontières délimitées par les blocs précédemment créés. Sachant quelles sont ces données, les éléments de tableau, le travail ne consisterait plus qu'à générer des structures de données auxiliaires pour les transmettre entre processus et générer des appels aux fonctions MPI pertinentes.

Malgré les idées énoncés dans cette sous-section, cette proposition de solution est restée à un stade théorique. En effet, plusieurs problèmes, explicités à la sous-section suivante, nous ont empêché de progresser dans cette direction. Ces problèmes sont autant d'ordre théorique que pratique.

3.1.3 Problèmes liés à l'approche

L'idée de mettre en oeuvre la parallélisation automatique de programmes séquentiels à l'aide de la compilation polyédrale n'a pas été prometteuse à cause de plusieurs problèmes. Le premier de ces problèmes est que le modèle polyédral de compilation est un modèle très récent et encore en pleine exploration. Bien que des essais aient déjà été effectués au niveau de la parallélisation automatique pour systèmes à mémoire distribué [Red14, Bon13, Bon11], il n'en demeure pas moins que peu de succès ont été obtenus à ce jour. Tenter d'utiliser le modèle polyédral dans ce contexte est un projet nécessitant potentiellement plusieurs années. Pour expliciter ce point, l'utilisation du modèle polyédral implique de faire face à deux problèmes d'envergure: surmonter les limitations présentes dans le modèle dans son état courant et trouver des tactiques de distribution de données dans le contexte des systèmes à mémoire partagée.

Comme il a déjà été mentionné au chapitre précédent, le modèle de compilation polyédral est limité dans ce qu'il peut traiter et optimiser. De fait, toute boucle ou imbriquement de boucles ne respectant pas les conditions nécessaires pour former un SCoP ne peut être traité. Ceci proscrit déjà bien des boucles *while* pour lesquelles, même si du point de vue du programmeur elles présentent des opportunités de pa-

rallélisation, le compilateur ne peut produire de boucles *for* équivalentes. De plus, le modèle polyédral est soumis au même problème d'analyse de pointeurs que les approches plus traditionnelles d'analyse de dépendances. Par conséquent, bien des boucles, notamment celles opérant avec des itérateurs ou travaillant sur des tableaux dynamiques, ne peuvent être parallélisées.

En plus des limites intrinsèques au modèle polyédral de base, la parallélisation automatique pour systèmes distribués doit aussi faire face au défi qu'est la distribution efficace des données. Bien que des approches pour la parallélisation de calcul sur systèmes à mémoire distribuée aient déjà été proposées dans le cadre de la compilation polyédrale [Tom13], la seule, à notre connaissance, qui s'est vraiment attardée au problème de la distribution optimale des données est [Red14]. Dans cet article, les auteurs tentent de reproduire un algorithme de distribution automatisée de données qui avait été d'abord développé dans le cadre de recherche visant à améliorer le langage High Performance Fortran. En d'autres termes, tenter de créer un outil permettant à la fois la parallélisation du calcul et la distribution intelligente de données dans le contexte du modèle de compilation polyédrale dépasse largement le cadre d'un projet de maîtrise considérant l'investissement exigé en temps.

Un autre problème important rencontré est que *Polly* est une plateforme encore relativement jeune. Pour être plus précis, *Polly* est encore immature au niveau du nombre de transformations qu'il peut appliquer à un programme sous représentation polyédrale et au niveau des informations qu'il peut acquérir à partir d'une analyse des SCoPs qu'il a identifiés. Plus concrètement, *Polly* est incapable de produire des informations détaillées sur la distance entre les dépendances mémoire, surtout dans le cas de tableaux multidimensionnels. On peut donc savoir qu'une boucle est potentiellement parallélisable, mais sans plus. Ceci a pour conséquence directe de limiter les transformations qu'il peut effectuer sur un programme ainsi que d'ajouter un niveau de difficulté supplémentaire pour tout développeur désirant créer une nouvelle transformation. Par exemple, la transformation appelée *loop tiling* [All02] ne peut tout simplement pas être réalisée dans le contexte de *Polly*. Modifier Polly pour permettre cette transformation nécessiste un effort considérable et sans cette modification, il est impossible de réaliser dans un temps raisonnable la solution initialement envisagée. Elle a donc été abandonné. Au moment d'écrire ses lignes, *Polly* présente encore ces

faiblesses, ses auteurs ayant choisi de concentrer leurs efforts sur d'autres aspects de la plateforme dans leur plan de travail.

3.1.4 Solution potentielle

La solution potentielle qui semble la plus prometteuse est de ne pas tenter d'implémenter la parallélisation pour système à mémoire distribuée à même *Polly*. De fait, *Polly* peut être utilisé en tant qu'outil à part entière de par sa nature en tant que passe d'optimisation (dans le sens large du terme) au sein de l'infrastructure de compilation LLVM. En tenant compte de ce fait, il est concevable de bâtir un outil de parallélisation automatique ayant comme cible des systèmes distribués, tout d'abord, en utilisant *Polly* pour effectuer des analyses et des transformations et, ensuite, en exploitant un programme spécifique de génération de code pour des systèmes distribués. Cette idée a été concrétisé dans le projet *Molly* [Kru14] qui exploite la plateforme *Polly* pour extraire les dépendances polyhédrales présentes dans les SCoPs d'un programme. Pour réaliser cette tâche, des annotations de programme, de nouvelles instructions intrinsèques à LLVM et de nouvelles structures de données à être utilisées dans le langage de programmation C++ sont ajoutées. Le Programme 3.3 permet d'illustrer cette approche. Dans cet exemple, on peut observer que les structures utilisées, les tableaux *front* et *back*, sont issues de *Molly*. En effet, ces structures sont préférées aux structures usuelles, car elles ont été implantées de manière à être facile à analyser par *Molly*. De plus, on peut constater l'utilisation de l'annotation de fonction *[[molly::pure]]*. Cette dernière a pour but de préciser que la fonction ne causera aucun effet de bord ce qui simplifie le travail d'analyse de *Molly*. L'ensemble de ces ajouts, structures et annotations pour programmes et passes de transformations pour le compilateur, font donc en sorte que la parallélisation de ce programme sur une grappe de calcul devient possible.

```
#include <molly.h>

[[molly::pure]]
bool hasLife(bool hadLife, int neighbors) {
  if (hadLife)
    return 2 <= neighbors && neighbors <= 3;
  return neighbors == 3;
}

molly::array<bool,1024,1024> front;
molly::array<bool,1024,1024> back;

int main() {
  for (int i = 0; i < 100; i+=1) {
    for (int x=1,w=front.length(0); x < w-1; x+=1)
      for (int y=1,h=front.length(1); y < h-1; y+=1) {
        auto neighbors = front[x-1][y] + front[x][y+1];
        neighbors += front[x+1][y] + front[x][y-1];
        auto living = hasLife(front[x][y], neighbors);
        back[x][y] = living;
      }
    for (int x=1,w=back.length(0); x < w-1; x+=1)
      for (int y=1,h=back.length(1); y < h-1; y+=1)
        front[x][y] = back[x][y];
  }
  return EXIT_SUCCESS;
}
```

Programme 3.3 – Jeu de la vie de Conway avec *Molly*

3.2 Parallélisation automatique par détection de dépendances

À la suite de cet échec, nous avons considéré des approches plus traditionnelles au niveau de la parallélisation automatique. Plus particulièrement, nous nous sommes intéressés aux méthodes basées sur la détection de dépendances notamment une méthode relativement récente appelé *Decoupled Software Pipelining* (DSWP). Cette section introduit tout d'abord les bases du DSWP. Ensuite, elle présente une extension de cette méthode au domaine des systèmes à mémoire distribuée. Enfin, elle expose les problèmes rencontrés et les avenues à emprunter pour les régler dans un autre contexte.

3.2.1 *Decoupled Software Pipelining*

Pour cette solution, la mise en oeuvre d'un outil qui effectue la parallélisation automatique de programmes ayant comme cible des systèmes distribués est inspiré de l'algorithme appelé *Decoupled Software Pipelining* (DSWP) [Ott05] ainsi que de ses variations [Li,, Ram08]. Cette famille d'algorithmes propose de diviser un programme présentant des opportunités de parallélisation de manière à créer des tâches qui communique entre elles qu'à des moments bien précis. Pour y arriver, on extrait d'abord le graphe de dépendances du programme. Ensuite, à l'aide de phases successives de regroupement de noeuds, des tâches parallèles sont créées. Par après, le code machine est généré en utilisant une bibliothèque permettant la manipulation de fils d'exécution et en prenant soin d'ajouter des appels à des fonctions de synchronisation là où elles sont nécessaires. Étant donné que ces algorithmes ont connu du succès pour la parallélisation automatique de programmes sur systèmes à mémoire partagée, l'objectif était de les étendre afin qu'ils puissent également effectuer la parallélisation automatique ayant comme cible les systèmes à mémoire distribuée.

3.2.2 Idée proposée

L'approche proposée, qui est inspirée de l'algorithme de base de DSWP, suppose que, dans un système distribué, le rapport entre le temps de communication et le

```
void while_func() {
  int i = 0;
  while (i < 100)
    ++i;
}
```

Programme 3.4 – Code original C++

temps de calcul est petit, sans quoi l'utilisation d'un système à mémoire distribuée perd son intérêt. Donc, au lieu de produire du code parallèle orienté données, le but est de créer du code parallèle orienté tâches. Pour illustrer cet algorithme, cette sous-section présente à l'aide d'un exemple le traitement qu'il effectue. Cet exemple est basé sur le Programme 3.4 plus spécifiquement sous sa représentation en LLVM IR, c'est-à-dire le Programme 3.5.

La première étape dans l'algorithme proposé consiste à construire le graphe de dépendances du programme (PDG). En d'autres termes, on veut construire une structure indiquant quelles instructions dans la représentation intermédiaire du programme possèdent des dépendances entre elles. Le résultat visé est illustré à la Figure 3.3. Ici, il faut préciser que les liens en vert représentent des dépendances mémoire et les liens en pointillés représentent des dépendances de contrôle. Pour obtenir ce résultat, il faut d'abord identifier les dépendances mémoire. Ceci est fait en se servant d'une structure de données appelée *def-use chain*. Cette structure est en fait une simple liste liant ensemble l'instruction définissant une variable dans la représentation intermédiaire avec les instructions l'utilisant. On constate donc que c'est une structure appropriée pour représenter les dépendances mémoire impliquant une écriture et une lecture.

Pour ce qui est des dépendances de contrôle, l'algorithme utilisé pour les découvrir est celui présenté dans [Fer87]. Brièvement, cet article propose d'utiliser une structure de données appelée arbre des post-dominateurs pour déterminer quels blocs de base ne sont que conditionnellement exécutés, c'est-à-dire, quels blocs peuvent ne pas être exécuter si un branchement ne conduit pas à ces blocs. Une fois ces blocs identifiés, une dépendance de contrôle est indiquée entre leurs instructions et le branchement qui réfère à un tel bloc.

Une fois le PDG construit, on peut commencer à le raffiner dans le but de dégager

```
; ModuleID = 'test.cpp'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define void @_Z10while_funcv() #0 {
  %i = alloca i32, align 4
  store i32 0, i32* %i, align 4
  br label %1

; <label>:1                                       ; preds = %4, %0
  %2 = load i32* %i, align 4
  %3 = icmp slt i32 %2, 100
  br i1 %3, label %4, label %7

; <label>:4                                       ; preds = %1
  %5 = load i32* %i, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* %i, align 4
  br label %1

; <label>:7                                       ; preds = %1
  ret void
}

attributes #0 = { nounwind uwtable
"less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false"
"no-nans-fp-math"="false" "stack-protector-buffer-size"="8"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.6.0 "}
```

Programme 3.5 – Code en représentation intermédiaire LLVM IR

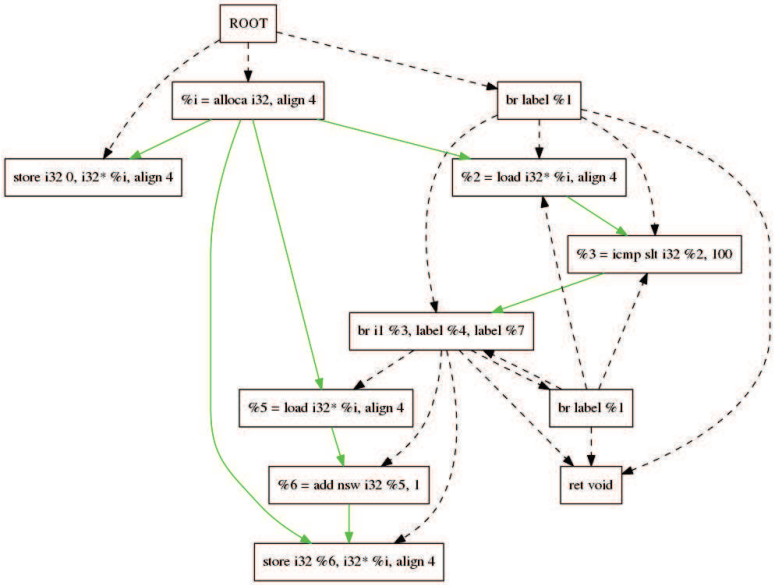

Figure 3.3 – Graphe de dépendance du programme

des tâches à exécuter de manière parallèle. La première étape consiste à regrouper ensemble les instructions constituant une composante fortement connexe. Rappelons qu'une composante fortement connexe dans un graphe est un sous-graphe dans lequel il existe un chemin entre tout couple de sommets. Dans le contexte présent, on doit comprendre qu'une composante fortement connexe correspond à un ensemble d'instructions dont les dépendances entre elles font en sorte qu'elles ne pourront être exécutées d'une autre façon que séquentielle. Il faut préciser que ce sont autant les dépendances mémoire que les dépendances de contrôle qui sont impliquées dans la détection de composantes fortement connexes. Le résultat de cette première étape de création de tâches est illustré par la Figure 3.4. On remarque la présence d'un noeud étiqueté SCC (de l'anglais *Strongly Connected Component*) qui vient remplacer un ensemble d'instructions qui forment une composante fortement connexe dans le graphe de la Figure 3.3.

La prochaine étape dans le processus de parallélisation automatique consiste à appliquer au graphe obtenu à l'étape précédente une transformation appelée fusion typée [Ken94]. Brièvement, cette transformation consiste à regrouper ensemble des noeuds dans un graphe de dépendances s'ils sont considérés comme ayant le même

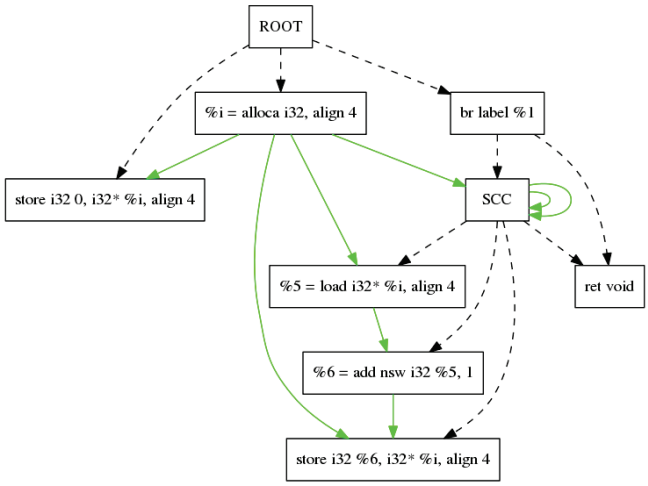

Figure 3.4 – Graphe de dépendances du programme avec composantes fortement connexes fusionnées

type. Dans le cas présent, on considère que deux noeuds sont du même type s'ils possèdent exactement les mêmes dépendances de contrôle. L'intuition est que chaque boucle ou imbrication de boucles dans un programme peut potentiellement être considérée comme étant une tâche à part entière dans la version parallèle du programme que l'on tente de produire. De plus, l'algorithme de fusion utilisé présente des conditions supplémentaires pour décider de la fusion de deux noeuds donnés. En effet, si un des noeuds est un noeud créé par la fusion de noeuds constituant une composante fortement connexe, la fusion n'est pas effectuée. On veut de cette manière s'assurer de maximiser le parallélisme extrait en faisant attention à ne pas forcer une trop grande partie du programme à s'exécuter séquentiellement. Également, on empêche la fusion de deux noeuds candidats qui causerait la création d'une nouvelle composante fortement connexe dans le graphe.

Une fois le graphe fusionné obtenu, les prochaines étapes consiste en la distribution des données et la génération de code machine augmenté d'appels à des fonctions MPI. Cependant, ces tâches furent autant la cause de nombreux problèmes que les victimes de problèmes déjà présents dans les étapes précédentes. En effet, on peut observer dans les Figures 3.4 et 3.5 que l'outil développé n'est pas en mesure de faire un regroupement intelligent des instructions qui exposerait le parallélisme sous-jacent

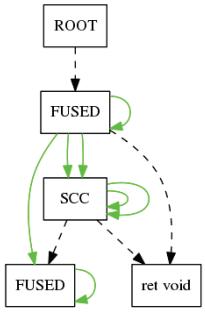

Figure 3.5 – Graphe de dépendances du programme après fusion

d'un programme qui présente réellement des opportunités de parallélisation.

3.2.3 Problèmes liés à l'approche

Le principal problème lié à cette approche est le fait que la détection de dépendances offerte par l'infrastructure LLVM est très délicate à utiliser et souvent ne donne pas facilement les informations escomptées. Ce problème a été rencontré à deux reprises. Une première fois dans la passe d'analyse de dépendances d'une boucle et dans la passe d'analyse mémoire. Une deuxième fois dans l'utilisation de *use-def chain*. Dans le premier cas, les résultats étaient souvent limités dès que l'on dépassait des cas triviaux. En effet, ces passes ne semblaient pas être en mesure de fournir des informations pertinentes sur les dépendances présentes dans la partie du code à paralléliser. Effectivement, dans bien des cas, les dépendances mémoire étaient bien identifier. De plus, pour obtenir ces résultats, de nombreuses passes d'analyse et de transformations visant à simplifier le code devaient être effectuées. En d'autres termes, beaucoup d'efforts devait être mis en oeuvre pour au final bien peu de résultats.

Dans un second cas, une *def-use chain* permet de découvrir les dépendances dans un programme donné, plus particulièrement les dépendances au niveau des boucles. Une *def-use chain* est une structure de données mettant en relation une instruction LLVM IR définit une variable à un ensemble d'instructions LLVM IR qui utilise cette variable. L'idée d'utiliser une telle structure provient du fait que la représentation intermédiaire de LLVM est sous forme SSA et donc permet de mettre en relief les diverses dépendances à divers endroits dans le programme. Le problème avec cette

idée est que le graphe de dépendances produit à l'aide de *def-use chains* contenait beaucoup trop de liens entre ces noeuds résultant en peu de, voir aucune, possibilités de parallélisation alors qu'il en existaient réellement. De plus, outre la difficulté de paralléliser le calcul en se basant sur des *def-use chains*, l'utilisation de cette structure ne règle pas le problème posé par la distribution de données. De fait, cette structure ne donne aucune information sur la distance entre les dépendances. Ceci pose un problème important, car, dans le processus de parallélisation automatique, il faut savoir si une dépendance empêche la parallélisation si elle est portée par la boucle. Bref, bien que LLVM soit une plateforme relativement mature pour le développement de compilateurs et d'optimisateurs, ce qu'il propose au niveau des boucles, autant pour leur analyse que leur transformations reste encore à ce jour limité pour des besoins plus avancés tel que la parallélisation automatique pour systèmes à mémoire distribuée.

3.2.4 Bilan

Ce chapitre a mis en évidence comment il était difficile de mettre en oeuvre des solutions satisfaisantes de parallélisation automatique de programmes, en particulier s'ils s'exécutent sur des systèmes distribués. D'une part, une des avenues étudiées, mais rapidement mise de côté, est basée sur l'enrichisssement d'un langage de programmation séquentielle qui n'a pas été conçu au départ pour supporter la parallélisation automatique de programmes. D'autre part, la revue de la littérature indique qu'il existe des projets en cours, tel celui qui porte sur le développement du langage de programmation parallèle Chapel [Cha07], qui tentent de résoudre le problème de la distribution intelligente et potentiellement automatique des données tout en facilitant le développement d'applications hautement parallèles. Elle a également mise en valeur des efforts consacrés à des solutions qui suggèrent de donner davantage de contrôle aux programmeurs sur la répartition et l'exécution de leurs applications. Par exemple, le langage de programmation Halide [Rag13] est une solution qui propose de découpler l'algorithme à exécuter de la façon dont il doit être exécuté. Nous aurions pu emprunter les approches proposées dans Chapel et Halide, mais l'investissement en temps était majeur. Nous avons plutôt opté pour une autre solution, celle d'utili-

ser des annotations pour diriger le processus de parallélisation automatique. Elle est présentée au chapitre suivant.

Chapitre 4

Parallélisation par annotations

Ce chapitre présente une troisième approche pour transformer un programme écrit originalement de façon séquentielle en un programme mettant à profit le fait qu'il sera exécuté sur un système distribué, plus particulièrement sur une grappe de calcul. La section 4.1 propose un nouvel outil, montre comment il s'intègre à l'infrastructure LLVM et au compilateur Clang, et donne des indications relatives à son utilisation. La section 4.2 explique comment ce nouvel outil parallélise automatiquement un programme. Plus spécifiquement, elle détaille la préparation de code LLVR IR et sa transformation, ainsi que la réalisation des transformations au niveau de la représentation intermédiaire.

4.1 Vue d'ensemble

Cette section présente une description de haut niveau de l'outil permettant la parallélisation automatique de programmes pour systèmes distribués, nommé *Clang-MPI*. Elle débute par une présentation de l'architecture de l'outil en prenant soin de bien expliciter comment il intervient dans le processus de compilation du compilateur *Clang*. Ensuite, elle montre, à l'aide d'une série d'exemples d'utilisation de l'outil, dans quels cas il est utile. Finalement, elle indique les limitations de cet outil et certaines avenues qui pourraient être empruntées pour lever ces limitations.

4.1.1 Architecture

L'outil *Clang-MPI* s'avère en fait être une variante du compilateur pour langages de la famille de C de LLVM, *Clang* [Fou14]. La différence entre la version originale de *Clang* et la version que développée dans le cadre de ce mémoire est qu'elle est capable de prendre en charge un nouveau *pragma* qui permet de spécifier la façon dont les instructions constituant une boucle seront modifiées pour tirer profit du fait que le programme s'exécutera sur un système à mémoire distribué tel qu'une grappe de calcul. Pour être plus précis, *Clang-MPI* permet la transformation d'une boucle ou d'un imbrication de boucles avec l'annotation *pragma clang loop distribute(arg)*. L'argument *arg* peut prendre deux valeurs soit *master* ou *worker*. Le lecteur devinera ici que les programmes générés par le processus de parallélisation automatique suivent le paradigme maître-esclave. La sous-section suivante donne un aperçu des effets de l'utilisation de ces annotations peut avoir sur un programme à haut niveau.

Au-delà de ce qui est apparent à l'utilisateur, *Clang-MPI* consiste aussi en une nouvelle passe au niveau de l'infrastructure LLVM. Comme il a déjà été mentionné, le compilateur *Clang* exploite LLVM dans son processus de compilation. Dans le but de profiter de l'énorme travail déjà accompli pour rendre LLVM un outil de qualité industrielle, il a été décidé que l'application des transformations découlant de l'usage des annotations introduites dans *Clang-MPI* se ferait au niveau de LLVM. Par conséquent, les annotations ne servent qu'à ajouter des métadonnées au niveau des boucles lors de la procédure de transformation d'un programme écrit en C/C++ en un programme écrit en LLVM IR. Par la suite, la génération de code machine est faite en utilisant des outils de LLVM. Une description plus étoffée de ce processus de compilation est donnée à la section suivante.

4.1.2 Exemple d'utilisation

Dans le but d'illustrer le travail qu'effectue *Clang-MPI*, trois exemples de transformations sont présentés, un pour l'argument *master* et deux pour l'argument *worker*. Dans chaque cas, deux programmes sont présentés. Tout d'abord, on présente un bout de code dans lequel une annotation est présente. Par la suite, le même programme est présenté dans sa forme transformé pour une exécution sur un système à mémoire

```
void init(int N, int A[]){
#pragma clang loop distribute(master)
    for (int i = 0; i < N; ++i)
      A[i] = i;
}
```

Programme 4.1 – Exemple d'utilisation du pragma *master*

distribuée. En d'autres termes, on présente une version MPI du programme équivalente à sa version originale. Il est important de rappeler que *Clang-MPI* n'est pas un outil de transformation de programme source à source et donc la version MPI du programme n'est présenté que pour illustrer le code en LLVM IR que *Clang-MPI* produit à partir d'un programme annoté.

Le premier exemple d'utilisation, donné par le Programme 4.1, constitue un dans lequel le programmeur souhaite que le travail d'initialisation d'un tableau A soit fait sur le noeud maître et nulle part ailleurs. Pour ce faire, la boucle où l'initialisation est effectuée est annotée avec *pragma clang loop distribute(master)*.

La façon d'identifier un processus maître avec MPI est de se baser sur le rang des processus démarrés au lancement de MPI. Parmi ceux-ci, on en désigne un, le plus souvent le processus 0, comme étant celui dont les tâches principales concernent la gestion des processus plutôt qu'au calcul à effectuer par le programme. Suivant cette manière de faire, *Clang-MPI* désigne le processus 0 comme étant le processus maître. Par conséquent, il isole les boucles annotées avec l'argument *master* de façon à ce qu'elles n'exécutent lorsque le rang du processus, tel que retourné par la fonction *MPI_Comm_rank*, correspond à celui désigné comme maître. Le résultat final de ce processus peut être observé au Programme 4.2.

L'illustration de l'utilisation de l'argument *worker* nécessite deux exemples étant donné qu'il peut être utilisé dans deux cas de figure. Ainsi, il peut être autant utilisé pour une boucle simple que pour un imbrication de boucles. Dans le cas d'une boucle simple, l'utilisation de l'annotation est aussi simple que l'utilisation de l'annotation avec l'argument *master*: il n'y a aucun détail à prendre en considération outre celui de savoir si la boucle annotée mérite de l'être. Un exemple d'une telle situation est donné au Programme 4.3.

```
void init(int N, int A[]){
  int MyRank;
  MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

  if (MyRank == 0) {
    for (int i = 0; i < N; ++i)
      A[i] = i;
  }
}
```

Programme 4.2 – Résultat de l'utilisation du pragma *master*

```
void compute(int N, int A[N]) {
#pragma clang loop distribute(worker)
  for (int i = 0; i < N; ++i) {
    if (i < N / 2)
      A[i] = A[i] * 100;
    else
      A[i] = A[i] + 100;
  }
}
```

Programme 4.3 – Exemple d'utilisation du pragma *worker* avec une boucle simple

La transformation d'une boucle désignée comme *worker* passe par l'ajout de trois éléments. Le premier est la modification des bornes originales de la boucle. Ainsi, compte tenu que la boucle est exécutée par divers processus, il faut faire en sorte que le travail entre ces processus soit réparti de la façon la plus équitable possible. Pour ce faire, les bornes sont modifiées pour tenir compte du nombre de processus lancés par MPI et leur rang. Les informations nécessaires à ce calcul sont obtenues à l'aide d'appels aux fonctions *MPI_Comm_rank* et *MPI_Comm_size*. Elles sont utilisées pour déterminer de nouvelles bornes de façon à réduire le travail des *workers* aux données qu'ils possèdent. Un exemple de bornes modifiées est donné dans le Programme 4.4. Ainsi, on observe que les variables *from* et *to* forment l'intervalle sur lequel un processus donné agit.

Après avoir divisé le calcul du corps de la boucle, il faut s'attarder à distribuer les structures impliquées dans la boucle annotée. Ceci se fait en deux parties. La première partie consiste à distribuer les structures de données vers tous les processus démarrés par MPI. Ceci est réalisé en générant un appel à la fonction MPI Scatter. La seconde partie, inverse à la première, consiste à regrouper les parties des structures de données précédemment distribuées. Ceci passe par la génération d'un appel à la fonction *MPI_Gather*. Un appel à cette fonction fait en sorte que chaque processus envoie ses parties de structures de données à un processus donné. Dans le cas présent, chaque processus envoie sa partie de la structure de données au processus maître pour que ce dernier reconstruise la structure de données mise à jour. Le Programme 4.4 illustre le résultat de la transformation du Programme 4.3 par le processus de parallélisation de *Clang-MPI*.

L'application de l'annotation *pragma clang loop distribute(worker)* à une boucle faisant partie d'un imbrication de boucles nécessite, quant à elle, un peu plus de réflexion. Effectivement, l'annotation affecte non seulement les bornes de la boucle choisie, mais aussi l'ensemble des structures de données accédées à tous les niveaux à l'intérieur de cette boucle. Dans le Programme 4.5, la boucle la plus externe a été choisie comme cible de parallélisation automatique. Le choix de la boucle la plus englobante est le choix prescrit dans la plupart des situations. De fait, un système à mémoire distribuée n'est vraiment bien exploité que lorsque que le temps de calcul est plus important que le temps de communication.

```
void compute(int N, int A[N]) {
  int MyRank;
  int WorldSize;
  MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
  MPI_Comm_size(MPI_COMM_WORLD, &WorldSize);

  int from = MyRank * N / WorldSize;
  int to = (MyRank + 1) * N / WorldSize;

  MPI_Scatter(A, N / WorldSize, MPI_INT, &A[from],
              N / WorldSize, MPI_INT, 0,
              MPI_COMM_WORLD);

  for (int i = from; i < to; ++i) {
    if (i < N / 2)
      A[i] = A[i] * 100;
    else
      A[i] = A[i] + 100;
  }

  MPI_Gather(&A[from], N / WorldSize, MPI_INT, A,
             N / WorldSize, MPI_INT, 0,
             MPI_COMM_WORLD);
}
```

Programme 4.4 – Résultat de la transformation d'une boucle simple à l'aide du pragma *worker*

```
void compute(int N, double A[N][N], double B[N][N],
             double C[N][N]) {
#pragma clang loop distribute(worker)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      C[i][j] = 0;
      for (int k = 0; k < N; ++k) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Programme 4.5 – Exemple d'utilisation du pragma *worker* avec des boucles imbriquées

Le Programme 4.6, issu de la parallélisation du Programme 4.5, permet de pousser plus loin la présentation de plusieurs éléments de la transformation de boucles désignées comme *worker* qui ne furent pas abordés dans le cadre de l'exemple avec une boucle simple. Ainsi, on peut observer que la distribution des données peut se faire à l'aide de plus d'une fonction MPI. En effet, des appels à *MPI_Scatter* ne sont fait que pour distribuer des structures de données accédées à l'aide de la variable d'induction de la boucle annotée. Ce choix repose sur le fait que la partition des données doit suivre la partition du calcul et que ce dernier est divisé sur la base de la variable d'induction de la boucle annotée. Toutes les autres structures de données sont intégralement transmises à chaque processus impliqué dans le calcul. Ce choix permet d'effectuer facilement une multitude de calculs sans devoir confronter le problème très difficile de la partition optimisée des structures de données. Finalement, suivant la même logique que pour la distribution, le regroupement des données est effectué à l'aide d'appels à la fonction *MPI_Gather* pour autant que la structure accédée en écriture fasse appel à la variable d'induction de la boucle annotée.

```
void compute(int N, double A[N][N], double B[N][N],
             double C[N][N]) {
  int MyRank;
  int WorldSize;
  MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
  MPI_Comm_size(MPI_COMM_WORLD, &WorldSize);

  int from = MyRank * N / WorldSize;
  int to = (MyRank + 1) * N / WorldSize;

  MPI_Scatter(A, N * N / WorldSize, MPI_DOUBLE, &A[from],
              N * N / WorldSize, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
  MPI_Bcast(B, N * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  for (int i = from; i < to; ++i) {
    for (int j = 0; j < N; ++j) {
      C[i][j] = 0;
      for (int k = 0; k < N; ++k) {
        C[i][j] += A[i][k] * B[j][k];
      }
    }
  }

  MPI_Gather(&C[from], N * N / WorldSize, MPI_DOUBLE, C,
             N * N / WorldSize, MPI_DOUBLE, 0,
             MPI_COMM_WORLD);
}
```

Programme 4.6 – Résultat de la transformation de boucles imbriquées à l'aide du pragma *worker*

4.1.3 Limitations et solutions potentielles

Bien qu'il ne soit qu'une preuve de concept, le projet, dans son état actuel, comporte des limitations. Cette approche comporte plusieurs limitations. Les plus importantes concernent les faiblesses des annotations, de la distribution des données et du paradigme maître-esclave.

La limitation la plus apparente, tout du moins du point de vue de l'utilisateur, est que les annotations offertes par *Clang-MPI* sont limitées au niveau des boucles. Ceci découle du fait que l'outil est basé sur les annotations de boucles déjà présentes au niveau des extensions des langages C/C++ offertes par le compilateur *Clang*. En effet, *Clang* permet déjà de spécifier quelques annotations au niveau des boucles pour guider le compilateur dans les transformations potentielles qu'il peut leur appliquer tel que le déroulement de boucle ou la vectorisation. Une solution potentielle pour éliminer cette limitation serait de laisser tomber cette base pour plutôt tenter de produire des extensions de langages telles que celles offertes par l'outil OpenMP. En effet, OpenMP permet de définir des portées arbitraires contenant une série d'instructions à exécuter en parallèle suivant l'annotation de cette portée. Les implications au niveau du travail à faire pour générer du code LLVM IR serait par contre très nombreuses et importantes. Ainsi, il faudrait carrément introduire des énoncés et de nouvelles expressions dans la grammaire du langage et par la suite implanter beaucoup de nouvelles fonctionnalités à l'analyseur syntaxique de *Clang* pour prendre en charge les nouveaux noeuds introduits dans de potentiels arbres syntaxiques abstraits générés à partir de programmes C/C++.

Un autre élément limitant de beaucoup les possibilités de l'outil développé est le fait qu'il n'effectue aucune analyse de dépendance de données entre les divers accès mémoire faits dans le cadre d'une boucle ou d'un imbrication de boucles. Ce faisant, il force l'utilisateur à être lui-même responsable de déduire si la boucle est une bonne candidate pour la parallélisation. Cette charge de travail donnée au programmeur rend le processus de parallélisation automatique bien moins transparent qu'il ne devrait l'être. L'idée serait donc de procéder à une vérification des dépendances dans une boucle pour potentiellement informer l'utilisateur que la boucle à paralléliser ne se prête pas à cette action en précisant si possible quelle dépendance affecte ainsi le processus de parallélisation automatique. Qui plus est, mettre en branle une analyse de

dépendances mémoire pourrait aussi permettre d'agir dans plus de cas en identifiant les cas où, bien qu'il y ait présence de dépendances, des modifications préliminaires, telles que celles proposées dans [All02], permettraient tout de même de procéder à la parallélisation automatique de la boucle désignée.

La distribution des données est aussi un facteur limitant dans *Clang-MPI*. Ainsi, la distribution présentement implémentée est très rudimentaire et fait donc en sorte que plus de communication que nécessaire est générée. Tel que mentionné plus tôt, le fait de passer trop de temps à communiquer des données dans un système distribué réduit l'intérêt d'utiliser une telle plateforme pour accomplir la tâche donnée. Une façon d'améliorer *Clang-MPI* de ce point de vue passerait par l'analyse des dépendances de données entre les instructions d'une boucle. Ce faisant, on pourrait déterminer des ensembles de données nécessitant d'être communiqués entre les processus pour assurer le bon déroulement du calcul. Grosso modo, il faudrait ajouter à *Clang-MPI* les notions de *live-in sets* et de *live-out sets* telles que décrites dans les travaux d'Uday Bondhugula [Bon11, Red14].

La dernière limitation est le fait que *Clang-MPI* ne génère que des programmes obéissant au paradigme maître-esclave. Bien entendu, cette limitation est liée au choix de MPI comme technologie de calcul parallèle distribué. Le problème de se limiter à ce paradigme est qu'il empêche la parallélisation de programmes orientés tâches plutôt qu'orientés données. Même si ceci n'est pas tant un grave problème dans le cadre du calcul scientifique qui, la plupart du temps, cadre bien dans ce paradigme, devient un inconvénient lorsqu'on sort de ce créneau. Une solution potentielle consisterait à changer la technologie aidant à la distribution de données choisie. Un remplacement potentiel pourrait être GASNet [Bon02], une bibliothèque de communication déjà utilisée dans divers projets qui tente d'implémenter le paradigme PGAS. Le plus important est que le choix retenu permet la mise en place de programmes du type *Multiple Programs, Multiple Data* et non seulement des programmes du type *Simgle Program, Multiple Data*.

4.2 Transformation de la représentation intermédiaire

Cette section présente la partie la plus significative du projet. Elle contient une explication de la méthode de parallélisation automatique que nous avons implémentée pour systèmes à mémoire distribuée à travers un outil. Elle jette les bases de ce que doit être la représentation intermédiaire de programmes transformés par cet outil. Les transformations sont détaillées tant pour le maître que pour l'esclave (parfois appelé travailleur).

4.2.1 Préparation

Avant de procéder à la transformation des boucles annotées et à l'ajout de fonctions MPI et d'appels à ces fonctions, il faut préparer le code LLVM IR. Ceci passe par l'appel de deux passes offertes par LLVM: *mem2reg* et *simplifycfg*.

La passe de transformation appelée *mem2reg* a pour effet de promouvoir les références mémoires à des références de registres. En d'autres termes, cette passe transforme la représentation intermédiaire de façon à minimiser les instructions de type *store* et de type *load* en les remplaçant, dans la mesure du possible, par de nouvelles variables dans la représentation intermédiaire. Un bon modèle mental du travail effectué est de considérer que la représentation intermédiaire se retrouve maintenant sur une machine possédant une infinité de registres. De plus, et ce qui est plus intéressant pour les transformations que *Clang-MPI* doit effectuer, cette transformation a pour conséquence de faire apparaître des noeuds phi (voir Chapitre 2) dans la représentation intermédiaire au niveau des boucles. Ces noeuds sont importants étant donné que les informations qu'ils contiennent sont utilisées dans la transformation d'une boucle annotée *worker*.

Avant de détailler le travail effectué par la passe appelée *simplifycfg*, il convient de fournir une explication plus en profondeur que celle donnée au Chapitre 1 de ce qu'est un CFG. Ainsi, comme il a été précédemment mentionné, un CFG est une représentation graphique d'un programme équivalente bien souvent à la représentation intermédiaire utilisée par un compilateur donné. Comme tout graphe, le CFG est

composé de deux éléments: des sommets et des arcs. Les sommets sont appelés blocs de base. On définit un bloc de base comme un segment de code, bien souvent écrit dans un langage intermédiaire, dont les instructions qui le composent sont nécessairement toutes exécutées si l'une d'entre elles est exécutée. De plus, ces instructions sont nécessairement exécutées séquentiellement, c'est-è-dire selon l'ordre dans lequel elles apparaissent dans le bloc. Ces segments de code se terminent toujours par une instruction de branchement. Les arcs représentent l'ordre d'exécution des divers blocs de base composant le CFG. Ils permettent de représenter les structures de contrôle présentes dans le programme original.

La passe *simplifycfg* sera appliquée à la représentation intermédiaire du programme après la passe *mem2reg*. Cette passe est, en réalité, un amalgame de passes toutes appliquées dans le but de simplifier le CFG du programme compilé, et ce, fonction par fonction. Cette passe élimine le code considéré comme mort et fusionne des blocs de base suite à cette élimination. Plus explicitement, cette passe effectue les tâches suivantes: elle supprime les blocs de base sans prédécesseurs et elle fusionne un bloc de base et son prédécesseur si ce dernier est son unique prédécesseur et qu'il n'a qu'un successeur. Cette transformation est intéressante, car elle produit une représentation intermédiaire avec laquelle il est plus facile de d'effectuer des opérations et dans laquelle il est plus facile de trouver des erreurs liées à la génération de code parallèle.

En plus des deux passes précédemment décrites, la préparation de la représentation intermédiaire inclut également l'ajout de nouveaux éléments dans le module contenant la ou les boucles traitées. Ainsi, il faut envisager l'ajout de déclarations aux fonctions MPI_Init, *MPI_Finalize*, *MPI_Scatter*, *MPI_Bcast* et *MPI_Gather* qui ont toutes les chances d'être utilisées dans le programme résultant du processus de parallélisation automatique. De plus, certaines structures de données MPI, notamment les communicateurs et les types de données MPI, sont également déclarées dans le module. De surcroît, la fonction principale du module doit comporter un appel à la fonction *MPI_Init* et un appel à la fonction *MPI_Finalize*, respectivement au début et à la fin de ses instructions.

Une fois que cette préparation a été appliquée à la représentation intermédiaire extraite du programme à paralléliser, le traitement des pragmas présents dans le

```
void func(float A[32][1000]) {
  for (int i = 0; i < 32; ++i) {
    for (int j = 0; j < 1000; ++j) {
      printf("%f", A[i][j]);
    }
    printf("\n");
  }
}
```

Programme 4.7 – Fonction imprimant une matrice sur la sortie standard

programme peuvent être traiter.

4.2.2 Parallélisation du maître

Au niveau de la transformation de boucles annotées par *pragma clang loop distribute(master)*, le but recherché est d'isoler la boucle dans une conditionnelle de façon à ce qu'elle ne soit exécuter que par un processus donné, soit le processus 0. Pour donner un exemple de cette technique, la boucle illustrée par le CFG présenté à la figure 4.1 est transformée selon l'algorithme de parallélisation du maître. Ce CFG a été obtenu en convertissant d'abord le Programme 4.7 en LLVM IR, puis en appliquant les passes *mem2reg* et *simplifycfg*.

Pour effectuer cette transformation, on doit d'abord identifier le bloc de base servant de pré-entête pour la boucle à paralléliser. En d'autres termes, on veut identifier le bloc de base se situant juste avant le bloc de base où un choix doit être fait pour décider si le programme doit entrer dans une boucle. Dans le cas traité, il s'agit du bloc ayant le nom *%0*. Par la suite, on introduit dans ce bloc une variable pour contenir le rang du processus dans lequel le code s'exécute. Par après, un appel à la fonction *MPI_Comm_rank* est fait pour donner une valeur à la variable précédemment créée. Après cela, une comparaison avec 0 est générée pour déterminer si la boucle doit s'exécuter. Pour utiliser le résultat de la comparaison, le branchement inconditionnel du bloc de pré-entête devient un branchement conditionnel sur le résultat de la comparaison. Le bloc de pré-entête est alors suivi autant du premier bloc de la boucle que du bloc de sortie. Le code de la boucle est donc de cette manière isolé des processus

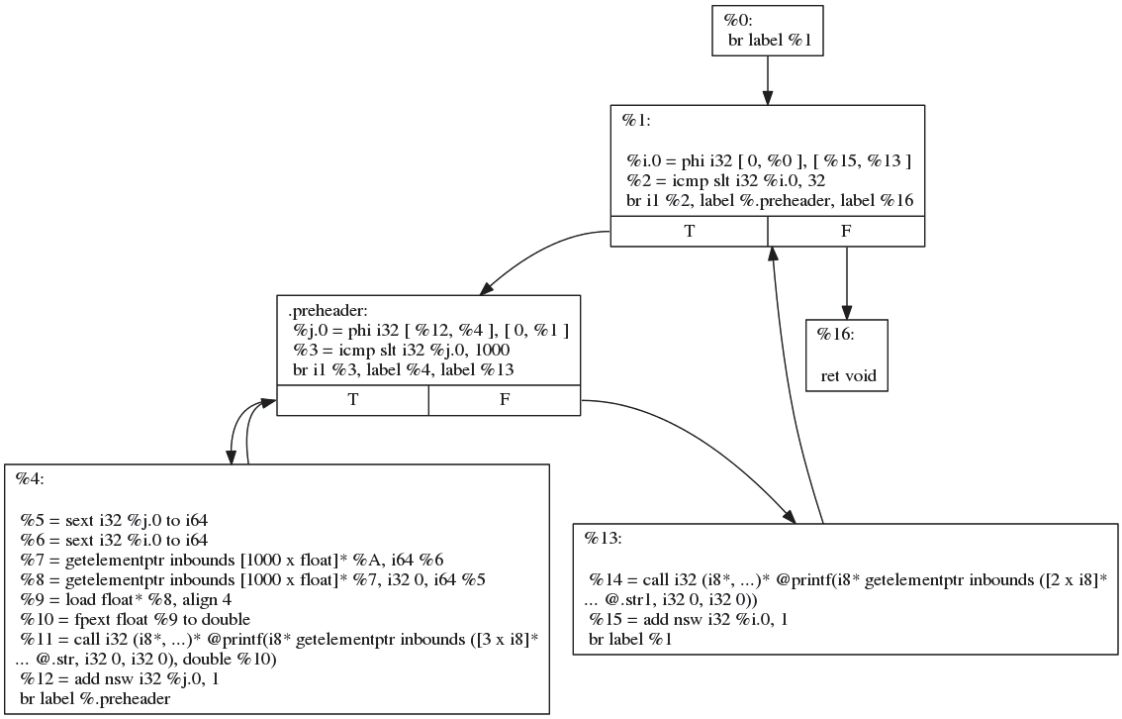

Figure 4.1 – Représentation intermédiaire d'un programme affichant une matrice sur la sortie standard

autres que le processus maître. Le résultat de cette transformation peut être observé à la Figure 4.2.

4.2.3 Parallélisation des travailleurs

Pour ce qui est des boucles annotées par *pragma loop distribute(worker)* le but recherché est de les structurer de manière à ce que les itérations soient réparties de façon équitable entre les processus impliqués dans l'exécution du programme. Le Programme 4.5 illustre une boucle qui est en mesure de se faire paralléliser par *Clang-MPI*.

Pour atteindre le but recherché, la première étape consiste à créer des variables auxiliaires qui contiennent le rang du processus courant ainsi que le nombre total de processus s'exécutant. Ces variables apparaîssent dans le CFG de la fonction contenant la boucle au niveau du bloc de pré-entête. Une fois ces variables créées, elles se

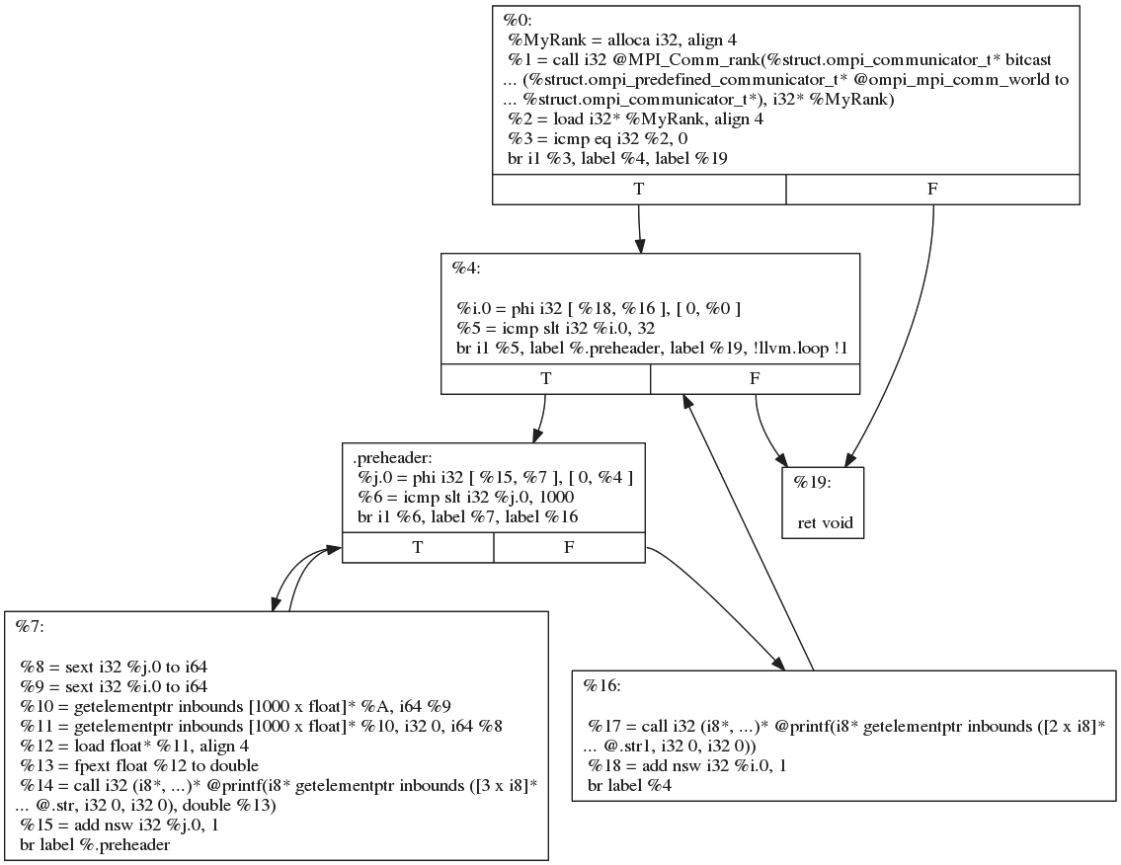

Figure 4.2 – Représentation intermédiaire d'un programme MPI où le processus maître est responsable d'afficher une matrice sur la sortie standard

voient affecter une valeur au travers des appels aux fonctions *MPI_Comm_rank* et *MPI_Comm_size* qui sont générés par la suite.

À partir des valeurs de ces variables, les nouvelles bornes de la boucle en cours de traitement sont créées. En effet, pour s'assurer du partage le plus équitable du traitement à effectuer, chaque processus exécute les itérations de la boucle allant de $MonRang * NbProcessus/N$ à $(MonRang + 1) * NbProcessus/N$ où N est la borne supérieure originale de la boucle. Cette modification des bornes de la boucle se fait évidemment au niveau du bloc d'entête de la boucle.

Après avoir réparti le traitement, la prochaine étape consiste à répartir les données impliquées dans ce traitement. Pour ce faire, on génère des appels aux fonctions

MPI_Scatter, *MPI_Bcast* et *MPI_Gather*.

La fonction *MPI_Scatter* envoie des parties de données d'une certaine taille à tous les processus s'exécutant. Elle est donc une candidate de choix pour répartir les données accédées en lecture à l'aide de la variable d'induction de la boucle traitée. En effet, chaque processus n'a maintenant qu'une partie du traitement à effectuer qui lui est dictée par le nouvel intervalle d'itérations qu'il se voit affecté. Par conséquent, les données qu'un processus accède vont aussi se situer sur le nouvel intervalle de la variable d'induction de la boucle traitée. Pour générer l'appel à cette fonction, il faut préparer deux éléments. Le premier élément est une variable qui contient le nombre d'éléments de la structure de données à transmettre sur le réseau. Ceci est fait en accédant à la taille de la structure de données à partager que l'on divise par le nombre de processus dans le système. Le second élément est l'adresse du point d'accès de la structure de données. Ceci est fait en générant des instructions d'accès mémoire qui respecte le type de la structure de données accédée, c'est-à-dire un pointeur ou un tableau statique.

Une fois cela fait, un appel à *MPI_Scatter* est généré et on lui donne un nom indiquant à quelle structure de données il se rattache. En effet, l'expérience a démontré que LLVM génère plusieurs instructions pour un même accès mémoire qui amènent *Clang-MPI* à croire qu'une structure donnée devrait être distribuée au moyen de *MPI_Scatter*. Si aucune mesure n'est prise, *Clang-MPI* génère plusieurs appels à *MPI_Scatter*, un pour chaque instruction dont son analyse conclut au besoin de distribuer une structure de données. Par conséquent, pour éviter les redondances, on s'assure donc, à l'aide d'instructions d'appel de fonction nommées, de ne pas envoyer la même structure de données sur le réseau à de multiples reprises. Ainsi, il n'y a qu'un appel à la fonction *MPI_Scatter* pour une structure de données avec la variable *ScatterX*. Toute tentative de générer une seconde instruction qui crée une telle variable est interceptée et bloquée par *Clang-MPI*.

Pour ce qui est de la fonction *MPI_Bcast*, elle a pour effet d'envoyer une copie de la structure de données impliquée à tous les processus présents dans le système. Dans le contexte présent, elle est utilisée pour envoyer des copies des structures de données dont l'accès, strictement en lecture, est fait par des variables d'induction appartenant à des boucles internes de la boucle traitée. La création d'un appel à cette fonction est

relativement aisée, le seul élément pour lequel de nouvelles instructions doivent être générées est la taille de la structure de données à copier. Une fois que cette information est affectée à une variable, l'appel à la fonction *MPI_Bcast* est fait en utilisant les informations présentes dans le contexte du module courant. Aussi, tout comme les appels à *MPI_Scatter*, les appels à *MPI_Bcast* sont concrètement introduits dans le programme au niveau du bloc de pré-entête de la boucle concernée et sont aussi vérifiés pour qu'un seul appel soit fait par les structures de données concernées.

Finalement, au niveau de la fonction *MPI_Gather*, un appel unique à celle-ci est généré pour chaque structure de données accédée en écriture au moyen de la variable d'induction de la boucle traitée. Compte tenu que cette fonction est essentiellement l'inverse de la fonction *MPI_Scatter*, la préparation de ces arguments est la même que pour ceux de *MPI_Scatter*. La seule différence à ce niveau se retrouve au niveau de l'ordre des paramètres. L'autre différence est qu'un appel à *MPI_Gather* est placé dans le bloc de sortie d'une boucle et non dans sa pré-entête.

La Figure 4.4 montre le résultat final du processus de parallélisation appliqué au programme de la Figure 4.3.

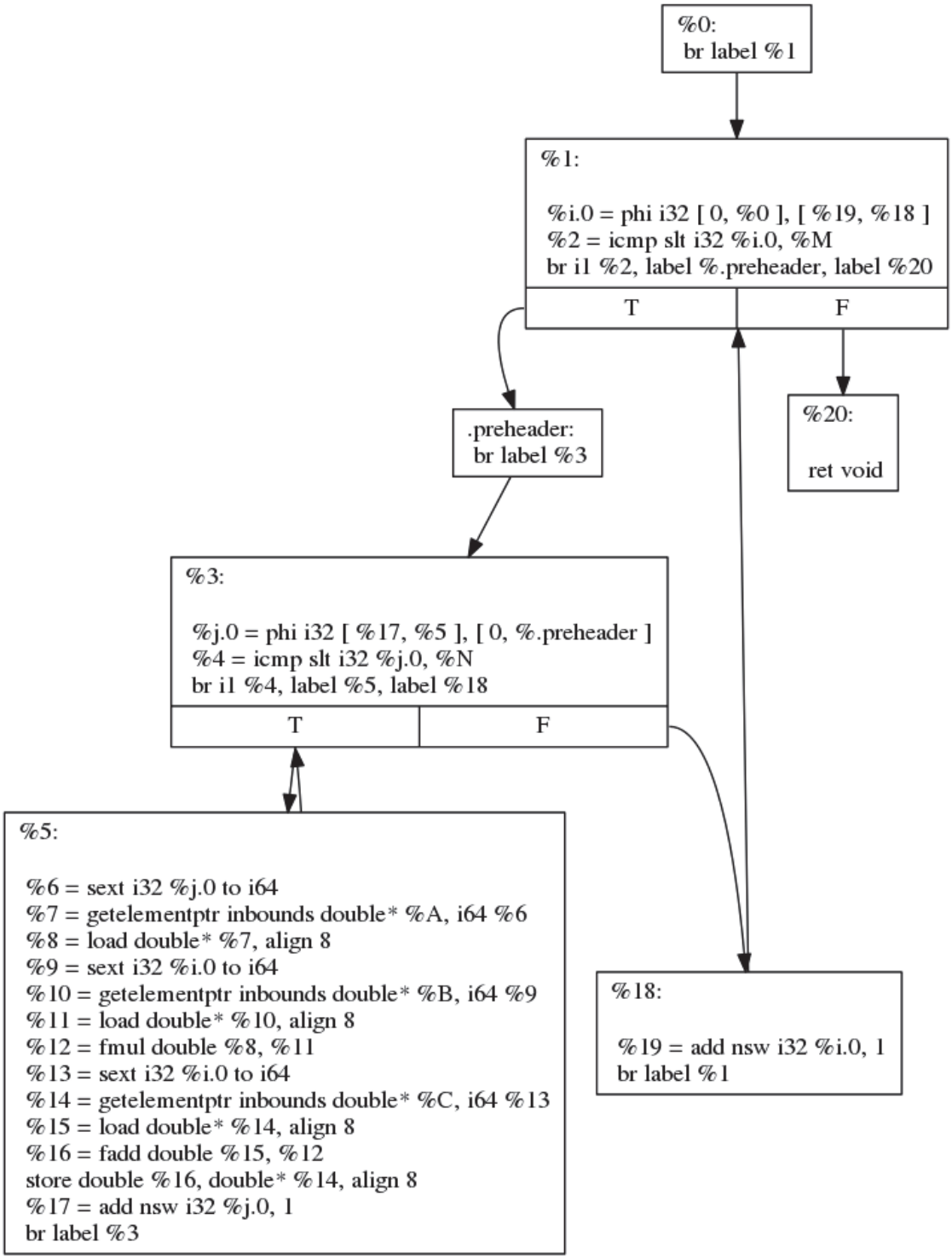

Figure 4.3 – Représentation intermédiaire d'un programme effectuant une multiplication de matrices

```
%0:
 %MyRank = alloca i32, align 4
 %1 = call i32 @MPI_Comm_rank(%struct.ompi_communicator_t* bitcast
... (%struct.ompi_predefined_communicator_t* @ompi_mpi_comm_world to
... %struct.ompi_communicator_t*), i32* %MyRank)
 %WorldSize = alloca i32, align 4
 %2 = call i32 @MPI_Comm_size(%struct.ompi_communicator_t* bitcast
... (%struct.ompi_predefined_communicator_t* @ompi_mpi_comm_world to
... %struct.ompi_communicator_t*), i32* %WorldSize)
 %3 = load i32* %MyRank, align 4
 %4 = mul nsw i32 %3, %M
 %5 = load i32* %WorldSize, align 4
 %6 = sdiv i32 %4, %5
 %7 = load i32* %MyRank, align 4
 %8 = add nsw i32 %7, 1
 %9 = mul nsw i32 %8, %M
 %10 = load i32* %WorldSize, align 4
 %11 = sdiv i32 %9, %10
 %12 = bitcast double* %B to i8*
 %13 = load i32* %WorldSize, align 4
 %14 = sdiv i32 %M, %13
 %15 = sext i32 %6 to i64
 %16 = getelementptr inbounds double* %B, i64 %15
 %17 = bitcast double* %16 to i8*
 %18 = load i32* %WorldSize, align 4
 %19 = sdiv i32 %M, %18
 %ScatterB = call i32 @MPI_Scatter(i8* %12, i32 %14, %struct.ompi_datatype_t*
... bitcast (%struct.ompi_predefined_datatype_t* @ompi_mpi_double to
... %struct.ompi_datatype_t*), i8* %17, i32 %19, %struct.ompi_datatype_t* bitcast
... (%struct.ompi_predefined_datatype_t* @ompi_mpi_double to
... %struct.ompi_datatype_t*), i32 0, %struct.ompi_communicator_t* bitcast
... (%struct.ompi_predefined_communicator_t* @ompi_mpi_comm_world to
... %struct.ompi_communicator_t*))
 %20 = bitcast double* %C to i8*
 %21 = load i32* %WorldSize, align 4
 %22 = sdiv i32 %M, %21
 %23 = sext i32 %6 to i64
 %24 = getelementptr inbounds double* %C, i64 %23
 %25 = bitcast double* %24 to i8*
 %26 = load i32* %WorldSize, align 4
 %27 = sdiv i32 %M, %26
 %ScatterC = call i32 @MPI_Scatter(i8* %20, i32 %22, %struct.ompi_datatype_t*
... bitcast (%struct.ompi_predefined_datatype_t* @ompi_mpi_double to
... %struct.ompi_datatype_t*), i8* %25, i32 %27, %struct.ompi_datatype_t* bitcast
... (%struct.ompi_predefined_datatype_t* @ompi_mpi_double to
... %struct.ompi_datatype_t*), i32 0, %struct.ompi_communicator_t* bitcast
... (%struct.ompi_predefined_communicator_t* @ompi_mpi_comm_world to
... %struct.ompi_communicator_t*))
 %28 = bitcast double* %A to i8*
 %BcastA = call i32 @MPI_Bcast(i8* %28, i32 %M, %struct.ompi_datatype_t*
... bitcast (%struct.ompi_predefined_datatype_t* @ompi_mpi_double to
... %struct.ompi_datatype_t*), i32 0, %struct.ompi_communicator_t* bitcast
... (%struct.ompi_predefined_communicator_t* @ompi_mpi_comm_world to
... %struct.ompi_communicator_t*))
 br label %29
```

Figure 4.4 – Représentation intermédiaire d'un programme MPI où le processus maître effecte une multiplication de matrices

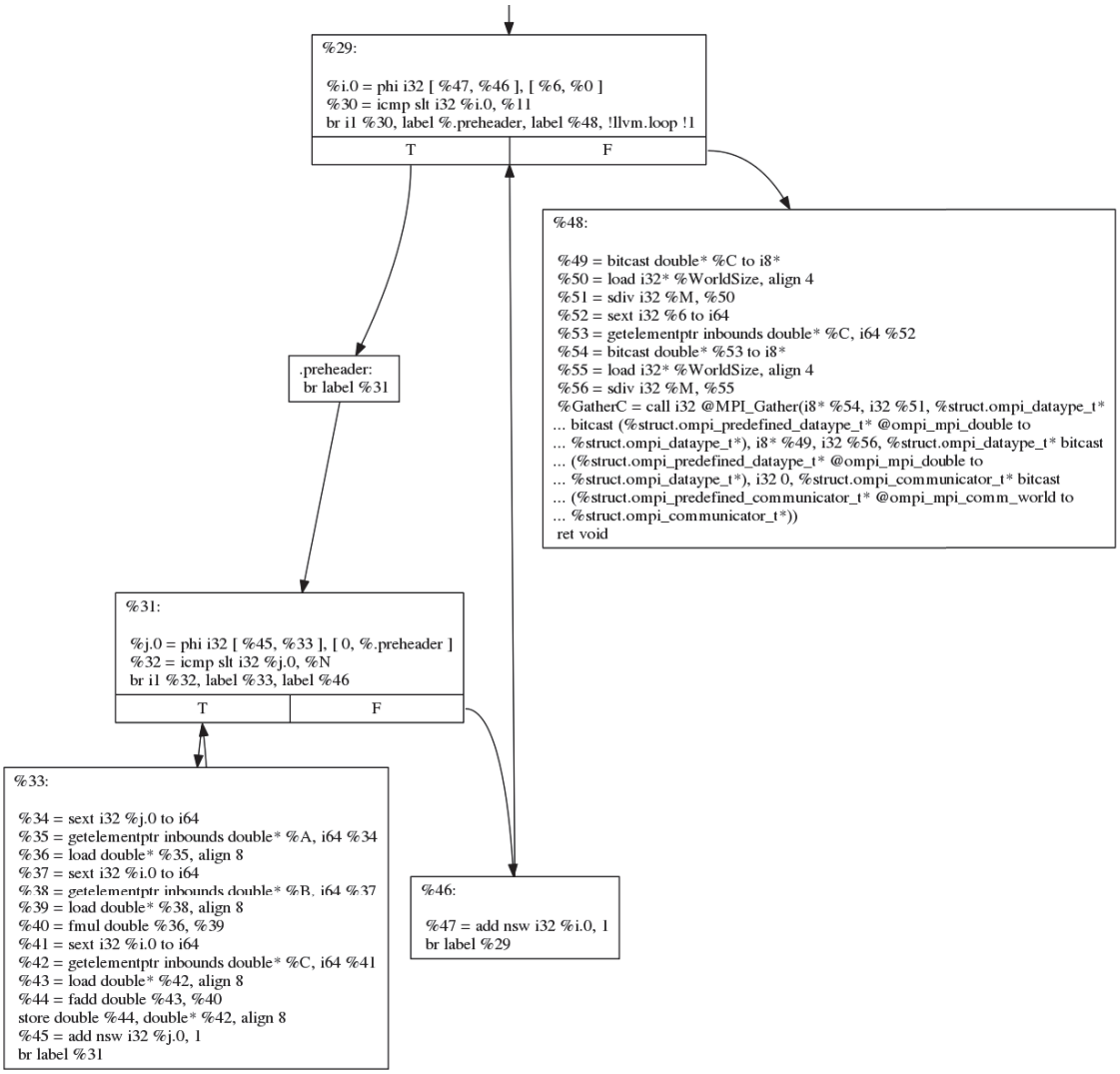

Figure 4.4 – Représentation intermédiaire d'un programme MPI où le processus maître effecte une multiplication de matrices (suite)

Chapitre 5

Expériences

Ce chapitre a pour but d'évaluer l'outil *Clang-MPI* à l'aide de tests afin de dégager des conclusions qui s'imposent, en particulier sur son efficacité et sa pertinence. Cette évaluation est faite à partir, d'une part, de bancs d'essai issus de la suite *PolyBench* et, d'autre part, d'une étude de cas qui consiste à paralléliser automatiquement un réseau de neurones destiné à être exécuté sur un système à mémoire distribuée. Ces test sont décrits respectivement dans les sections 5.1 et 5.2.

5.1 PolyBench

Cette section présente l'évaluation de *Clang-MPI* à travers le banc d'essai PolyBench. Tout d'abord, elle décrit brièvement *PolyBench* et les programmes qu'il contient. Ensuite, elle présente les résultats du banc d'essai tout en soulignant le contexte dans lequel ils ont été obtenus. Enfin, elle conclut sur quelques limitations tirées de ces expériences.

5.1.1 Description

PolyBench [Pou12] est un ensemble de bancs d'essai créé dans le but d'offrir une base standard pour évaluer les travaux de recherche effectués sur le modèle polyédral de compilation. Dans cette optique, il offre de nombreux programmes dont le coeur

de leur exécution est dominé par une ou plusieurs boucles. Ces programmes sont principalement issus des domaines des mathématiques, de la physique et de la chimie. Il s'agit donc de programmes riches en opportunités de parallélisation automatique pour un compilateur utilisant le modèle polyédral.

5.1.2 Résultats

Pour évaluer *Clang-MPI*, une sélection de bancs d'essai faisant partie de *PolyBench* a été faite. Pour chacun de ces bancs d'essai retenus, trois exécutables ont été produits. Le premier est le programme original compilé sans ajout de *pragmas*. Il est utilisé comme référence pour déterminer s'il est approprié de le paralléliser. Le second exécutable provient de la compilation avec *Clang-MPI*, mais en insérant des *pragmas* dans le programme original. Le dernier exécutable correspond à une variante du programme original, mais parallélisé manuellement en utilisant la bibliothèque MPI. Toutes ces versions ont été compilées avec l'argument *-O2* comme seul argument d'optimisation et l'argument *-DEXTRALARGE_DATASET* pour utiliser à chaque fois les jeux de données les plus grands offerts par *PolyBench*. Tous les exécutables ont été placés sur un noeud de calcul du super-ordinateur *Mammouth*. Plus précisément, le banc d'essai *PolyBench* tire avantage du serveur de calcul *Mammouth-parallèle* 2 de façon à ce que les exécutables exploitent simultanément un maximum de 24 coeurs.

Les Figures 5.1, 5.2, 5.3 et 5.4 présentent les résultats obtenus lors de ces expériences. On constate pour les bancs d'essai *2mm* et *3mm* que *Clang-MPI* est capable de produire des programmes ayant des temps d'exécution proches de ceux obtenus avec des programmes parallélisés manuellement. Par contre, les résultats avec les bancs d'essai *gemver* et *gesummv* montrent qu'il y a un certain écart entre ce qui est attendu de *Clang-MPI* au niveau des temps d'exécution des programmes transformés et ce que l'on est capable d'obtenir manuellement.

Aussi, le Tableau 5.1 donne un comparatif entre le temps d'exécution de la version séquentielle d'un programme avec les versions parallèles, créées à l'aide de MPI et de *Clang-MPI*, les plus lentes. Précisons que ces résultats ont tous été obtenus en utilisant un seul noeud de la grappe de calcul *Mammouth* composé de deux processeurs AMD

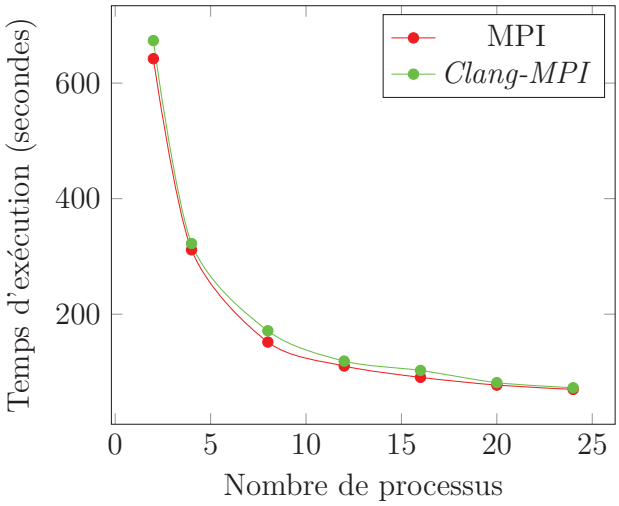

Figure 5.1 – Comparatif du temps d'exécution du banc d'essai *2mm*

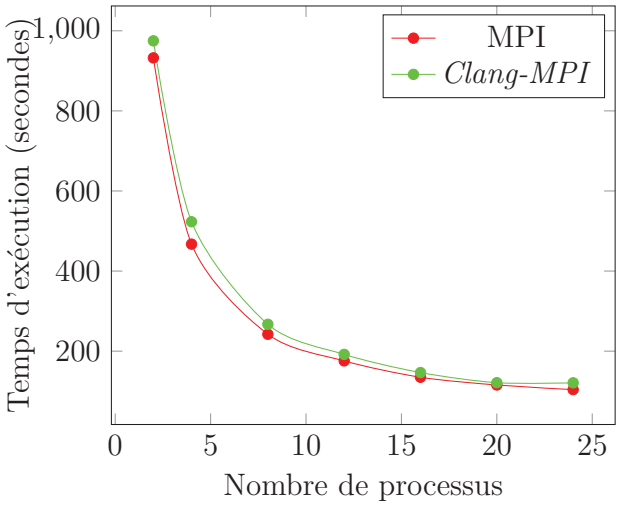

Figure 5.2 – Comparatif du temps d'exécution du banc d'essai *3mm*

douze coeurs. On peut constater qu'au niveau des bancs d'essai *gemver* et *gesummv*, le temps pris pour exécuter un programme issu de *Clang-MPI* est plus grand que celui pris par la version séquentielle du même programme.

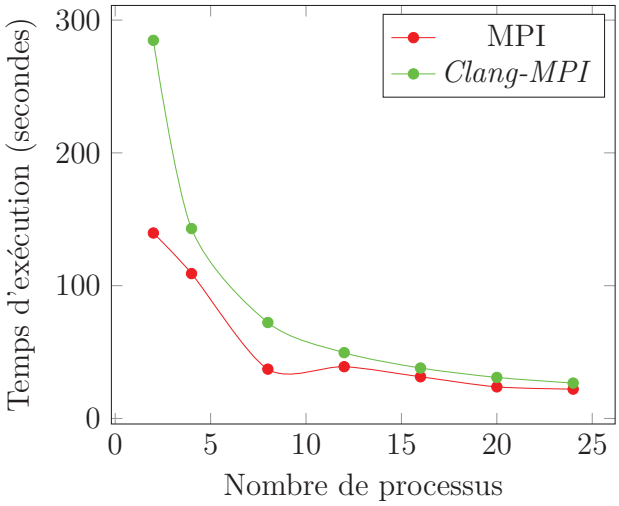

Figure 5.3 – Comparatif du temps d'exécution du banc d'essai *gemver*

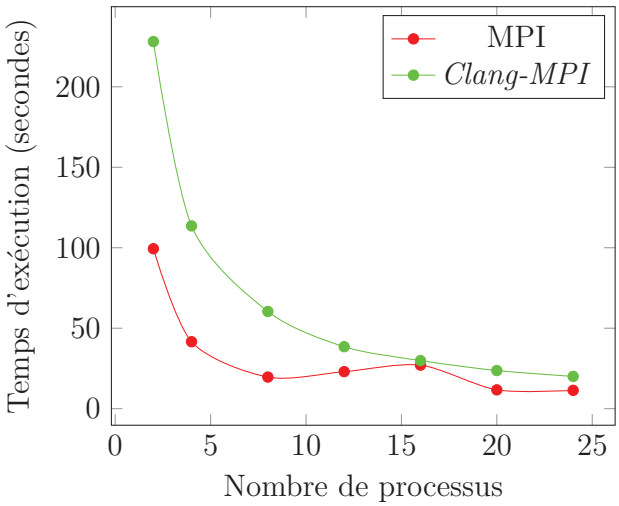

Figure 5.4 – Comparatif du temps d'exécution du banc d'essai *gesummv*

5.1.3 Discussion

Les résultats obtenus grâce à *PolyBench* peuvent être expliqués par un choix d'architecture fait en début de projet. Ainsi, chaque boucle annotée avec un *pragma* de *Clang-MPI* est considérée indépendamment des autres. Ceci fait en sorte que le nombre d'appels à des fonctions de MPI dans un programme transformé par *Clang-*

Banc d'essai	Séquentielle	MPI	*Clang-MPI*
2mm	1159.486	642.230	673.614
3mm	1876.381	932.425	974.930
gemver	276.010	139.658	284.740
gesummv	140.742	99.385	228.111

Tableau 5.1 – Comparatif des temps d'exécution en secondes de la version séquentielle d'un banc d'essai avec les version parallèles

MPI est beaucoup plus grand qu'il ne peut l'être dans un programme écrit en se utilisant directement MPI.

Au niveau des bancs d'essai *2mm* et *3mm*, comme on peut le voir dans les figures 5.1 et 5.2, ce choix n'affecte pas beaucoup le temps d'exécution de ces programmes. Ceci s'explique par le fait que ces bancs d'essai comportent soit un petit nombre de boucles soit des boucles accédant à un petit nombre de structures de données. Par conséquent, la partie du programme qui dédiée à la communication n'est pas significative et n'affecte pas les gains obtenus en exécutant le programme de façon parallèle.

Évidemment, les bancs d'essai *gemver* et *gesummv*, ne remplissent pas ces conditions comme peuvent en faire foi les figures 5.3 et 5.4. Par conséquent, *Clang-MPI* produit dans des cas comme ceux-là, des programmes qui regroupent tout calcul intermédiaire sur le noeud maître, à l'aide d'un appel à *MPI_Scatter*, et les redistribuent ensuite sur tous les processus impliqués dans le calcul à l'aide d'un appel à *MPI_Bcast*. On convient que cette façon de faire n'est pas optimale. Il vaut mieux que la distribution et le regroupement des données soient fait le moins souvent possible.

Une amélioration possible est de ne pas appliquer les transformations de *Clang-MPI* boucle par boucle. Par exemple, on peut appliquer les transformations sur la base de fonctions. Ainsi, on met en commun tous les accès mémoire effectués dans toutes les boucles d'une fonction donnée pour ensuite tenter de générer le nombre minimal d'appels de fonctions MPI. Ultimement, on peut également d'appliquer les transformations au niveau des modules. De cette façon, on peut garantir que le nombre d'appels à des fonctions MPI est réellement minimal. Le travail pour mettre en place ces améliorations requiert par contre des efforts qui dépassent le cadre de ce projet

de maîtrise.

Un autre point de discussion important à aborder est le choix des bancs d'essai. Ainsi, tous ceux qui ont été utilisés pour obtenir les résultats précédemment présentés ne comportaient aucune dépendance mémoire portée par la boucle dans leur noyau d'exécution. Pour réitérer ce qui avait été souligné au précédent chapitre, *Clang-MPI* n'effectue aucune analyse de dépendances mémoire et donc, il n'est pas en mesure de traiter de tel cas. Il ne possède pas les outils nécessaires pour faire des réarrangements, qui peuvent parfois être très complexes, au niveau des accès mémoire ou de l'ordonnancement des itérations d'une boucle donnée. Par conséquent, tout essai visant à paralléliser une boucle comportant des dépendances portées par la boucle serait évidemment voué à un échec. C'est donc pour cette raison que le nombre de bancs d'essai utilisés pour évaluer *Clang-MPI* est relativement petit.

5.2 Parallélisation automatique d'un réseau de neurones

Dans cette section, on discute de la parallélisation de l'algorithme de rétropropagation dans un réseau de neurones et de l'automatisation de ce processus. Tout d'abord, la section 5.2.1 offre une brève présentation de ce qu'est un réseau de neurones. La section suivante, la section 5.2.2, discute de réseaux de neurones parallèles et donne diverses raisons pour expliquer la difficulté de les implémenter sur systèmes à mémoire distribuée. Ensuite, à la section 5.2.3, les résultats des expériences de parallélisation automatique d'un réseau de neurones sont exposés. Finalement, à la section 5.2.4, on fait une critique de l'approche prise pour paralléliser un réseau de neurones sur une grappe en soulignant ses avantages et ses inconvénients.

5.2.1 Réseau de neurones artificiels

Un réseau de neurones artificiels est une structure de données inspirée des réseaux de neurones biologiques qui peut apprendre, à l'aide d'un algorithme d'apprentissage, à effectuer diverses tâches telles que traduire des textes ou reconnaître des formes dans des images. La forme de base d'un réseau de neurones est constituée de plu-

sieurs couches de neurones artificiels qui sont fortement interconnectés entre elles. Ces couches peuvent être divisées en trois catégories. La première est la couche d'entrée qui sert à acquérir les données à traiter. La seconde catégorie est celles de couches cachées, les couches n'ayant pas d'interactions directes avec le monde extérieur. La dernière catégorie est celle de la couche de sortie qui sert à effectuer des prédictions autant dans un contexte d'apprentissage que d'utilisation après apprentissage. La figure 5.5 donne un exemple d'un réseau de neurones comprenant ces trois catégories de couches.

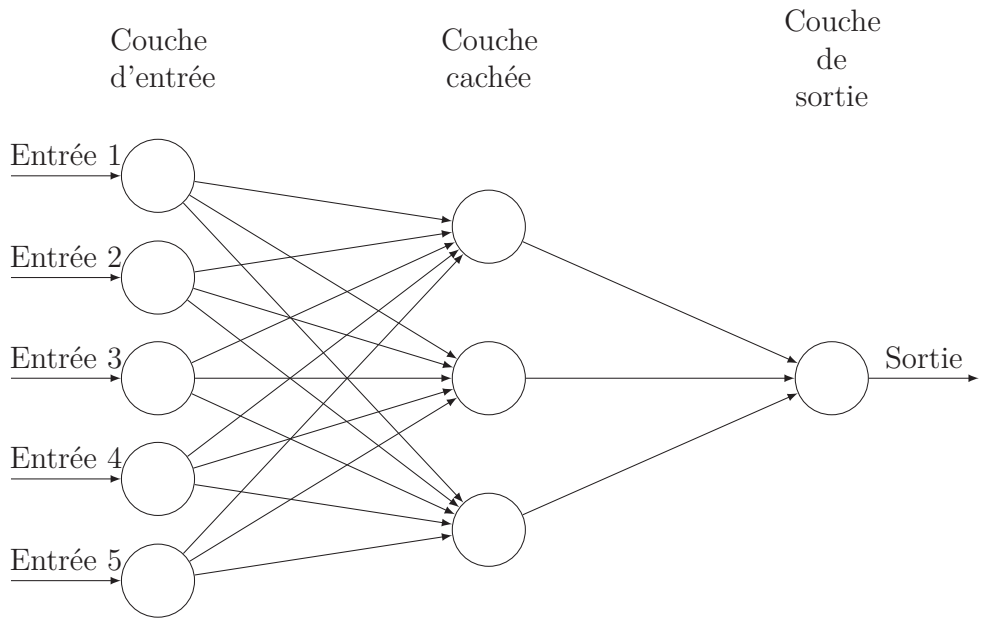

Figure 5.5 – Réseau de neurones

Le processus par lequel un réseau de neurones peut apprendre est composé de deux étapes. La première de ces étapes consiste à faire une propagation avant de l'information à traiter, acquise par la couche d'entrée, jusqu'à la couche de sortie en passant par toutes les couches cachées que possèdent le réseau. Pour effectuer cette propagation avant, on commence par pré-activer une couche en appliquant la formule suivante:

$$\mathbf{a}^{(k+1)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k)} \tag{5.1}$$

Cette formule indique qu'on effectue une multiplication matrice-vecteur avec la couche précédente $\mathbf{h}^{(k)}$ et la matrice de poids $W^{(k)}$ reliant les noeuds de cette dite couche avec la couche que l'on veut activer $\mathbf{a}^{(k+1)}$. À ce résultat, on additionne un biais $\mathbf{b}^{(k)}$. Par la suite, on active la couche en y appliquant une non linéarité $\mathbf{g}$ tel que la fonction sigmoïde ou la fonction *tanh*.

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \tag{5.2}$$

On répète ce processus jusqu'à ce qu'on arrive au niveau de la couche de sortie. À ce niveau, on effectue une prédiction sur ce que les données représente (par exemple, en reconnaissance de caractères, est-ce que cette image représente le chiffre 1). Pour ce faire, on active la couche de sortie avec une fonction appropriée au contexte de l'expérience.

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) \tag{5.3}$$

Un exemple de fonction d'activation, toujours dans le contexte de reconnaissance de caractères, est la fonction softmax, représentée par l'équation 5.4, qui permet de faire une prédiction dans un contexte de classification multi-classes.

$$\mathbf{o}(\mathbf{a}) = \left[\frac{exp(a_1)}{\Sigma_c exp(a_c)} \quad \ldots \quad \frac{exp(a_C)}{\Sigma_c exp(a_c)}\right]^T \tag{5.4}$$

Par la suite, la seconde partie du processus d'apprentissage, la rétropropagation, débute. Le but ici est de propager les gradients des erreurs que le réseau aura pu commettre en effectuant des prédictions. Pour modéliser ces erreurs, on utilise une fonction de coût comme, par exemple, la log-vraisemblance négative.

$$-\log f(\mathbf{x})_y \tag{5.5}$$

Le but recherché est de minimiser cette fonction de coût. Pour ce faire, on calcule d'abord le gradient de la couche de sortie.

$$\nabla \mathbf{a}^{(K)} - \ln f(\mathbf{x})_y \Leftarrow -(\mathbf{e}(y) - f(\mathbf{x})_y) \tag{5.6}$$

Subséquemment, on calcule le gradient des paramètres, la matrice de poids et le biais, de la couche cachée traitée à l'aide des équations 5.7 et 5.8.

$$\nabla \mathbf{W}^{(k)} - \ln f(\mathbf{x})_y \Leftarrow (\nabla \mathbf{a}^{(k)} - \ln f(\mathbf{x})_y) {\mathbf{h}^{(k-1)}}^T \tag{5.7}$$

$$\nabla \mathbf{b}^{(k)} - \ln f(\mathbf{x})_y \Leftarrow (\nabla \mathbf{a}^{(k)} - \ln f(\mathbf{x})_y) \tag{5.8}$$

Ensuite, on effectue le calcul des gradients de la couche cachée à l'aide des équations 5.9 et 5.10.

$$\nabla \mathbf{h}^{(k)} - \ln f(\mathbf{x})_y \Leftarrow {\mathbf{W}^{(k)}}^T (\nabla \mathbf{a}^{(k+1)} - \ln f(\mathbf{x})_y) \tag{5.9}$$

$$\nabla \mathbf{a}^{(k)} - \ln f(\mathbf{x})_y \Leftarrow (\nabla \mathbf{h}^{(k)} - \ln f(\mathbf{x})_y) \odot \mathbf{g}(\mathbf{a}^{(k)})) \tag{5.10}$$

On répète ce calcul de gradients des paramètres et des couches cachées pour toutes les couches cachées comprises dans le réseau de neurones.

Une fois la rétropropagation des erreurs effectuée, on peut procéder à mettre à jour le réseau de neurones. En d'autres termes, l'ensemble des matrices de poids reliant les diverses couches ensembles et des biais, dénotés θ, sont modifiées en utilisant l'équation 5.11.

$$\theta = \theta + \alpha\delta \tag{5.11}$$

Brièvement, cette formule indique que chaque matrice de poids reliant deux couches du réseau de neurones ensemble et chaque biais sont mise à jour à l'aide des gradients des erreurs et du taux d'apprentissage α.

Ce processus complet est répété pour toutes les données sur lesquelles le réseau de neurones a à s'entraîner. Un passage complet sur toutes les données d'entraînement est appelé une époque. Il est courant que le processus d'entraînement complet comporte plusieurs époques. On arrête normalement ce processus lorsque le réseau de neurones

semble commencer à se sur-entraîner sur un ensemble de données particulier. On détecte ce phénomène en observant l'erreur sur l'ensemble de validation.

5.2.2 Réseau de neurones parallèle

Le processus d'apprentissage dans un réseau de neurones présente plusieurs opportunités de parallélisme à fine granularité. Ceci est dû au fait qu'il est constitué principalement de multiplication de matrices et de vecteurs. Cet algorithme est donc un bon candidat pour la parallélisation sur systèmes à mémoire partagée tels que les processeurs multi-coeurs et les processeurs graphiques.

Il y a par contre un problème avec l'utilisation de tels systèmes dans le cadre de l'apprentissage parallèle dans un réseau de neurones: celui de la mémoire. En effet, la mémoire de ces systèmes, particulièrement celle des processeurs graphiques, est bien souvent insuffisante lorsque l'on veut mettre en place des réseaux de neurones de taille colossale. Dans une telle situation, le seul choix qui s'offre est celui de paralléliser le réseau de neurones sur un système à mémoire distribuée.

Une première façon d'implémenter un tel réseau sur systèmes distribués consiste à conserver le réseau de neurones dans sa forme de base et de l'augmenter avec une partition de ses structures internes. En d'autres termes, il s'agit de distribuer et de regrouper les structures de données impliquées dans une étape du processus d'apprentissage, et ce, à chaque étape de ce processus. Plus précisément, la couche cachée que l'on calcule, la couche cachée précédente et la matrice de poids liant ces deux couches doivent être traitées de façon distribuée. Ceci implique que pour faire fonctionner le calcul, il faut aussi recalculer les bornes des boucles impliquées pour qu'elles n'agissent que sur les données appartenant aux processus sur lesquels elles s'exécutent. Aussi, au niveau des données, il faut ajouter des instructions pour diviser et regrouper les données avant et après chaque étape du processus. Une telle façon de faire implique que l'initialisation des structures du réseau de neurones doit se faire sur un seul processus qui est désigné comme maître. Cette approche est celle préconisée dans l'évaluation de *Clang-MPI* compte tenu des modifications qu'il est capable d'effectuer sur un programme donné.

Une seconde façon d'implémenter un réseau de neurones distribué est de modifier

explicitement l'algorithme d'apprentissage pour qu'il prenne avantage du système sur lequel il est mis en oeuvre. L'algorithme d'apprentissage profite alors d'un parallélisme d'une plus forte granularité que celle de l'approche précédente.C'est dans cette avenue que les travaux de Google des dernières années s'inscrivent, notamment la plateforme *DistBelief* [Dea12]. Brièvement, cette approche est principalement basée sur du parallélisme de modèles et non pas du parallélisme à l'intérieur même du modèle. En d'autres termes, le modèle que l'on souhaite entraîner est répliqué plusieurs fois et les copies produites participent toutes à l'entraînement du modèle. Pour ce faire, elles communiquent entre elles les mises à jour qu'elles font à leurs paramètres par le biais d'un serveur de paramètres centralisé qui a pour tâche de regrouper toutes ces mises à jour. Cette approche étant intrinsèquement parallèle, elle ne fut pas retenue. En effet, il n'y a pas moyen de l'exprimer de façon séquentielle et, par conséquent, pas moyen de s'en servir comme base dans une expérience de parallélisation automatique.

5.2.3 Résultats

Dans le cadre des expériences pour tester l'applicabilité de *Clang-MPI* à la parallélisation du processus d'apprentissage dans un réseau de neurones, une version séquentielle de l'algorithme fut d'abord implémentée. Cette dernière devait remplir plusieurs rôles. Le premier de ceux-ci était d'aider à garantir le bon fonctionnement de l'implémentation de l'algorithme compte tenu qu'elle constitue sa version la plus simple. Le second de ces rôles est de servir de base pour le processus de parallélisation automatique qu'on désire tester. Le dernier rôle de cette version est de fournir un jalon contre lequel on peut juger de la pertinence de paralléliser l'algorithme d'apprentissage sur la base du temps nécessaire à l'entraînement du réseau de neurones.

Une seconde version fut aussi implémentée pour servir de point de comparaison dans un contexte parallèle. On veut, au travers ce programme, être capable de juger de la performance du programme parallélisé automatiquement face à un programme écrit à la main par un développeur. Évidemment, cette version a été implémentée à l'aide de la bibliothèque MPI.

Au final, les tentatives de parallélisation automatique d'un réseau de neurones sur un système distribué se sont toutes soldées par des échecs. Ainsi, bien que la

version séquentielle fut vérifiée pour garantir qu'elle était effectivement capable d'apprendre, lorsque les *pragmas* étaient mis en place, le réseau se remplissait vite de *NaN* contrecarrant par le fait même tout le processus d'apprentissage.

5.2.4 Discussion

L'échec de la parallélisation automatique d'un réseau de neurones est due à plusieurs facteurs. Le premier est que *Clang-MPI* éprouve rapidement des difficultés à paralléliser des imbrications de boucles dans lesquels il a plus d'une boucle annotée avec un *pragma*. En effet, il existe plusieurs situations pour lesquelles *Clang-MPI* n'arrive pas à produire le résultat attendu.

Une de celle-ci se manifeste lorsqu'un *pragma* est appliqué autant à la boucle externe d'un imbrication qu'à une boucle interne de la même imbrication. Ainsi, et comme il l'a été expliqué plus en profondeur au chapitre 3, l'application d'un *pragma* à une boucle fait en sorte que tout son contenu, toutes les boucles internes qui la compose, est analysé pour générer les appels adéquats à des fonctions MPI. Aussi, on s'assure, au moyen d'instructions nommées, que la même structure n'est pas distribuée à plusieurs reprises. En tenant compte de ces deux faits, on peut donc comprendre qu'il risque d'y avoir certaines interférences entre une boucle externe et une boucle interne car leurs analyses se recoupent. Ceci a comme conséquence que certaines distributions de données seront omises, ce qui risque d'entraîner une détérioration du temps de calcul. Compte tenu que l'algorithme d'apprentissage dans un réseau de neurones comporte de telles situations, cela peut en partie expliquer l'échec de paralléliser un réseau de neurones en utilisant *Clang-MPI*. Un exemple concret est donné au Programme 5.1.

On constate ici que l'on tente de paralléliser autant la boucle externe qui parcoure la structure *Layer* que la boucle interne qui parcoure la structure *Input*. On remarque aussi que les deux boucles sont utilisées pour parcourir la structure *Weight*. C'est d'ailleurs cette dernière structure empêcher toute parallélisation. De fait, ce processus est tout d'abord appliqué à la boucle interne. La structure *Weight* est distribué selon la variable d'induction *j*, ce qui correspond ici aux colonnes. L'intervalle de la boucle est également modifié pour chacun des processus de façon à ce qu'elle soit cohérente

```
// Calcul de la pre-activation d'une couche cachee
#pragma clang loop distribute(worker)
  for (int i = 0; i < M; ++i)
#pragma clang loop distribute(worker)
    for (int j = 0; j < N; ++j)
      Layer[i] += Input[j] * Weight[i * M + j];
```

Programme 5.1 – Exemple d'utilisation de *Clang-MPI* sur une boucle externe et une boucle interne appartenant au même imbriquement de boucles

avec l'exécution prévue. Par la suite, *Clang-MPI* parallélise la boucle externe. C'est à ce point que le problème se manifeste. Ainsi, les intervalles des boucles sont modifiées par le processus de parallélisation sans tenir compte l'une de l'autre, sauf pour ne pas redistribuer des données déjà distribuées. Ce faisant, certains processus risquent de ne pas avoir les données nécessaires pour effectuer correctement leur travail.

Mentionnons que cette version de la multiplication vecteur-matrice a été la première mise en oeuvre dans le cadre des expériences de la parallélisation automatique d'un réseau de neurones. Compte tenu des problèmes rencontrés, une autre approche a été tentée pour effectuer la parallélisation automatique d'un réseau de neurones. Dans cette approche, seules les boucles les plus externes contenues dans la boucle des époques sont annotées d'un *pragma*. Les autres *pragmas* ont été enlevés. Cette approche n'a pas donné les résultats attendus.

En effet, pour renchérir sur le sujet d'interférence, une situation bien plus grave survient lorsque l'on tente de paralléliser au moins deux boucles qui sont internes à une boucle externe. Ainsi, dans un tel contexte, les boucles partagent un même bloc de pré-entête, ce qui rend l'interférence lors de la génération du code de communication encore plus problématique, voire carrément ingérable. Ainsi, un cas d'interférence dans une telle situation risque de faire en sorte que les structures contenant des résultats temporaires soient distribuées avant même qu'elles aient reçues ces résultats temporaires. Ceci fait évidemment en sorte que le calcul final est faussé si le programme est encore en mesure de le produire. En effet, dans un tel cas, on a autant de chance d'observer un résultat faux que d'observer le programme terminé à cause d'une erreur de segmentation. Un exemple d'une telle situation, provenant de la dernière version du réseau de neurones écrit dans le cadre de nos expériences, est donné

```
// Entrainement du reseau de neurones pour NB_EPOCHS epoques
for (int i = 0; i < NB_EPOCHS; ++i) {

  /* ... */

// Initialisation de la couche cachee
#pragma clang loop distribute(worker)
  for (int j = 0; j < M; ++j)
    Layer[j] = 0.0;

// Pre-activation de la couche cachee
#pragma clang loop distribute(worker)
  for (int j = 0; j < M; ++j)
    for (int k = 0; k < N; ++k)
      Layer[j] += Input[k] * Weight[j * M + k];

// Activation de la couche cachee
#pragma clang loop distribute(worker)
  for (int j = 0; j < M; ++j)
    Layer[j] = 1.0 / (1.0 + exp(-Layer[j]));
}

  /* ... */

}
```

Programme 5.2 – Exemple d'utilisation du pragma *master*

au Programme 5.2.

La situation dans le Programme 5.2 est que l'on tente de paralléliser trois boucles distinctes mais partageant la même boucle externe. De plus, ces trois boucles vont venir modifier la même structure de données soit *Layer*. Dans une telle situation, l'algorithme de parallélisation de *Clang-MPI* fait en sorte que la distribution et le regroupement des données de la structure *Layer* ne se produit que pour la première des trois boucles. En effet, lors de la parallélisation des boucles subséquentes, *Clang-MPI* détecte que des appels à des fonctions MPI visant à manipuler *Layer* ont été produits. Cette détection se fait en observant les instructions du bloc de pré-entête de

la boucle à paralléliser. Tel qu'énoncé précédemment, les boucles d'un même niveau dans un imbrication de boucles partagent le même bloc de pré-entête. Par conséquent, dans le but de minimiser les communications, *Clang-MPI* ne génère pas d'appels à des fonctions MPI pour les autres boucles.

Le problème majeur ici est que la solution au problème de la parallélisation des boucles internes n'est vraiment pas évident. Par exemple, une solution envisageable est de générer les appels aux fonctions MPI à l'intérieur du bloc d'entête d'une boucle, car on sait que ce bloc est unique à chaque boucle. Bien que cette solution soit possible, elle n'est pas désirable, car les instructions contenues dans ce bloc sont assurément exécutées une fois par tour de boucle. On peut donc concevoir l'explosion du temps de communication d'un programme parallélisé de la sorte.

On peut aussi être tenté, dans le cas des boucles d'un même niveau dans un imbrication de boucles, de générer des appels aux fonctions MPI dans des blocs de sortie de boucle choisis. Cette approche n'est pas une solution aux problèmes éprouvés compte tenu que, dans un tel cas, le bloc de sortie d'une boucle correspond au bloc d'entête de la boucle suivante. On se retrouve alors avec le même problème que précédemment.

Une autre solution à ce problème est de modifier les *pragmas* pour qu'ils soient une entité à part entière du langage comme le sont ceux d'OpenMP et non une simple annotation de boucle comme ils le sont présentement. Ceci permettrait une plus grande flexibilité et précision dans le parallélisme qu'ils peuvent exprimer et mener à une représentation intermédiaire plus facile à paralléliser automatiquement. Cependant, un projet d'une telle envergure est inconcevable dans le temps limité d'une maîtrise. À titre de comparatif, il a fallu de trois à quatre ans à une équipe de développeurs professionnels pour implanter le standard OpenMP dans le compilateur *Clang*. Il est raisonnable de croire qu'il faudrait au moins autant de temps pour implanter un prototype basé sur ce modèle de *pragmas* agissant dans le contexte des systèmes à mémoire distribuée utile pour paralléliser un réseau de neurones.

Bien que cette limitation courante soit navrante, il faut ici se rappeler le type de système visé par l'outil développé dans le cadre de cette maîtrise. En effet, en se fixant comme but de paralléliser des programmes pour qu'ils exécutent sur des systèmes à mémoire distribuée, on se fixe aussi indirectement le but de minimiser

le temps passé à communiquer des données entre les divers processus exécutant le programme. Dans cette optique, il est parfaitement acceptable de se concentrer tout d'abord sur la parallélisation du niveau le plus externe d'un imbrication de boucles. De fait, c'est à ce niveau qu'une itération comporte le plus d'opérations à effectuer et donc, potentiellement, le moins de communication entre les divers processus.

Un autre facteur bien important dans l'explication de cet échec est le domaine d'application visé, les réseaux de neurones. Tel que mentionné dans la présentation des réseaux de neurones parallèles, l'algorithme de base pour le faire apprendre est intrinsèquement séquentiel. En d'autres termes, on y retrouve beaucoup de dépendances envers des calculs précédents, entre autres les calculs de couches cachées précédentes, qui ne semblent pas bien s'accorder avec *Clang-MPI*, surtout lorsque ces calculs font partie de boucles internes. Un meilleur choix d'application "réelle" pouvant tirer profit de *Clang-MPI* aurait donc été de mise. Il faut par contre avouer qu'à la vue de ce facteur, cet échec n'est pas une surprise. En effet, les implémentations de réseau de neurones distribués sont rares et celles qui parviennent à obtenir des résultats intéressants y sont parvenus en modifiant d'une façon ou d'une autre l'algorithme d'apprentissage. Un exemple est le système *DistBelief* [Dea12] mis au point par Google qui a été décrit précédemment.

Conclusion

Dans ce mémoire, nous avons présenté l'outil de parallélisation automatique *Clang-MPI*. Cet outil permet la parallélisation automatique sur systèmes à mémoire distribuée de code C/C++ en annotant des boucles choisies avec un *pragma*. Ce processus de parallélisation s'occupe de partitionner les boucles annotées ainsi que les structures de données utilisées dans la boucle. Pour effectuer ce travail, *Clang-MPI* produit d'abord du code en LLVM IR. Ce code est ensuite analysé et transformé pour y intégrer des appels de fonctions MPI qui s'occupent des besoins de communication du programme produit pour système distribué. Par après, on laisse le soin à LLVM de terminer le processus de compilation en produisant du code pour l'architecture pour laquelle le programme est compilé.

Avant d'arriver à cet outil, deux essais majeurs ont été effectués pour tenter de paralléliser automatiquement des programmes scientifiques sur des systèmes à mémoire distribuée. Le premier de ces essais consistait à utiliser une plateforme de compilation polyédrale, *Polly*, pour arriver à cette fin. L'idée était d'utiliser les outils fournis par cette plateforme pour découvrir comment un programme donné pourrait être partitionné sur un système distribué. Par la suite, à l'aide d'un nouveau générateur de code MPI, le programme aurait été parallélisé en se servant des résultats de l'analyse précédente. Cette approche ne fut pas un succès, et ce, pour plusieurs raisons. Les plus importantes concernent la nouveauté et la difficulté intrinsèque du problème que l'on tentait de résoudre ainsi que le manque de maturité de la plateforme choisie.

Le second de ces essais consistait à créer une variation de l'algorithme de parallélisation automatique connu sous le nom de *Decoupled Software Pipelining* qui pourrait le transposer dans le cadre des systèmes à mémoire distribuée. Ainsi, cet algorithme a pour but de créer des programmes parallèles orientés tâches à partir

de programmes originalement séquentiel. Ceci semblait donc être un bon candidat pour la parallélisation automatique visant les systèmes à mémoire distribuée. Malheureusement, l'incomplétude des passes d'analyse de dépendances mémoires offertes par LLVM ainsi que la difficulté de dégager des partitions intelligentes des données impliquées dans le programme firent en sorte que cet essai se solda lui aussi par un échec.

Suite à ces deux essais, nous avons abandonné l'idée de paralléliser automatiquement des programmes de façon totalement implicite. Ceci m'a amené à développer *Clang-MPI*, un outil de parallélisation automatique utilisant des annotations de boucles (*pragmas*). Cet outil est celui sur lequel ce mémoire à principalement porter.

Pour évaluer le travail fait avec *Clang-MPI*, des expériences avec le banc d'essais *PolyBench* ont été effectuées. Ces dernières ont démontrées que *Clang-MPI* pouvait être compétitif avec des implémentations parallélisées manuellement en utilisant la bibliothèque MPI dans certains cas. Dans d'autres cas, par contre, *Clang-MPI* produit des programmes dont le temps d'exécution est beaucoup plus important que les programmes qui utilisaient uniquement MPI.

De plus, au niveau du but principal de cette maîtrise, c'est-à-dire la parallélisation automatique d'un algorithme d'apprentissage dans un réseau de neurones, force est de constater que ce projet fut un échec. Ainsi, l'ajout de *pragmas* à une implémentation séquentielle de l'algorithme d'apprentissage dans un réseau de neurones faisait en sorte que le programme ne produisait plus de bons résultats. Comme il a été dit au chapitre 5, les raisons de cet échec sont liées à la forme de l'algorithme qui ne se prête pas facilement au contexte des systèmes parallèles à mémoire distribuée et des choix d'architectures faits tout au long du projet.

Ainsi, à la vue des résultats obtenus, on peut constater plusieurs faiblesses ou limitations de *Clang-MPI*. En fait, plusieurs de ces limitations avaient déjà pu être inférées lors de la phase de conception de l'outil de parallélisation automatique parce qu'ils découlent de choix faits à cette étape. Le meilleur exemple de ce constat est l'incapacité de *Clang-MPI* à paralléliser des boucles internes. De surcroît, il accuse une limitation très évidente dans le fait que les seules parties d'un programme qu'il peut paralléliser sont les boucles présentes dans ce dernier. De plus, il ne peut effectuer son travail de parallélisation automatique qu'en considérant chaque imbriquement de

boucles de façon indépendante. Ceci fait en sorte qu'il génère beaucoup plus d'appels d'échange de messages qu'il est nécessaire, expliquant ainsi les résultats obtenus avec *PolyBench*. De surcroît, *Clang-MPI* n'effectue aucune analyse de dépendances ce qui le contraint à ne pouvoir paralléliser que des boucles dans lesquelles le programmeur est confiant qu'elles ne contiennent pas de dépendances portées par la boucle. Bref, l'outil, dans son état actuel, demeure très limité dans son champ d'application.

Pour aller de l'avant, le projet devrait changer de façon significative. Tout d'abord, si l'approche par annotations est conservée, il faudra étendre les annotations de façon à ce qu'elles puissent être appliquées non pas seulement sur des boucles, mais aussi sur des portées définies par les programmeurs. De plus, le contenu même des annotations devra lui aussi être enrichi pour qu'un programmeur puisse indiquer comment les données doivent être partitionnées et quels sont les éléments qui doivent partagés ou non. En d'autres termes, ces annotations devraient ressembler beaucoup plus à ce que *OpenMP* offre, mais pour des systèmes à mémoire distribuée. Il faut par contre garder en tête que ce travail est colossal et que des problèmes, comme l'analyse efficace de dépendances mémoire et la distribution efficace de données dans le but de minimiser les communications entre les noeuds, demeurent entiers et risquent de conduire au même constat d'échec.

Bibliographie

[Aho06] Aho, A. V., Lam, M. S., Sethi, R. et Ullman, J. D. : Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[All02] Allen, R. et Kennedy, K : Optimizing Compilers for Modern Architecture. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[Bag10] Baghdadi, S., Grosslinger, A. et Cohen, A. : Putting automatic polyhedral compilation for GPGPU to work. *In* Proceedings of the 15th Workshop on Compilers for Parallel Computers, Vienne, Austriche, juillet 2010.

[Bas04] Bastoul, C. : Improving Data Locality in Static Control Programs. Thèse de doctorat, Université Paris 6, Pierre et Marie Curie, France, France, décembre 2004.

[Ben07] Benson G. et Fedosov, A. : Python-based distributed programming with Trickle. *In* Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '07, pages 30–36. CSREA Press, 2007.

[Ber10] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G. et al : Theano: a CPU and GPU math expression compiler. *In* Proceedings of the Python for Scientific Computing Conference (SciPy), juin 2010.

[Bon02] Bonachea, D. : Gasnet. Specification, Berkeley, CA, USA, octobre 2002. Version 1.1.

[Bon11] Bondhugula, U. : Automatic Distributed Memory Code Generation Using the Polyhedral Framework. Rapport technique, Indian Institute of Science, 2011.

[Bon13] U. 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, pages 33:1–33:12, New York, NY, USA, 2013. ACM.

[Cha07] Chamberlain, B. L., Callahan, D. et Zima, H. P. : Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications, 21(3):291–312, août 2007.

[Cho04] Choy, R., Edelman, A., Gilbert, J. R., Shah, V. et Cheng, D : Star-P: High productivity parallel computing. *In* In 8th Annual Workshop on High-Performance Embedded Computing (HPEC 04), 2004.

[Con12] Continuum Analytics : Numba. http://numba.pydata.org, 2012.

[Cou77] Cousot, P. et Cousot, R. : Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In* Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[Dea12] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V. et al : Large scale distributed deep networks. *In* Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States., pages 1232–1240, 2012.

[Fer87] Ferrante, J., Ottenstein, K. J. et Warren, J. D. : The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319–349, juillet 1987.

[For09] Message Passing Interface Forum : MPI: A message-passing interface standard. Specification, septembre 2009. Version 2.2.

[Fou14] LLVM Foundation : Clang. http://clang.llvm.org, 2014.

Bibliographie

[Goo09] Google : Unladen Swallow - a faster implementation of Python. https://code.google.com/p/unladen-swallow/, 2009.

[Gri04] Griebl, M. : Automatic Parallelization of Loop Programs for Distributed Memory Architectures. Université de Passau, 2004.

[Gro11] Grosser, T. : Enabling polyhedral optimizations in LLVM. Mémoire de D.E.A., Université de Passau, Allemagne, avril 2011.

[Kar13] Karrenberg, R., Koster, M., Leissa, R., Yevgeniya, K. et Hack, S. : Noise: User-defined optimization strategies. *In* European LLVM Conference, Paris, France, avril 2013.

[Ken94] Kennedy, K. et Mckinley, K. S. : Typed fusion with applications to parallel and sequential code generation. Rapport technique, 1994.

[Ken07] Kennedy, K., Koelbel, C. et Zima, H. P. : 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, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[Kri13] Kristensen, M. R. B., Lund, S. A. F., Blum, T., Skovhede, K. et Vinter, B. : Bohrium: unmodified NumPy code on CPU, GPU, and cluster. *In* PyHPC '13: Proceedings of the 3rd Workshop on Python for High-Performance and Scientific Computing, New York, NY, USA, novembre 2013. ACM.

[Kru14] Kruse, M. : Introducing Molly: Distributed memory parallelization with LLVM. Computing Research Repository, abs/1409.2088, 2014.

[Lat04] Lattner, C. et Adve, V. : LLVM: A compilation framework for lifelong program analysis and transformation. *In* Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, pages 75–88, Washington, DC, USA, 2004. IEEE Computer Society.

[Li,] Li, F., Pop, A. et Cohen, A : Automatic extraction of coarse-grained data-flow threads from imperative programs. IEEE Micro, 32(4):19–31.

[Mel02] Mellor-Crummey, J. M., Adve, V. S., Broom, B., Chavarría-Miranda, D. G., Fowler, R. J., Jin, G. et al : Advanced optimiza-

tion strategies in the Rice dHPF compiler. Concurrency and Computation: Practice and Experience, 14(8-9):741–767, juillet 2002.

[Mid12] Midkiff, S. P. : Automatic Parallelization:An Overview of Fundamental Compiler Techniques. Morgan & Claypool, 2012.

[Mor98] Morgan, R. : Building an Optimizing Compiler. Digital Press, Newton, MA, USA, 1998.

[Num98] Numrich, R.W. et Reid, J. : Co-array fortran for parallel programming. SIGPLAN Fortran Forum, 17(2):1–31, août 1998.

[Oli07] Oliphant, T. E. : Python for scientific computing. Computing in Science and Engineering, 9(3):10–20, mai 2007.

[Ope13] OpenMP Architecture Review Board : OpenMP application program interface. Specification, juillet 2013. Version 4.0.

[Ott05] Ottoni, G., Rangan, R., Stoler, A. et August, D. I. : Automatic thread extraction with decoupled software pipelining. *In* Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[Pou10] Pouchet, L. N. : Polyhedral compilation foundations. Ohio State University, janvier 2010. présentation orale.

[Pou12] L. N. Pouchet : PolyBench: The polyhedral benchmark suite. http://www.cs.ucla.edu/~pouchet/software/polybench/, 2012.

[Rag11] Ragesh A. : A framework for automatic openMP code generation. Mémoire de D.E.A., Indian Institute of Technology, India, avril 2011.

[Rag13] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. et Amarasinghe, S. : Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *In* Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[Ram08] Raman, E., Ottoni, G., Raman, A., Bridges, M. J. et August, D. I. : Parallel-stage decoupled software pipelining. *In* Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[Ram13] Ramashekar, T. et Bondhugula, U : Automatic data allocation and buffer management for multi-GPU machines. ACM Transactions on Architecture and Code Optimization, 10(4):60:1–60:26, décembre 2013.

[Red14] Reddy, C. et Bondhugula, U : Effective automatic computation placement and data allocation for parallelization of regular programs. *In* Proceedings of the 28th ACM International Conference on Supercomputing, ICS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[Rub12] Rubinsteyn, A., Hielscher, E., Weinman, N. et Shasha, D. : Parakeet: A just-in-time parallel accelerator for Python. *In* Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism, HotPar'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[Sar12] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O., Grove, D., Grove, D. et al : X10 language specification. Specification, IBM, janvier 2012. Version 2.2.

[Sim14] Simbürger, A. et Grösslinger, A. : On the variety of static control parts in real-world programs: From affine via multi-dimensional to polynomial and just-in-time. Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, 2014.

[Tom13] Tomofumi, Y : Beyond Shared Memory Loop Parallelism in the Polyhedral Model. Thèse de doctorat, Colorado State University, USA, 2013.

[Wel96] Wells, J. B. : Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. *In* Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, pages 176–185. Society Press, 1996.