Les Compilateurs

Paul Feautrier
Ecole Normale Supérieure de Lyon
46 Allée d'italie
69364 LYON CEDEX 07 FRANCE
Paul.Feautrier@ens-lyon.fr

10 mars 2009

Résumé

On présente l'état de l'art et l'évolution récentes des techniques de compilation

1 Introduction

La paresse est la première qualité de l'informaticien. L'informatique ne s'est développée que parce que certains en ont eu assez de faire des multiplications en série ou de trier des fiches en carton. Les ordinateurs ont permis de remplacer ces activités répétitives par l'écriture d'un programme. Mais l'activité de programmation est elle-même en partie répétitive, et le principe du moindre effort veut qu'elle soit automatisée dans la mesure du possible.

L'utilisation d'un compilateur permet de programmer dans un langage de haut niveau, plus concis et moins sujet à erreur que le langage machine. Un programme écrit dans un tel langage doit être traduit pour être exécuté. Ce processus est maintenant bien compris ; la section 2 sera consacrée à la présentation de ses différentes étapes. Mais depuis les origines, c'est-à-dire la réalisation du premier compilateur FORTRAN par John Backus, la qualité de cette traduction est un souci permanent : il s'agit de ne pas trop perdre de performance par rapport à la programmation en langage machine. Ce sont les techniques d'optimisation qui permettent de limiter cette perte de performance, et ce sont sur elles que se concentre actuellement la recherche. Je leur consacrerai la suite de cet exposé.

2 Techniques de compilation

2.1 Eléments fondamentaux d'un langage de programmation

Je suppose ici que mes lecteurs ont déja rencontré un langage de programmation, que ce soit l'un des classiques – FORTRAN, Pascal, C, etc. – ou un langage plus spécialisé comme SQL, langage pour l'interrogation des bases de données. Un compilateur est un traducteur, et la première chose à comprendre, c'est ce qu'il doit faire pour engendrer un programme exécutable.

A quoi ressemble un langage de haut niveau ? Un hybride entre le langage naturel et la notation algébrique ? Mais aussi bien le langage naturel que la notation algébrique sont ambigus et imprécis, et ce n'est que par une longue pratique que l'on apprend à s'en servir avec aisance. Au contraire, un langage de programmation a une grammaire rigide, et une sémantique qui serait presque aussi rigide si des raisons techniques ne venaient interférer. Cette grammaire et cette sémantique doivent être non ambigües, ce qui n'est pas si facile à obtenir.

Comme dans la notation algébrique, les données manipulées sont désignées par des symboles (les *identificateurs*) et non pas citées. A la différence de l'algèbre, elles peuvent être organisées en structures complexes. Au contraire, l'ordinateur cible ne connait et ne manipule que des chaînes de bits, à la rigueur des entiers.

Les manipulations de données sont spécifiées par des instructions complexes, qui peuvent utiliser plusieurs étages d'opérations arithmétiques ou autres. Le matériel, lui, n'exécute jamais qu'une opération à la fois.

Enfin, alors que les ordinateurs ne peuvent prendre que des décisions binaires, les langages de haut niveau disposent d'instructions de contrôle sophistiquées, qui peuvent provoquer des répétitions (boucles) ou des choix multiples (*switch*).

En résumé, pour compiler un programme, il faut tout d'abord en faire l'analyse grammaticale. Vient ensuite la construction de l'image de la mémoire du programme cible, et enfin la génération du programme équivalent en langage machine. Entre chacunes de ces phases peuvent s'insérer des étapes d'optimisation.

2.2 Analyse syntaxique

Les grammaires utilisées pour les langages de programmation ont été simplifiées à l'extrême. On les appelle "hors contexte " parce que leurs règles sont purement locales et ne dépendent pas du contexte

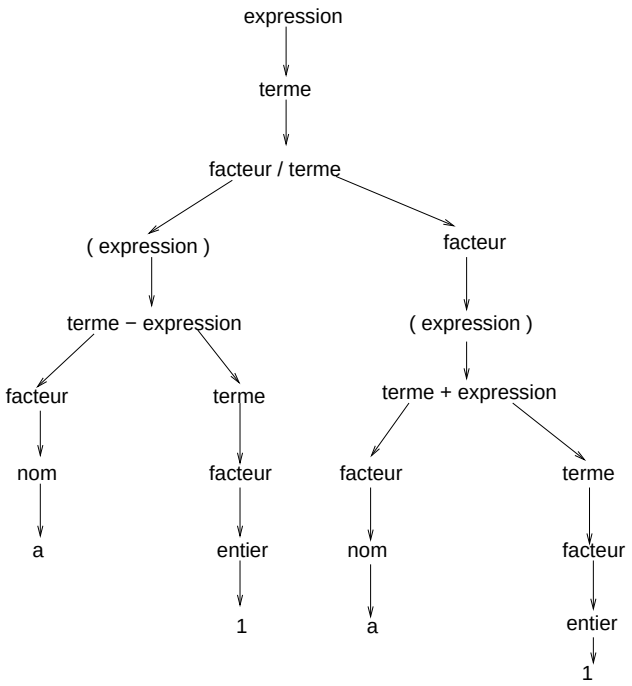

FIG. 1 – Un arbre d'analyse grammaticale

dans lequel elles sont appliquées.

Par exemple, un fragment de la grammaire des expressions arithmétiques pourrait être :

```
facteur ::= identificateur | entier | ( expression )
terme ::= facteur | facteur * terme
                  | facteur / terme
expression ::= terme | terme + expression
                     | terme - expression
```

Pour compléter cette grammaire il faudrait expliquer ce qu'est un identificateur et ce qu'est un entier : ces détails sont laissés à l'imagination du lecteur.

Si `a` est un identificateur et `1` un entier, cette grammaire engendre l'expression `(a - 1)/(a + 1)` de la façon suivante : tout d'abord, `a` est un facteur, puis un terme d'après les 2 premières règles appliquées en succession. De même, `1` est un facteur, puis un terme puis une expression. La dernière règle indique alors que `a-1` et `a+1` sont des expressions, puis que `(a-1)` et `(a+1)` sont des facteurs. On en déduit de la même façon l'expression finale. La figure 1 est une représentation intuitive du schéma d'analyse grammaticale (*parse*

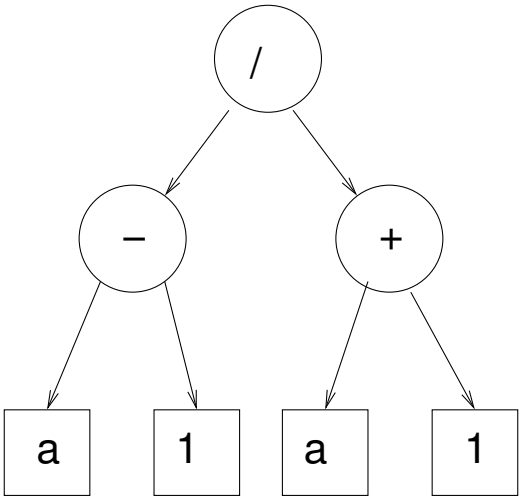

FIG. 2 – Représentation intermédiaire

tree) de l'expression étudiée.

Pour chaque grammaire hors contexte non ambigüe, on peut construire un algorithme qui, étant donné un texte, décide s'il est correct et donne son arbre d'analyse. Mais on peut aller plus loin : pour toute une classe de grammaires, il est possible de construire automatiquement cet analyseur. Le plus connu de ces "compilateurs de compilateur" est Yacc. Pour sa description on se reportera au traité [AHO 86].

2.3 La représentation intermédiaire

Comme l'exemple l'a montré, l'un des effets de l'analyse grammaticale est de découper le texte du programme en fragments cohérents, que l'on traitera ensuite de façon plus ou moins autonome. Par exemple, l'une des questions que l'on peut se poser au sujet de l'expression `(a-1)/(a+1)` est : quel est le premier opérande de l'addition ? C'est l'analyse grammaticale qui permet de trouver que c'est l'identificateur `a`, et nom pas le texte `(a-1)/(a`, qui d'ailleurs n'a pas de sens. La représentation intermédiaire sert à stocker ce type d'information. Les compilateurs modernes utilisent pour cela l'arbre d'analyse syntaxique (souvent simplifié). Le diagramme de la figure 2 est beaucoup plus économique que celui de la figure 1 mais rend à peu près les mêmes services.

Toutefois, la représentation syntaxique n'est pas toujours optimale. Il n'est pas évident, par exemple, de trouver l'instruction qui suit une instruction donnée. On utilise pour cela une autre représentation, le

graphe de contrôle, version au goût du jour des anciens organigrammes. De même, il faudra une troisième représentation si l'on souhaite simplifier les expressions arithmétiques, par exemple pour remplacer `0 * x` par `0`.

Au total, on trouve dans un compilateur plusieurs représentations intermédiaires, dont il faut garantir la cohérentce, ce qui n'est pas une mince affaire.

2.4 Optimisation

Cette phase consiste en des transformations de la RI, qui doivent améliorer les performances du programme sous la contrainte que les résultats n'en soient pas modifiés. Certaines optimisations sont indépendantes de la cible : il est (presque) toujours bénéfique de ne pas refaire plusieurs fois le même calcul. D'autres, au contraires, dépendent de l'état courant de l'architecture informatique : ce n'est que depuis que les ordinateurs ont des caches que l'on se préoccupe de la gestion de la localité, et ces recherches deviendront inutiles si l'on trouve d'autres moyens pour compenser la lenteur des accès à la mémoire. Les techniques d'optimisation seront présentées en détail en section 3.

2.5 Gestion de la mémoire

Dans les langages modernes, toute donnée doit être *déclarée*. Par exemple, en C on écrira :

```
short int n[10];
float x;
char c;
```

pour indiquer que `n` désigne un tableau de dix entiers codés sur deux octets, que `x` est un nombres codé en "virgule flottante" (quatre octets) et que `c` est un caractère (un octet). A partir de ces indications, le compilateur peut calculer simplement la taille de chaque objet et lui affecter une place en mémoire. En général, l'algorithme d'affectation est très simple et suit servilement l'ordre des déclarations. Si on suppose que la déclaration de `n` est la première du programme, on le placera à l'adresse 0, et il occupera 20 octets. `x` viendra à l'adresse 20, et `c` à l'adresse 24. Ceci fait, le compilateur disposera d'un dictionnaire, où, en face de chaque objet, on trouve son adresse.

Mais dans les langages modernes, la durée de vie des données peut être variable. Les données qui existent pendant toute la durée du programme sont dites globales. Par contre, les données utilisées par une fonction n'existent que lorsque la fonction est active ; elles sont détruites quand la fonction se termine. Le principe de l'affectation de ces

données temporaires est le même que ci-dessus, mais les adresses se réfèrent à la *fenêtre d'activation* de la fonction. Elles changent d'une activation à l'autre. Enfin, il existe des données *dynamiques*, dont la création et la destruction sont sous le contrôle du programmeur ; le compilateur les ignore.

2.6 Génération de code

Très généralement, il s'agit d'appliquer une suite de transformations à la représentation intermédiaire jusqu'à la rendre équivalente à un programme en langage machine ou en langage d'assemblage, qu'il n'y a plus qu'à imprimer. La phase de transformation doit tenir compte des particularités de la machine cible. On peut en faire une implémentation *ad hoc*, et il faudra la reprendre pour chaque nouveau processeur. On a cherché à construire des générateurs de code universels paramétrés par une description de la machine cible. Cette description prend en général la forme d'un système de ré-écriture de la représentation intermédiaire, parfois préalablement linéarisée (générateurs de Graham-Glanville).

> Soit à traduire l'expression `(a-1)/(a+1)` pour un processeur RISC (*reduced instruction set computer*). Une telle machine ne peut effectuer un calcul que si les deux opérandes sont dans des registres ; il existe des instructions spéciales pour copier un mot de mémoire dans un registre ou l'inverse. Le diagramme 2 montre clairement que les opérandes de la division ne sont pas des registres. On leur affecte donc deux registres, par exemple `R1` et `R2` qu'il faudra garnir convenablement. Le programme devient :
>
> ```
> R1 = a - 1;
> R2 = a + 1;
> R1 = R1 / R2;
> ```
>
> Il faut faire de même pour la soustraction, et pour l'addition. Au total, le programme est devenu :
>
> ```
> R1 = a;
> R3 = 1;
> R1 = R1 - R3;
> R2 = a;
> R4 = 1;
> R2 = R2 + R4;
> R1 = R1 / R2;
> ```
>
> A ce point, le tour est joué. Il suffit d'une transformation syntaxique très simple pour écrire `LD R1,a` à la place de la

première ligne, ou `DIV R1,R1,R2` à la place de la dernière. Dans la première instruction, on remplace `a` par l'adresse qui lui a été affectée par la phase précédente.

On remarque au passage que le code obtenu n'est pas le meilleur possible : `a` est lu deux fois, et on aurait pu utiliser moins de registres. C'est précisémment le travail de la phase d'optimisation d'éviter ou de corriger ces défauts.

3 Techniques d'optimisation

3.1 Analyse sémantique

Pour respecter la règle d'invariance des résultats, le compilateur doit avoir une compréhension, même sommaire, du fonctionnement du programme, et doit pouvoir juger de l'impact d'une modification. La recherche des informations nécessaires à cette compréhension constitue la phase d'analyse sémantique du programme. La difficulté, c'est qu'une analyse parfaite est impossible. Elle permettrait, en effet, de répondre à certaines questions (comme la terminaison ou l'équivalence) qui sont indécidables dès que le langage permet de simuler une machine de Turing. Il y a deux façons de résoudre ce problème. On peut tout d'abord se contenter d'analyses approximatives. On doit choisir soigneusement le sens de l'approximation : une analyse approximative ne doit jamais conduire à la génération d'un programme erroné. L'autre possibilité est de restreindre la classe des programmes analysés. Si ceux-ci n'ont plus la puissance de la machine de Turing (par exemple, si la terminaison est garantie), rien ne s'oppose à une analyse exacte et complète. On verra un exemple de cette situation en section 4.

Les analyses locales ont pour but de prouver que la valeur d'une variable (ou d'un groupe de variables) a une certaine propriété en un point du programme. Par exemple, si dans la portion de code :

```
real x[100] ;
    ...
S : x[i]=.. ;
```

on peut prouver la propriété $0 < i < 100$ en S, on pourra se dispenser du test de validité de l'indice. Mais dans de nombreux cas, on a besoin du concept de dépendance. Deux instructions sont en dépendance si on ne peut les intervertir sans courir le risque de changer les résultats du programme. Il y a des dépendances de données (par exemple quand une instruction utilise le résultat de celle qui la précède), et des dépendances de contrôle (on ne peut intervertir le test d'une conditionnelle

et ses branches). Les dépendances de données les plus importantes sont les dépendances de flot, qui relient entre elles les créations de nouvelles valeurs (les instructions d'affectation), et les utilisations de ces valeurs. Par exemple, dans le code :

```
S : x = 0 ;
...
T : y = x*x ;
```

on cherchera à établir que la seule source (*reaching definition*) de x dans l'instruction T est l'instruction S. Il faut pour cela montrer que la partie du code non représentée ne contient pas d'affectation à x et qu'il est impossible de parvenir en T sans passer par S. Si on y parvient, on aura démontré que $x = 0$ en T et que l'on peut remplacer cette instruction par

```
y = 0;
```

Les méthodes d'analyse des programmes sont maintenant bien comprises et ressortent toutes soit de la méthode de l'interprétation abstraite qui sont présentées ci-dessous, soit de méthodes géométriques que sont l'objet de la section 4.1

3.2 Interprétation abstraite

Il existe depuis la nuit des temps une technique, l'*interprétation*, dont le but est de simuler le fonctionnement d'une machine A à l'aide d'un logiciel qui s'exécute sur une machine B. A et B peuvent être identiques ou différents ; il se peut même que A n'existe pas ! Conçue au début comme un outil de mise au point (dans ce cas A et B sont le plus souvent identiques) cette technique a de nos jours de multiples applications. A peut être par exemple une machine future, dont le concepteur veut vérifier le bon fonctionnement. Inversement, A peut être une machine du passé, qui n'existe plus physiquement, mais dont on veut encore utiliser les logiciels. A peut être trop complexe pour être réalisable : c'est le cas des interprètes Basic de certaines calculettes. Enfin, A peut servir d'outil de portabilité : c'est le cas de la machine virtuelle de Java.

En fait, on peut associer à tout langage de programamtion de multiples interprètes, obtenus en modifiant le type des données et les opérations qui agissent sur elles. Par exemple, on peut imaginer que les variables d'un langage tel que FORTRAN, au lieu d'être des nombres, deviennent des expressions algébriques, et que les opérations de l'arithmétiques sont remplacées par celles du calcul formel. L'outil obtenu effectue l'*exécution symbolique* du programme, ce qui fournit des renseignement très utiles pour sa compréhension et sa mise au

point. Cette méthode est cependant limitée à l'analyse de programmes simples, parce que l'automatisation du calcul formel est difficile.

Mais l'idée la plus féconde est celle qui consiste à remplacer les opérations exactes du langage de programmation par des opérations approchées. Par exemple, au lieu de calculer une somme, on peut se contenter de déterminer le signe du résultat connaissant les signes des opérandes. Il suffit de réfléchir un instant à cet exemple pour voir que parmi les résultats de l'analyse, il faut prévoir le cas où le signe du résultat est indéterminé. La présence de ce tiers non exclu ("oui", "non", "je ne sais pas") est caractéristique de ce genre de méthode.

L'interprétation abstraite fournit un cadre général pour la conception systèmatique d'analyse de programmes, en général efficaces mais approximatives. Elle a été appliquée à l'étude du signe et des intervalles de variation des variables, à l'analyse du contrôle, à la recherche des sources, à la propagation des constantes, à l'analyse des pointeurs, etc. On en trouvera une présentation plus systématique dans le traité [AHO 86].

3.3 Optimisations classiques

Leur but est d'éliminer les calculs inutiles. Par exemple, la propagation des constantes consiste à effectuer au moment de la compilation tous les calculs dont les arguments sont connus, de façon à ne pas avoir à les refaire pendant l'exécution. L'optimisation la plus importante en ce genre est celle qui consister à éliminer les invariants de boucle (*hoisting*). Une expression est invariante dans une boucle si chaque itération calcule la même valeur. Il est avantageux de déplacer l'évaluation de cette valeur unique avant la boucle. On reconnaît une expression invariante à ce que les sources de ses variables sont toutes extérieures au corps de la boucle.

3.4 Optimisations liées à l'architecture de la cible

Les gains de performance des processeurs modernes sont dus pour partie aux progrès de la technologie, et pour une autre partie aux progrès de l'architecture. De ce coté, on peut identifier deux sources d'accélération : l'exploitation du parallélisme caché dans les programmes séquentiels, et l'exploitation de la localité dans les accès à la mémoire. Dans les deux cas, les architectes ont prévu de cacher ces mécanismes, et l'utilisateur peut programmer comme s'il disposait d'un ordinateur séquentiel muni d'une mémoire à temps d'accès uniforme. Cependant, on s'est aperçu depuis longtemps que ces dispositifs fonctionnent

mieux si on les prend en compte au moment de la programmation. Par exemple, le détecteur de parallélisme caché fonctionne mieux si on regroupe ensemble des opérations indépendantes. Ce type d'optimisation est très complexe et non portable. Il est donc en général caché dans le compilateur.

Comme il n'est pas possible de présenter ici toutes les optimisations liées à l'architecture, nous nous bornerons à présenter une méthode, le pipeline logiciel. Le problème de l'amélioration de la localité sera traité en section 4. Les processeurs modernes disposent d'unités de calcul indépendantes, souvent spécialisées. Soit un processeur équipé d'un additionneur et d'un multiplieur. Pour simplifier on suppose que chaque opération prend un cycle, et on ignore les accès à la mémoire. Soit l'instruction :

```
y = a*x + b ;
```

Il n'y a pas de parallélisme, car l'additionneur doit attendre la fin de la multiplication pour opérer. Soit maintenant la boucle :

```
for(i=0 ; i<n ; i++)
 y[i] = a*x[i]+b ;
```

On peut effectuer en parallèle, à l'itération i, le produit `a*x(i)` et l'addition de l'itération $i-1$. On peut représenter cette optimisation comme une transformation de la boucle, qui devient :

```
r[0] = a*x[0] ;
for(i=1 ; i<n-1 ; i++){
  r[i%2] = a*x[i] ;
  y[i-1] = r[(i-1)%2] + b ;
}
y[n-1]=r[n-1]+b ;
```

Naturellement, r n'est pas un tableau, mais une paire de registres utilisés de façon cyclique. La boucle ainsi transformée devient un pipeline logiciel. Les deux instructions qui précèdent et suivent la nouvelle boucle en sont le prélude et le postlude. Si n est grand, la performance de l'ordinateur est doublée par rapport à une compilation naïve. Existe-t-il des méthodes générales pour la construction de tels pipelines logiciels ?

Il est assez facile d'imaginer des méthodes d'ordonnancement pour du code sans boucle. On peut par exemple utiliser une table de réservation : une matrice dont les lignes représentent les unités de calcul et dont les colonnes représentent le temps.On marque une case si l'unité correspondant à la ligne est occupée pendant le cycle correspondant à la colonne. On cherche à placer chaque opération le plus tôt possible (le plus à gauche) en respectant le flot des données et les réservations

déjà faites. Cette méthode ne donne pas nécessairement le code optimal, mais elle est simple et on peut borner la perte de performance qu'elle entraîne.

La méthode de l'ordonnancement modulo (*modulo scheduling*) fonctionne de façon analogue, a ceci près que l'on commence par choisir la période T du pipeline logiciel. Si l'on utilise une ressource à l'instant t, elle sera également utilisée aux instants $t + T$, $t + 2T$, etc. et on marque en conséquence la table de réservation. Il se peut alors qu'il soit impossible de trouver un ordonnancement valide. Dans ce cas, on recommence pour une valeur supérieure de T, et ceci jusqu'à succès. Il est intéressant de voir apparaître sur cet exemple un cas d'interférence entre optimisations. Les processeurs modernes ont tous des registres, qu'il est important de bien utiliser pour éviter des accès coûteux à la mémoire. La boucle originale utilise un registre pour stocker le produit `a*x[i]`, alors que le pipeline logiciel en utilise deux. Il est fréquent qu'une version optimisée d'une boucle ait besoin de plus de registres que la version initiale, ce qui peut imposer des écritures de résultats intermédiaires en mémoire, lesquelles peuvent faire perdre les gains de performance dus au pipeline logiciel. Il s'agit là d'une situation très courante en optimisation, et ce genre d'interaction est impossible à prendre en compte par le matériel, et très difficile à traiter pour le programmeur. Des études récentes cherchent à coupler le pipeline logiciel et l'allocation des registres à l'aide de la programmation linéaire en nombres entiers.

4 La parallélisation automatique et le modèle polyédrique

La parallélisation automatique est un cas particulier de l'optimisation où la cible est une machine parallèle ou vectorielle. Dans le cas des programmes de calcul numérique intensif, qui sont basés sur des algorithmes d'algèbre linéaire hautement répétitifs (boucles DO), agissant sur des tableaux et utilisant des mécanismes d'indexation assez simples (on parle de programmes *réguliers*), il a été possible de développer une théorie complète : le modèle polyédrique.

4.1 L'approche géométrique

Dans l'espace à n dimensions, un polyèdre est un ensemble de points satisfaisant à un système d'inégalités linéaires. Par exemple, dans l'espace à deux dimensions, l'ensemble $D = \{x, y \mid 0 \leq x < n, 0 \leq y < x\}$ est un polyèdre (en fait, un triangle rectangle isocèle). Soit maintenant

la boucle :

```
for(i=0 ; i<n ; i++)
 for(j=0 ; j<i ; j++)
S : y[i]=y[i]+a[i][j]*x[j];
```

L'instruction S va être exécutée i fois pour chaque valeur de i, soit $n(n+1)/2$ fois au total. Il faut nommer chacune des exécutions ou opérations : il suffit pour cela d'associer à chacune d'elles les valeurs courantes de i et de j. Nous parlerons de l'opération $\langle i, j \rangle$. De l'analyse des bornes des boucles on déduit $0 \leq i < n$ et $0 \leq j < i$, ce qui signifie que le vecteur $\langle i, j \rangle$ appartient au polyèdre D ci-dessus. Mais les compte-tours d'une boucle sont des entiers. Le domaine d'itération de S est donc l'ensemble des points entiers contenus dans D, objet mathématique que l'on appelle un Z-polyèdre, et dont la dimension n'est autre que le nombre de boucles entourant S. Cette façon de représenter un programme a de multiples avantages. La théorie des polyèdres et des Z-polyèdres a été bien développée pour les besoins de la Recherche Opérationnelle. De nombreuses opérations (par exemple l'intersection de deux polyèdres) sont triviales, et deux algorithmes bien connus, le Simplex et l'algorithme de Fourier, permettent de décider si un polyèdre est vide ou non. La succession temporelle des opérations est donnée par l'ordre lexicographique des vecteurs d'itération. Vu les contraintes que nous imposons aux fonctions d'indice, les opérations d'indexation conduisent à calculer des images de polyèdres par des fonctions affines, et les dépendances peuvent être décrites exactement par des Z-polyèdres, c'est-à-dire comme des systèmes de contraintes linéaires en nombres entiers. Il y a ensuite plusieurs façons d'exhiber le parallélisme caché du programme. La plus parlante consiste à dater chaque opération : deux opérations exécutées à la même date sont exécutées en parallèle. L'ordonnancement doit satisfaire une contrainte de causalité : une opération ne peut commencer que si tous ses opérandes sont disponibles, ou encore si toutes ses sources sont terminées. On obtient de cette façon un système de contraintes qui lui aussi se résout par la programmation linéaire.

Pour paralléliser l'exemple ci-dessus, il faut d'abord rechercher les sources de l'instruction S. Les tableaux `A` et `x` ne sont pas modifiés ; on peut supposer qu'ils existent au moment où la boucle commence : ils n'imposent pas de contrainte à l'ordonnancement. Par contre, la source de `y[i]` à l'itération $\langle i, j \rangle$ est clairement l'itération $\langle i, j-1 \rangle$. L'ordonnancement θ doit donc vérifier :

$$\theta(i, j) > \theta(i, j-1)$$

et il est facile de voir que $\theta(i, j) = j$ est une solution.

La génération d'un programme parallèle à partir d'un ordonnancement est un problème d'énumération des opérations dans l'ordre des temps croissants. Il a reçu beaucoup d'attention et de nombreuses solutions ont été proposées, mais son étude est assez technique et nous entraînerai trop loin ici. Dans le cas de l'exemple, il faut trouver quelles sont les opérations à effectuer à l'instant t. Vu l'ordonnancement choisi, elles satisfont $0 \leq i < n, j = t, 0 \leq j < i$, ce qui se simplifie en $t < i < n$. Ceci conduit au programme :

```
for(t=0 ; t<n ; t++)
 forall(i=t+1 ; t<n ; t++)
   y[i]=y[i]+a[i][t]*x[t];
```

où on utilise le mot clef `forall` pour signaler une boucle parallèle.

Ces quelques indications constituent les linéaments d'une méthode de compilation pour ordinateurs parallèles. Pour en faire une méthode pratique, d'autres problèmes plus techniques doivent être résolus. Par exemple, la méthode dégage en général trop de parallélisme pour les architectures courantes. Le problème de l'ordonnancement sous contrainte de ressources (aussi bien le nombre de processeurs que la taille de la mémoire) n'a pas pour le moment de solution satisfaisante, et on doit se contenter de méthodes de pavage plus ou moins *ad hoc*.

4.2 L'amélioration de la localité

On dit qu'un programme a de la localité lorsque, en un point donné de son déroulement, il a tendance à n'utiliser qu'une faible fraction de son espace mémoire. Cette fraction constitue l'espace de travail (*working set*) du programme ; naturellement, elle évolue au fur et à mesure de l'avancement des traitements. Un programme a d'autant plus de localité que ses espaces de travail sont plus petits. Il est important qu'un programme ait beaucoup de localité, qu'il soit exécuté sur une architecture parallèle ou séquentielle. Par exemple, lorsque l'on a construit un code parallèle, et donc alloué un ensemble d'opérations à chaque processeur, il faut rapprocher les données du processeur qui va les utiliser. Dans le modèle polyédrique, on y parvient de la façon suivante.

Il s'agit d'attribuer un numéro de processeur à chaque cellule de tableau et à chaque opération. Pour cela, on postule que ce numéro est une fonction affine des indices du tableau ou du vecteur d'itération de l'opération. Pour qu'il n'y ait pas de communications, il faut que chaque opération soit exécutée par le processeur dont la mémoire héberge ses opérandes et son résultat. En écrivant cette condition de placement pour chaque instruction, on obtient un système d'équations

linéaires et homogènes dont les inconnues sont les coefficients des fonctions de placement. Ce système a toujours la solution triviale, qui correspond au collapsus du calcul sur le processeur 0. Il n'en a en général pas d'autres. Pour trouver un placement intéressant, il faut ignorer certaines conditions de placement, ce qu'il faut compenser en prévoyant des communications résiduelles. De nombreuses heuristiques ont été développées pour bien choisir ces communications : par exemples celles de plus petit volume ou celles que l'architecture cible réalise efficacement.

La localité est tout aussi importante pour les machines séquentielles à cause de l'utilisation des caches pour mieux tolérer la latence de la mémoire. Il est tentant de charger le compilateur d'optimiser la localité du programme. En gros le principe de fonctionnement d'un cache est le suivant. Lors de la première utilisation d'une donnée, elle est lue en mémoire, mais également copiée dans le cache, petite mémoire rapide située à proximité du processeur. Lors des utilisations suivantes, la donnée est dans le cache, où elle sera accessible beaucoup plus rapidement. Au cours de l'exécution du programme, le cache se rempli progressivement ; après un certain temps, il faut y faire de la place en éjectant les données les plus anciennes. En gros, on a d'autant moins de chance de trouver une donnée dans le cache qu'il s'écoule plus de temps entre deux accès succesifs. Il faut rapprocher les accès successifs à une même donnée. Cette opération modifie l'ordre des opérations ; ceci n'est pas toujours légal, est c'est une analyse de dépendance qui permet de choisir les optimisations permises.

5 Conclusion : nouvelles machines, nouveaux langages

En résumé, le rôle d'un compilateur est d'assurer la transition entre les langages de haut niveau, souvent de conception ancienne, et l'architecture des processeurs modernes. Du coté des langages, la tendance est à la concision maximale, le compilateur étant chargé de fournir les détails manquants par un processus qui se rapproche de plus en plus de la programmation automatique. De l'autre coté, les architectures deviennent de plus en plus complexes, et des performances satisfaisantes ne peuvent être obtenues que par un processus d'optimisation presque hors de portée du programmeur, tant les facteurs à prendre en compte sont nombreux. La compilation devient une activité si complexe qu'il est difficile d'en avoir la maîtrise complète. On se rabat sur des enchaînements de compilateurs ou de phases de compilateur, qui sont de plus en plus difficiles à agencer, et qui ne fournissent qu'une solution sous-

optimale au problème général de l'écriture d'un programme efficace. Il manque ici une théorie de "grande unification", qui devrait prendre en compte à la fois l'efficacité du code et les contraintes de ressources (nombre de registres, d'unités fonctionnelles, taille des caches et de la mémoire). Un autre problème, qui a été beaucoup discuté mais qui n'est pas entièrement résolu est celui du compilateur paramétrable, capable de s'adapter le plus vite possible à l'évolution des processeurs.

Pour boucler la boucle, il faut se demander si la réalisation d'un compilateur est devenue une activité répétitive, suceptible elle aussi d'être automatisée. J'espère avoir convaincu le lecteur que l'on en est encore loin, sauf en ce qui concerne l'analyse syntaxique. La recherche en compilation a encore de beaux jours devant elle.

6 Références

Plutôt que de donner des références sous la forme usuelle, ce qui aurait pris trop de place, j'ai préféré indiquer quelques points d'entrée qui permettront au lecteur de mener sa propre recherche bibliographique. Le traité classique est :

[AHO 86] A.J. Aho, R. Sethi, J.D. Ullman : Compilers, Principle, Techniques and Tools, Addison-Wesley, 1986.

On peut également signaler :

[MUCH 97] S. S. Muchnick : Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.

[DARTE 96] G-R. Perrin et A. Darte (Eds.), The Data Parallel Programming Model , Springer, LNCS 1132, 1996.

Pour tout ce qui concerne les polyèdres et la programmation linéaire, on consultera

[SCHR 86] A. Schrijver, Theory of Linear and Integer Programming, Wiley, 1986.

On trouvera de nombreux articles sur la compilation dans la revue "Transaction on Programming Languages and Systems" de l'ACM. Les articles relatifs à la parallélisation automatiques se trouvent dans des revues plus spécialisées, comme "Int. J. of Parallel Programming" (Plenum), "J. of Parallel and Distributed Computing" (Academic Press) et "Parallel Computing" (Elsevier).

Les principales conférences du domaine sont "Principles of Programming Languages" (ACM), "Programming Languages Design and Implementation" (ACM) et, pour la partie parallèle, "Principles and Practice of Parallel Programming" (ACM) et "Parallel Architectures and Compiler Techniques" (IEEE).