Résumé

Il y a essentiellement deux méthodes de programmation parallèle; l'une consiste à spécifier explicitement tout le parallélisme, l'autre à le laisser découvrir et exploiter par un compilateur. Cette dernière méthode s'applique aussi bien à des langages sans séquencement : fonctionnels, logiques, etc. qu'aux langages séquentiels classiques tel Fortran. Nous présentons les deux opérations que doit effectuer un compilateur paralléliseur : la détection du parallélisme par des méthodes d'analyse de dépendances, et la génération du code parallèle. Pour cette dernière étape, nous présentons successivement un algorithme classique, l'algorithme de Allen et Kennedy, et des développements plus récents, qui s'inspirent des recherches sur la synthèse des réseaux systoliques.

1 Introduction

C'est maintenant un lieu commun que de constater que les processeurs modernes font et vont faire de plus en plus appel au parallélisme pour améliorer leurs performances. Ceci est vrai dans toute la gamme des puissances, depuis les processeurs sur un chip jusqu'aux super-ordinateurs. Mais pour les micro-processeurs, le besoin de parallélisme est limité, et on peut en trouver suffisamment à l'intérieur d'une instruction ou en analysant des séquences de quelques dizaines d'instructions. Le parallélisme peut alors être caché au programmeur. Cela ne suffit plus pour les super-ordinateurs, surtout dans leur variante massivement parallèle; on doit exploiter le parallélisme induit par l'application d'un même traitement à un grand nombre de données différentes. En apparence, ceci nécessite la collaboration de l'utilisateur, parce que les interactions entre traitements ne sont pas faciles à mettre en évidence, alors que le programmeur en a une connaissance au moins intuitive.

La programmation d'une machine parallèle peut ainsi se faire à bas niveau, par exemple en ajoutant à un langage de facture classique des instructions représentant

les opérations caractéristiques de la programmation parallèle – création de tâches, synchronisations, communications. L'exemple le plus connu de langage parallèle est Occam. Cette approche a le grand mérite de la transparence : le programmeur contrôle dans le détail l'implémentation de son algorithme; des études de performance et de complexité peuvent être menées valablement. Il s'agit donc de la méthode de choix pour le développement de nouveaux algorithmes.

Par contre, en tant que méthode de production de logiciel, elle souffre de graves défauts, entre autre la difficulté de mise au point des programmes ainsi rédigés, due pour l'essentiel au caractère non-déterministe de beaucoup d'architectures parallèles. De nombreux chercheurs se sont donc préoccupés, depuis les travaux de pionnier de Kuck [8], de construire des outils d'aide à la programmation parallèle.

On peut imaginer qu'un compilateur, plus sophistiqué que les logiciels usuels, se charge de rechercher le parallélisme implicite du programme source et de l'adapter à la machine cible. On peut utiliser un langage source *sans séquencement* – langage fonctionnel, à flot de données, logique, etc. – ou un langage plus classique tel Fortran 77. C'est cette approche qui constitue la *parallélisation automatique* au sens le plus courant du terme. La parallélisation automatique se donne ainsi trois objectifs : récupérer les codes existants, faciliter le développement de nouvelles applications, assurer la portabilité des programmes entre architectures parallèles.

Si le premier objectif était au départ considéré comme primordial, il a perdu beaucoup de son importance. Les anciens codes de calcul sont en effet souvent écrits dans des dialectes périmés de Fortran; ils ne peuvent être réutilisés que moyennant une révision complète.

Le deuxième objectif est, lui, beaucoup plus actuel. Il est très tentant de développer un algorithme en version séquentielle, avec tous les moyens et environnements existants, et de ne passer à la version parallèle que lorsque l'essentiel de la mise au point a été fait.

L'importance du troisième objectif, enfin, croit avec l'apparition à cadence rapide de nouvelles architectures. Les ordinateurs séquentiels se ressemblent tous, et il est tout à fait concevable de mettre au point un algorithme sur un IBM-PC pour l'exécuter sur un IBM-3090. Au contraire, des expériences récentes ont montré qu'il est très difficile de faire migrer un programme, même entre des machines aussi voisines que deux ordinateurs vectoriels d'origine différentes. Que dire alors de la migration entre un ordinateur vectoriel et une Connection Machine?

Les techniques de parallélisation sont en pleine évolution. Mais cependant les méthodes utilisées ont un air de famille. Le travail se fait toujours en deux étapes principales. Une phase d'analyse permet de rassembler des informations globales sur la façon dont les diverses instructions interagissent entre elles. On parle usuellement d'*analyse sémantique*, pour montrer que cette phase s'intéresse aux traitements exécutés par le programme et non pas seulement à la façon dont il est écrit.

La deuxième étape consiste à exploiter les résultats obtenus dans la phase d'analyse pour guider la génération du programme parallèle. On peut la voir comme l'identification dans le programme source des formes caractéristiques des opérations parallèles usuelles, comme le lancement de deux tâches ou la boucle `DOALL`. Naturellement, la reconnaissance de ces formes est d'autant plus facile – i.e., il y a d'autant moins de modèles à considérer – que la phase d'analyse a fourni une représentation plus synthétique du programme original.

La suite de ce chapitre se répartit naturellement suivant le schéma que nous venons de présenter. La section 2 est consacrée aux diverses méthodes d'analyse, à leurs performances et à leurs limitations. La section 3 présente le plus puissant des algorithmes classiques de génération de code, ainsi que la technique des transformations de programme, qui en est le complément naturel. La section 4 présente une nouvelle méthode, fondée sur des techniques d'ordonnancement, qui permet une approche plus synthétique des transformations de programme. En conclusion, nous chercherons à situer les limites de l'approche que nous présentons, et à suggérer quelques directions pour leur dépassement.

2 Analyse sémantique

2.1 Ordres d'exécution

L'analyse d'un programme en vue de son exécution parallèle ne peut pas s'appuyer sur la notion d'instruction, parce que c'est entre les différentes répétitions d'une même instruction que nous espérons trouver le plus de parallélisme. On doit considérer chaque exécution d'une instruction comme une entité distincte, une *opération*. Un programme doit être vu comme une ensemble d'opérations donné a priori, au moins conceptuellement. Mais un programme ne se réduit pas à l'ensemble de ses opérations, pas plus que l'on ne peut jouer une sonate en frappant simultanément l'ensemble des notes de la partition. Ici, comme en musique, l'*ordre d'exécution* des opérations est primordial. On est donc conduit à représenter un programme comme un *ensemble ordonné d'opérations*.

Dans un programme séquentiel, les opérations sont exécutées une à une dans un ordre fixé à l'avance. L'ordre associé à un tel programme est donc total. Au contraire, dans un programme parallèle, il n'y a aucun ordre entre opérations de deux tâches différentes. L'ordre associé à un tel programme est partiel. Le degré zéro du parallélisme est l'ordre total, le parallélisme maximum est associé à l'ordre vide.

Il est important de pouvoir caractériser de façon compacte les opérations d'un programme séquentiel et leur ordre d'exécution. Ceci n'est facile que si le programme est structuré, i.e. construit à partir d'instructions élémentaires par les opérations de mise en séquence, de boucle et de test. Dans ces conditions, la seule façon de provoquer la répétition d'une instruction est de l'englober dans une ou plusieurs boucles. On suppose que chaque boucle est équipée d'un compte-tour. On peut alors repérer une opération en donnant le nom de l'instruction exécutée et la liste des valeurs des compte-tours englobants. Il est commode de traiter cette liste comme un vecteur, le *vecteur d'itération* [7].

Quand à l'ordre séquentiel d'exécution, il se déduit essentiellement de la sémantique classique des nids de boucles. Nous avons montré [6] que l'ordre séquentiel $\prec$ s'écrit :

$$\langle r, \mathbf{a} \rangle \prec \langle s, \mathbf{b} \rangle \equiv \mathbf{a}[1..N_{rs}] \ll \mathbf{b}[1..N_{rs}] \vee (\mathbf{a}[1..N_{rs}] = \mathbf{b}[1..N_{rs}] \wedge T_{rs}), \qquad (1)$$

où $\ll$ est l'ordre lexicographique, N_{rs} est le nombre de boucles englobant simultanément r et s, et T_{rs} un booléen vrai ssi r est écrite avant s dans le texte du programme.

2.2 Dépendances

Nous savons que le résultat final d'un programme dépend d'une part de l'ensemble des opérations exécutées, mais aussi de l'ordre de cette exécution. Pour un programme séquentiel, cet ordre est fixé, et il n'y a donc qu'un seul résultat possible. Pour un programme parallèle, au contraire, il y a plusieurs exécutions possibles, obtenues en étendant de toutes les façons l'ordre partiel donné en un ordre total. Si le programme parallèle est la parallélisation d'un programme séquentiel, tous ces exécutions doivent donner le même résultat que le programme séquentiel– le programme parallèle doit être *déterministe*.

Considérons un programme séquentiel composé de deux opérations u et v telles que $u \prec v$. La version parallèle ne peut être déterministe que si ces deux opérations commutent :

$$u; v = v; u.$$

On note $u \perp v$ la relation de dépendance, négation de la relation de commutation.

Ce résultat se généralise de la façon suivante : Pour que $\mathcal{P}_{//} = \langle E, \prec_{//} \rangle$ soit équivalent à un programme séquentiel $\mathcal{P} = \langle E, \prec \rangle$, il suffit que, si deux opérations u et v sont en dépendance, elles soient exécutées dans le même ordre dans le programme séquentiel et dans le programme parallèle :

$$u \perp v \Rightarrow u \prec v \equiv u \prec_{//} v.$$

Une solution évidente est de prendre[1] :$\prec_{//} = (\prec \cap \perp)^+$.

La relation $\prec \cap \perp$ est le *graphe de dépendance détaillé* (GDD) du programme original.

Il est important d'observer que pour toute relation D contenant $\perp$, l'ordre $\prec_D = (\prec \cap D)^+$ engendre également un programme parallèle déterministe équivalent au programme original. Cette remarque autorise un calcul approximatif de la relation de dépendance, pourvu que l'approximation soit toujours *pessimiste*.

Le GDD est trop complexe pour être exploitable. Les divers graphes de dépendance (GD) utilisés dans la littérature sont obtenus en quotientant le GDD, i.e. en fusionnant tous les sommets exécutions de la même instruction. Le résultat est un graphe orienté dont les sommets sont les instructions du programme original. Il y a un arc de s vers t ssi il existe dans le GDD deux opérations $\langle r, \mathbf{a} \rangle$ et $\langle s, \mathbf{b} \rangle$ telles que $\langle r, \mathbf{a} \rangle \prec \langle s, \mathbf{b} \rangle$ et $\langle r, \mathbf{a} \rangle \perp \langle s, \mathbf{b} \rangle$.

Pour tester la commutativité de deux opérations, on se contente en général des conditions suffisantes dues à Bernstein [4]. A toute opération u on associe deux ensembles de cellules de mémoire, $L(u)$ les cellules lues, et $M(u)$ les cellules modifiées. On démontre alors facilement que pour que u et v commutent, il suffit que :

$$M(u) \cap L(v) = L(u) \cap M(v) = M(u) \cap M(v) = \emptyset. \tag{2}$$

Si on suppose, par exemple, que $u \prec v$, la violation de la première condition constitue une dépendance producteur-consommateur (PC ou *flow dependence*). Les autres termes correspondent respectivement aux dépendances consommateur-producteur (PC ou *anti-dependence*) et producteur-producteur (PP ou *output dependence*).

[1] L'opérateur + indique la fermeture transitive *stricte*.

2.3 Tests de dépendance

Le calcul approximatif de la relation de dépendance entre deux opérations est très simple : il suffit de former pour chacune les ensembles L et M, ce qui se fait par simple lecture de l'instruction exécutée, et de calculer les intersections figurant dans la formule (2). Mais, comme nous l'avons dit plus haut, ce calcul doit être répété un trop grand nombre de fois pour être physiquement possible. Le calcul direct du GD, par contre, est envisageable, parce que dès que nous avons trouvé une dépendance entre deux instances d'instructions, il est inutile de poursuivre l'analyse.

Il y a un arc du GD de type producteur-consommateur entre r et s s'il existe deux vecteurs d'itération, $\mathbf{a}$ et $\mathbf{b}$ tels que :

$$\langle r, \mathbf{a}\rangle \prec \langle s, \mathbf{b}\rangle \wedge M(r, \mathbf{a}) \cap L(s, \mathbf{b}) \neq \emptyset.$$

Le traitement des autres types de dépendance est similaire.

Le cas où les ensembles L et M ont une variable scalaire en commun est trivial; le point intéressant est le traitement des tableaux. Dans ce cas, ces ensembles vont être fonction du vecteur d'itération, et la condition $M \cap L \neq \emptyset$ va se présenter comme un système d'équations aux inconnues $\mathbf{a}$ et $\mathbf{b}$.

Pour simplifier la résolution du problème, il est usuel de supposer qu'il n'y a pas d'erreur dans les calculs d'indices. En conséquence, deux accès ne peuvent concerner la même cellule de mémoire que s'ils sont relatif au même tableau et que si les indices de même rang sont égaux. On peut parcourir les ensembles M et L, repérer les tableaux communs, et écrire pour chaque couple un système d'équations aux indices. Le traitement de ces équations n'est facile que si les indices ne dépendent que des compte-tours des boucles englobantes. Supposons que nous cherchions à identifier des violations des conditions de Bernstein associées à des accès au tableau `A`. Nous faisons l'hypothèse que dans l'instruction r (resp. s) cet accès est de la forme $\mathtt{A}[\mathbf{f}(\mathbf{a})]$ (resp. $\mathtt{A}[\mathbf{g}(\mathbf{b})]$). Nous devons alors rechercher les solutions entières de :

$$\mathbf{f}(\mathbf{a}) = \mathbf{g}(\mathbf{b}), \tag{3}$$

sous la condition :

$$\langle r, \mathbf{a}\rangle \prec \langle s, \mathbf{b}\rangle, \tag{4}$$

augmentée des contraintes que l'on peut déduire de l'analyse des bornes des boucles.

La résolution des équations (3) n'est praticable que si elles sont linéaires. Pour traiter (4), on remplace l'ordre lexicographique par une disjonction de contraintes linéaires :

$$\langle r, \mathbf{a}\rangle \prec \langle s, \mathbf{b}\rangle \equiv \bigvee_{p=0}^{N_{rs}-1} (\mathbf{a}[1..p] = \mathbf{b}[1..p] \wedge \mathbf{a}[p+1] < \mathbf{b}[p+1]) \vee (\mathbf{a}[1..N_{rs}] = \mathbf{b}[1..pN_{rs}] \wedge T_{rs}). \tag{5}$$

Puisque la contrainte (4) est une disjonction, la recherche des dépendances se ramène à $N_{rs}+1$ résolutions de systèmes de contraintes linéaires. On étiquette chaque arc du GD par la valeur de p qui a permis de trouver une solution, que l'on appelle sa *profondeur*. Nous écrirons $\langle r, s, p\rangle \in GD$ pour indiquer que nous avons détecté une dépendance de r vers s à la profondeur p.

Le calcul des dépendances se ramène ainsi à un problème connu : le test de faisabilité d'un système de contraintes linéaires. Une première approche est d'utiliser un algorithme général, comme par exemple un code de programmation linéaire en entiers. On pourra se reporter à [14] pour tout ce qui concerne ces algorithmes et leur complexité. Bien qu'il faille se méfier des arguments asymptotiques quand les tailles en jeu sont faibles, il est clair que ces méthodes sont coûteuses. Comme le nombre de dépendances à tester croit comme le carré de la taille du programme, il est important d'optimiser cette partie du traitement. On y parvient en se contentant d'une condition nécessaire d'existence des solutions, ce qui est bien une attitude pessimiste.

Une première idée est d'ignorer la contrainte d'intégrité qui doit être imposée aux solutions. On peut alors employer l'algorithme du simplexe, ou la méthode d'élimination de Fourier-Motzkin, de programmation plus simple.

On peut faire encore plus rapide en utilisant les tests de Banerjee [3], qui s'appuient sur la structure particulière du système de contraintes à traiter. La résolution de chaque équation aux indices correspond à la recherche des zéros d'une forme linéaire. La condition nécessaire d'existence de ces zéros est que la forme change de signe dans le domaine de variation des inconnues.

3 Algorithmes classiques de parallélisation

3.1 Tests élémentaires de parallélisme

Le but d'un algorithme de parallélisation est de déduire un programme parallèle de l'analyse du GD. Il existe par exemple des algorithmes de parallélisation rudimentaires, qui travaillent boucle par boucle. Pour qu'une boucle séquentielle isolée puisse être remplacée par une boucle parallèle, il suffit qu'il n'existe dans le GD aucun arc de profondeur 0 entre deux instructions de la boucle. Plus généralement, étant donné un nid de boucles parfaitement imbriqué, la boucle de niveau k peut être réécrite comme une boucle parallèle à condition qu'il n'existe aucun arc de profondeur $k-1$ entre instructions du corps de boucle.

3.2 Transformations de programmes

De nombreux algorithmes de parallélisation peuvent être vus comme un enchaînement de transformations appliquées au programme source avec la garantie que l'effet n'en est pas modifié. Les algorithmes élémentaires que nous venons d'évoquer pourraient être vus comme des transformations série/parallèle. Nous allons présenter un échantillon de transformations qui, bien que purement séquentielles, ont pour but de faire apparaître du parallélisme caché.

Normalisation du programme source Il s'agit d'obtenir un programme respectant du mieux possible les conditions permettant un fonctionnement efficace de l'analyseur sémantique. On peut entre autre éliminer les `GOTO`, dérécursiver les variables inductives, identifier les boucles `DO` cachées, expanser certaines procédures, etc. Ces transformations ne sont pas propres à la parallélisation automatique; pour un exposé d'ensemble, on consultera [1].

Modification du prédicat de séquencement Une transformation de ce type fait passer d'un programme $\mathcal{P}_1 = \langle E, \prec_1 \rangle$ à un autre programme $\mathcal{P}_2 = \langle E, \prec_2 \rangle$. Deux questions se posent :

- Les deux programmes sont-ils équivalents? C'est la question de la validité de la transformation.
- Le programme $\mathcal{P}_2$ est il "meilleur" (plus facile à paralléliser, plus efficace, etc.) que le programme $\mathcal{P}_1$? C'est la question d'opportunité.

Un premier exemple de transformations est l'éclatement de boucle :

```
do i = 1,n                    do i = 1,n
   S1                            S1
   S2              ⟶         end do
end do                        do i = 1,n
                                 S2
                              end do
```

La condition de validité de cette transformation est que, dans le GD, il n'y ait pas d'arc de `S1` vers `S2`. Son intérêt est que l'instruction `S1`, par exemple, peut porter des dépendances, alors que `S2` n'en porte pas. Dans ce cas, après éclatement, la deuxième boucle est parallèle, alors que la boucle originale ne l'est pas.

L'inversion de boucle est une autre transformation du même genre, qui peut être schématisée ainsi :

```
do i = 1,n                    do j = 1,m
   do j = 1,m                    do i = 1,n
      S            ⟶               S
   end do                        end do
end do                        end do
```

Ici, les conditions de validité ne se déduisent pas directement du graphe de dépendance. En gros, les systèmes de contraintes obtenus en associant un système d'équations aux indices et les conditions :

$$i < i',\ j > j',$$

ne doivent pas avoir de solution. L'intérêt de cette transformation, quand l'une des deux boucles se révèle être parallèle, est de la placer dans la position qui convient le mieux à l'architecture cible (par exemple à l'intérieur si la cible est vectorielle ou SIMD).

Modification du grain du calcul Dans tout ce qui précède, nous avons pris pour argent comptant la notion d'instruction. Nous avons en gros supposé que le travail de parallélisation s'effectuait au niveau des instructions du langage source. Or, sauf dans les cas les plus simples, celles-ci ne sont pas des blocs monolithiques. Une instruction complexe peut être éclatée :

```
x = f(g(y))    ⟶    temp = g(y)
                    x = f(temp)
```

Cette écriture suppose que `temp` est une variable qui n'est pas utilisée au point où la transformation est effectuée. La transformation inverse, d'ailleurs plus délicate à effectuer, est la *substitution avant*.

L'intérêt de cette transformation est de répartir les dépendances de l'instruction originale entre deux ou plusieurs instructions plus simples, ce qui augmente la probabilité de trouver du parallélisme.

Réorganisation de la mémoire Dans un programme, il est très fréquent de réutiliser une même cellule pour deux valeurs différentes si elles ne sont jamais utiles en même temps. Or, cette optimisation engendre aussitôt une dépendance PP, et presque toujours, par voie de conséquence, une ou plusieurs dépendances CP.

Si les deux définitions en conflit appartiennent à des boucles différentes, il suffit d'un simple renommage pour supprimer la dépendance parasite. Si par contre on doit traiter une réutilisation dans une boucle, aucun renommage ne peut supprimer la dépendance. Il faut remplacer le scalaire par un tableau indexé par le compte-tour de la boucle englobante. On dit que l'on a *expansé le scalaire*. La *reconstitution du flot des données* suit une règle simple :

```
do i = 1,n                          do i = 1,n
   y    = ... x ...                    y    = ... x(i-1) ...
   x    = e            ⟶              x(i) = e
   z    = ... x ...                    z    = ... x(i) ...
end do                              end do
```

Il est possible en généralisant, de mettre le programme source sous forme à assignation unique [6].

Transformations sémantiques Toutes les transformations que nous avons présentées jusqu'ici ne prennent pas en compte l'interprétation des opérations effectuées. Par exemple, les opérations de réduction :

$$s = \sum_{i=1}^{n} x_i.$$

sont usuellement écrites comme une boucle en Fortran et ne dégagent aucun parallélisme. Si on tient compte de l'associativité de l'addition, on peut écrire :

$$s = s_1 + s_2,\ s_1 = \sum_{i=1}^{n/2} x_i,\ s_2 = \sum_{i=n/2+1}^{n} x_i,$$

ce qui peut occuper deux processeurs. Si n est suffisamment grand, on peut poursuivre la subdivision jusqu'à dégager autant de parallélisme que nécessaire. Le problème de détecter et d'exploiter ces situations est un domaine de recherche actuellement actif.

Critique Pour toutes ces transformations, il n'existe pas de critère d'opportunité bien clair. Il est fréquent que le bon programme parallèle soit obtenu par une longue

suite de transformations, les programmes intermédiaires étant souvent moins efficaces que le programme de départ. Cette partie du travail de compilation se présente donc comme une recherche dans un arbre de programmes équivalents. L'espace de recherche est de taille en général énorme. Lorsque l'architecture cible est connue, il est possible d'utiliser des heuristiques pour guider la recherche, mais cette approche n'est pas satisfaisante en général. Nous allons voir qu'il est possible, dans certains cas, de trouver un fil d'Ariane dans le labyrinthe des transformations.

3.3 Algorithme de Allen et Kennedy

L'algorithme défini par Allen et Kennedy dans [2] s'applique non pas à un nid de boucle, mais à un programme complet sans tests. Il applique systématiquement l'éclatement de boucle et la détection des boucles parallèles en s'appuyant sur la notion de *composantes fortement connexes* (cfc) .

Dans un graphe orienté, deux sommets r et s appartiennent à un même cycle s'il existe un chemin de r vers s et un chemin de s vers r. La relation "appartenir à un même cycle" est à l'évidence une relation d'équivalence; les classes d'équivalence de cette relation sont les cfc du graphe et l'ensemble quotient en est le *graphe réduit* (GR), qui est acyclique.

Il y a une relation étroite entre boucles et cfc. En fait, si G_p est le GD réduit aux sommets de G et aux arcs de profondeur au moins égale à p et si H est une cfc de G_p alors il existe $p+1$ boucles emboîtées qui contiennent toutes les instructions de H.

Si les cfc sont associées aux boucles, le graphe réduit donne l'ordre à imposer aux boucles pour obtenir un fonctionnement déterministe. Cet ordre, en général partiel, peut être exploité ou non en fonction des caractéristiques de la cible. Si on souhaite ne pas perdre de parallélisme, on utilisera la notation suivante : soit Γ un ensemble ordonné à n éléments, et soit $B_1, \ldots, B_n$ n blocs d'instructions. $\Gamma(B_1, \ldots, B_n)$ sera le programme dans lequel les blocs $B_1, \ldots, B_n$ sont exécutés suivant l'ordre Γ.

La génération du code parallèle se fait par appel de l'algorithme récursif `codegen`. Cet algorithme a deux arguments : le premier est un ensemble d'instructions, et le second un entier dénotant une profondeur. L'appel initial s'applique à l'ensemble du programme à la profondeur 0.

`codegen`(G, p)
  Soit $H_1, \ldots, H_n$ les cfc du graphe G_p, et Γ le graphe réduit;
  pour $k = 1, n$
    si H_k est réduit à un seul élément r et si $N_{rr} = p$
    alors $B_k = r$,
    sinon il existe une boucle `do i = l,u` de niveau $p+1$ englobant H_k;
      soit $S =$ `codegen`$(H_k, p+1)$;
      s'il existe dans H_k un arc de profondeur p
      alors $B_k =$ `do i = l,u` S `end do`,
      sinon $B_k =$ `doall i = l,u` S `end do`;
  retourner $\Gamma(B_1, \ldots, B_n)$.

Suivant une remarque de la section 2.2, pour justifier cet algorithme, nous devons démontrer que tout couple d'opérations $\langle r, \mathbf{a} \rangle$ et $\langle s, \mathbf{b} \rangle$ en dépendance est cor-

rectement séquencé dans le programme résultant. Or le couple $\langle r, \mathbf{a}\rangle$, $\langle s, \mathbf{b}\rangle$ donne naissance à un arc du GD, $\langle r, s, p\rangle$ pour l'unique entier p tel que $\mathbf{a}[1..p] = \mathbf{b}[1..p]$. Nous allons en fait démontrer plus qu'il n'est nécessaire : si $\langle r, s, p\rangle \in GD$, alors tout couple $\langle r, \mathbf{a}\rangle$, $\langle s, \mathbf{b}\rangle$ associé est séquencé de la même façon que dans le programme original, même s'il n'est pas en dépendance. L'algorithme `codegen` peut être vu comme un algorithme d'élimination successive des arcs du GD. Il y a deux façons pour un arc de profondeur p d'être éliminé. Tout d'abord, lors d'un certain appel portant sur les sommets de G à la profondeur $q < p$, l'arc peut se retrouver "à cheval" sur deux cfc différentes. L'autre possibilité est qu'il appartienne à une cfc à la profondeur p; il sera alors éliminé à l'étape suivant à la profondeur $p + 1$. Au passage, puisque la profondeur des arcs du GD est bornée par le degré maximum d'imbrication des boucles, nous venons de démontrer la convergence de l'algorithme. Dans le premier cas, le séquencement désiré se déduit de l'expression du GR; dans l'autre, il découle du fait que la boucle englobante est séquentielle.

L'algorithme de Allen et Kennedy est susceptible d'une implémentation efficace, parce que les cfc et le graphe réduit peuvent être déterminés en temps linéaire par rapport au nombre d'arcs du GD grâce à un algorithme dû à Tarjan.

4 Nouvelles méthodes de parallélisation

L'étude de l'algorithme de Allen et Kennedy nous montre ce qu'il est possible d'obtenir en matière de pilotage des transformations. Cet algorithme combine la détection des boucles parallèles, la détection de blocs parallèles et l'éclatement de boucles en un algorithme efficace qui n'exige aucune exploration d'arbre. Il reste encore beaucoup de transformations à prendre en compte; un premier pas dans cette direction a été fait grâce à l'utilisation de bases de temps, [5, 11, 10, 13], une technique inspirée des recherches sur la synthèse des réseaux systoliques.

4.1 Notion de base de temps

Une *base de temps* (en anglais *schedule*) peut être vue comme une nouvelle façon de représenter les ordres d'exécution. Etant donné un programme $\langle E, \prec\rangle$, soit θ une fonction quelconque de l'ensemble E vers les entiers positifs. On peut lui associer l'ordre strict $\prec_\theta$:

$$u \prec_\theta v \equiv \theta(u) < \theta(v).$$

Cet ordre n'est pas nécessairement total : deux opérations sont incomparables si leurs bases de temps sont égales.

La condition pour que le programme $\langle E, \prec_\theta\rangle$ soit déterministe et équivalent à $\langle E, \prec\rangle$ est :

$$\forall u, v \in E : u \perp v \wedge u \prec v \Rightarrow \theta(u) < \theta(v). \tag{6}$$

Cette contrainte est appelée la *condition de causalité*. Une caractéristique importante d'une base de temps θ est sa *latence* :

$$L = \max \theta(E).$$

Cette information est à comparer au nombre total d'opérations du programme :

$$S = \mathrm{Card}(E),$$

qui est une estimation de la durée d'exécution séquentielle.

4.2 Bases de temps et transformations de programmes

Alors que la programmation parallèle usuelle est basée sur le concept de processus – ensemble maximal d'opérations totalement ordonnées – l'exploitation d'une base de temps se fait au travers de la notion de front :

$$\mathcal{F}(t) = \theta^{-1}(t) = \{u \in E \mid \theta(u) = t\}. \tag{7}$$

Le programme parallèle associé à θ est schématisé ci-dessous :

```
do t = 0, L
   exécuter en parallèle les opérations de F(t)
   synchroniser
end do
```

Si ce programme est exécuté sur un ordinateur disposant d'un nombre illimité de processeurs, et si chaque opération prend un temps unité, alors il est clair que les opérations du front $\mathcal{F}(t)$ débutent à l'instant t et se terminent à l'instant $t+1$. Si le nombre de processeurs est limité, les opérations d'un front devront être réparties aussi équitablement que possibles entre les processeurs. Il y aura alors deux sources d'inefficacité, un éventuel déséquilibre de charge et le temps pris par l'opération de synchronisation. Sous des hypothèses très larges, on montre [5] que la perte de rendement est négligeable si le taux moyen de parallélisme, S/L est grand devant le nombre de processeurs disponibles. Or, beaucoup d'algorithmes numériques ont des taux moyens de parallélisme qui croissent avec la taille de l'objet traité. Le programme parallèle ci-dessus est donc *asymptotiquement efficace*.

Le système de fronts ne peut être effectivement écrit que si la base de temps est connue explicitement et simplement, en pratique comme une forme affine par rapport aux composantes du vecteur d'itération. Si cette condition est remplie, il est possible d'écrire les fronts comme des boucles, en utilisant une technique de projection dérivée de la méthode de Fourier-Motzkin.

Considérons le programme ci-dessous (produit de deux matrices) :

```
      do i = 1,n
         do j = 1,n
1           c(i,j) = 0.
            do k = 1,n
2              c(i,j) = c(i,j) + a(i,k)*b(k,j)
            end do
         end do
      end do
```

On trouve facilement que les bases de temps sont :

$$\begin{aligned} \theta(1,i,j) &= 0, \\ \theta(2,i,j,k) &= k. \end{aligned}$$

La latence est n pour n^3 opérations. En construisant le système des fronts on se rend compte que l'itération 0 de la boucle sur le temps est particulière. Il est tentant de l'individualiser – c'est la transformation *loop peeling* :

```
doall i = 1,n
   doall j = 1,n
      c(i,j) = 0.
   end doall
end doall
do t = 1,n
   doall i = 1,n
     doall j = 1,n
       c(i,j) = c(i,j) + a(i,t)*b(t,j)
     end doall
   end doall
end do
```

On a procédé à un éclatement de boucle, une inversion, et à la détection de 4 boucles parallèles en une seule opération!

On peut montrer [11] que toutes les transformations classiques agissant sur le prédicat de séquencement correspondent à l'application de ce schéma à une forme spécifique de base de temps. Si par exemple la base de temps du corps d'un nid de boucles ne dépend que de l'un des compte-tours, la boucle correspondante est séquentielle, les autres étant parallèles, et la boucle séquentielle sera "extériorisée". Une base de temps qui combine linéairement plusieurs compte-tours correspond à une application de la méthode des vagues de Lamport [9].

On pourrait penser que pour prendre en compte les transformations "expansion de scalaires ou de tableaux" il suffise d'ignorer les dépendances CP et PP, mais ce n'est pas suffisant. L'expansion fait non seulement disparaître ces deux types de dépendance, mais aussi certaines dépendances PC qui ne correspondent à aucun transfert réel d'information.

Dans le code :

```
1     x = a;
      ...
2     x = b;
      ...
3     y = x
```

la dépendance PC $1 \rightarrow 3$ ne correspond à rien de réel et disparaît si on procède à un renommage.

Conserver ces dépendances parasites contraint inutilement la base de temps et conduit en général à des résultats sous-optimaux. La meilleure solution est de procéder à une expansion totale en utilisant les techniques de [6] avant de calculer la base de temps.

4.3 Détermination des BdT

On voit que tout repose sur la détermination des bases de temps, c'est-à-dire sur une méthode efficace de résolution de (6), que nous commençons par mettre sous la forme :

$$\forall u, v \in E : u \perp v \wedge u \prec v \Rightarrow \theta(u) + 1 \leq \theta(v). \tag{8}$$

Pour résoudre cet ensemble d'inégalités, on postule une forme simple pour θ :

$$\theta(r, \mathbf{a}) = \mathbf{h}_s.\mathbf{a} + k_s,$$

où $\mathbf{h}_s$ est un vecteur et où k_s est une constante, tous deux inconnus. Chaque instance de (8), si on y substitue la forme ci-dessus de θ, donne une contrainte linéaire. Si le nombre d'instances est fini, il est possible de résoudre le système ainsi obtenu par l'algorithme du simplexe. Le problème est que le nombre d'instances peut être très grand, voire même infini si le programme est paramétré. Les deux principales méthodes de résolution s'appuient sur des techniques permettant de résumer l'ensemble des instances de (8) en un système fini.

Cette réduction est possible parce que l'on doit écrire (8) en tout couple de points en dépendance dans le domaine d'itération, qui est en général un polyèdre. Il existe deux façons de spécifier un polyèdre : l'une comme solution d'un système d'inégalités linéaires, et l'autre comme enveloppe convexe d'un système de points. Dans ce dernier cas, il est facile de voir que pour qu'une inégalité linéaire soit vérifiée dans le polyèdre, il faut et il suffit qu'elle soit vérifiée aux points générateurs. La méthode des sommets [12] consiste donc à déterminer un ensemble générateur des domaines d'itération, à y écrire (8) et à résoudre le système de contraintes linéaires ainsi obtenu.

L'autre méthode utilise directement le système d'inégalités définissant les domaines d'itérations, et s'appuie sur un résultat classique de la théorie des inégalités linéaires, le lemme de Farkas sous forme affine [14]. Pour qu'une inégalité linéaire soit conséquence d'une système d'inégalités linéaires, il faut et il suffit qu'elle en soit combinaison affine positive. Le délai $\theta(v) - \theta(u) - 1$ peut donc être mis sous la forme d'une combinaison affine à coefficients positifs inconnus des inégalités définissant les domaines d'itération. Le résultat est une identité par rapport aux coordonnées des vecteurs d'itération. On peut identifier terme à terme et résoudre le système obtenu.

5 Conclusion

Au terme de cette revue des techniques de parallélisation, il est bon de faire le point sur ce qu'il est possible de faire et sur les obstacles qui inhibent l'action d'un paralléliseur.

Les paralléliseurs classiques sont basés sur le modèle de l'enchaînement de transformations. Cette approche est efficace aussi longtemps qu'on ne l'applique qu'à de petites régions du programme source, par exemple en travaillant boucle par boucle ou nid de boucles par nid de boucles. Par des techniques combinant l'approche pessimiste et l'isolement des parties de programme qui résistent à l'analyse, il est possible de réaliser des paralléliseurs robustes, capables de traiter des programmes à peu

près quelconques, naturellement avec des résultats qui peuvent se révéler décevants si le programme source est trop loin de l'idéal. La contrepartie est que l'on ne trouve par cette approche que des quantités limitées de parallélisme. Elle convient donc bien aux architectures vectorielles ou aux architectures MIMD avec un petit nombre de processeurs.

Les nouvelles méthodes, au contraire, traitent un programme dans son ensemble. Elles trouvent donc en général beaucoup de parallélisme, et sont l'outil idéal pour la programmation des architectures à parallélisme massif. Par contre, elles sont beaucoup plus coûteuses que les méthodes classiques. Le programme source doit respecter des normes très sévères : structuration, indices et bornes de boucles linéaires, pas de boucles `WHILE`, ni fonctions ni sous-programmes. Un premier plan d'attaque est d'essayer de relaxer certaines de ces contraintes : le travail est bien avancé en ce qui concerne le traitement des procédures et débute pour les boucles `WHILE`. L'autre approche est d'essayer de *durcir* un paralléliseur pour qu'il continue à fonctionner en mode dégradé en présence d'obstacles. Ce travail est à peine ébauché.

La programmation des architectures à parallélisme massif pose un autre problème, celui de la distribution des données. Dès que le nombre de processeurs dépasse quelques dizaines, il n'est plus possible, pour des raisons technologiques, de réaliser un espace mémoire uniforme. Chaque processeur possède donc sa mémoire propre et communique avec les autres processeurs par un réseau. Les dépendances PC correspondent à des échanges d'information qui devront passer par le réseau sauf si les données ont pu être implantées astucieusement. Le problème de la distribution peut être posé de façon analogue à celui de l'ordonnancement – on recherche une *base d'espace* – mais comme il est très lié à l'architecture de la cible, son étude est pour le moment beaucoup moins avancée.

Pour terminer il est frappant d'observer l'émergence d'un schéma général de parallélisation, composé d'une phase d'analyse et d'une phase de synthèse. L'analyse conduit à une représentation de plus en plus abstraite du programme source, et son résultat se met très bien sous la forme d'un programme en langage de très haut niveau, en fait proche de la notation mathématique usuelle. A ce programme sont associées des propriétés intrinsèques, comme la base de temps et sa latence, qui se rapportent à une exécution sur un processeur idéalisé. La phase de synthèse peut être vue, en première approximation, comme l'émulation du processeur idéal sur le processeur réel. Il est possible que ces remarques puissent guider la conception de nouveaux algorithmes, de nouvelles architectures parallèles, et surtout de nouveaux langages de programmation.

Bibliographie

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass, 1986.

[2] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM TOPLAS*, 9(4):491–542, October 1987.

[3] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston / Dordrecht / London, 1988.

[4] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.

[5] Paul Feautrier. Asymptotically efficent algorithms for parallel architectures. In M. Cosnard and C. Girault, editors, *Decentralized System*, pages 273–284, IFIP WG 10.3, North-Holland, December 1989.

[6] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[7] David J. Kuck. *The Structure of Computers and Computations*. J. Wiley and sons, New York, 1978.

[8] David J. Kuck, Toichi Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Trans. on Computers*, C-12:1293–1310, December 1972.

[9] Leslie Lamport. The parallel execution of do loops. *CACM*, 17:83–93, February 1974.

[10] Lee-Chung Lu. A unified framework for systematic loop transformations. *SIGPLAN Notices*, 26:28–38, July 1991. 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming.

[11] William Pugh. Uniform techniques for loop optimization. *ACM Conf. on Supercomputing*, 341–352, January 1991.

[12] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260, Manchester University Press, December 1987.

[13] Mourad Raji-Werth and P. Feautrier. On parallel program generation for massively parallel architectures. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*, North-Holland, October 1991.

[14] A. Schrijver. *Theory of linear and integer programming*. Wiley, NewYork, 1986.