Papiers présentés à la conférence Renpar 2002

Yaya SLIMANI et Denis TRYSTRAM

2 juin 2004

Introduction générale

L'objectif de cette section est simplement de faire commencer mon article sur la bonne page.

Chapitre 1

Parallélisation automatique, Histoire et perspectives

1.1. Introduction

Quelle idée de vouloir faire faire plusieurs actions à la fois à nos ordinateurs ! Cette performance dont nous serions bien incapables, pourquoi l'imposer à des machines dont nous avons décidé une fois pour toute qu'elles sont moins intelligentes que nous ? Du paradis de la programmation séquentielle, qui nous a chassé ?

Les coupables sont bien faciles à trouver : ce sont les électroniciens et leurs complices, les architectes informatiques. Les électroniciens ont décidés de relever le défi de la loi de Moore, et divisent par deux la finesse de leurs gravures tous les dix-huit mois (ou presque). Comme sous-produit de cette chasse à la poussière et aux erreurs d'alignement, ils entassent de plus en plus de portes logiques sur une même surface de silicium. Les architectes, bien sûr, ne veulent pas laisser ces portes oisives, et comme il ne sert à rien d'augmenter indéfiniment la précision des nombres, on en vient à multiplier les opérateurs indépendants : et voilà le parallélisme !

Les premiers architectes à doter leurs ordinateurs de parallélisme ont tout fait pour le cacher aux programmeurs ou pour le rendre inoffensif. Jusqu'à une époque récente, le parallélisme disponible au niveau le plus fin est resté caché derrière des interfaces de programmation séquentiels. Ces processeurs, superscalaires ou autres, ont donc du être dotés, par la force des choses, de paralléliseurs automatiques cablés, donc forcément limités. Ces paralléliseurs cablés ont d'ailleur l'avantage douteux de consommer une

Chapitre rédigé par Paul FEAUTRIER.

bonne partie des portes logiques en excès : le serpent se mord ainsi la queue ! Les processeurs VLIW, et plus récemment EPIC, où le parallélisme doit être pris en charge par le programmeur ou le compilateur, sont d'apparition récente et leur succès n'est pas encore assuré.

De même, les premiers multiprocesseurs symétriques sont apparus vers 1968. Mais il était bien entendu que chaque processeur traiterait un programme indépendant. Le parallélisme ainsi conçu est inaccessible à l'utilisateur, et ne sert qu'à augmenter la productivité de la machine. Ce n'est qu'avec l'apparition du système Unix que les programmeurs ont été jugé dignes de toucher au parallélisme ; encore cette dignité leur est elle fortement contestée. En effet, ils sont contraints de passer par le filtre des langages de programmation usuels. Ceux-ci sont tous modelés sur leur ancêtre commun, Fortran, qui ne fait que reproduire la structure des ordinateurs séquentiels du début des années 60. Ces langages ne connaissent pas le parallélisme et on doit recourir à des artifices – par exemple, des appels à une bibliothèque spéciale – pour l'utiliser.

Et pourtant ... Tous les systèmes permettant de cacher le parallélisme atteignent maintenant leurs limites. Les grappes et les grilles de calcul peuvent mettre en œuvre plusieurs milliers de processeurs. Une simple puce destinée au traitement du signal peut contenir de 50 à 100 unités fonctionnelles indépendantes. Qui va trouver le parallélisme nécessaire ? Un paralléliseur matériel a l'avantage de connaître complétement l'état présent du programme – registres, pointeurs, etc. – ainsi que des informations sur son passé – du moins s'il a pris soin de les recueillir. Par contre, il n'a aucune vision de l'avenir et ne travaille en général que sur une fenêtre de quelques dizaines d'instructions. La situation est inverse si la parallélisation est à la charge du programmeur. Il a une vision complète de son programme et de ses données, mais cette vision est très synthétique. De plus, il est susceptible de commettre des erreurs s'il rentre trop dans les détails. Son domaine est donc le parallélisme à gros grain, bien adapté aux grappes et aux grilles.

La parallélisation automatique se situe entre ces deux extrêmes. En principe, un compilateur peut entrer dans les détails sans se fatiguer ni se tromper. Il a de plus une vision complète du programme. Ce qui lui manque, ce sont les données, mais aussi les spécifications du programme, qui ne sont pas toujours déductibles du code source. Le reste de cet article a pour but de présenter les concepts de base de la parallélisation automatique, d'en expliquer la genèse et l'aboutissement : le modèle polyédrique. La parallélisation automatique doit maintenant résoudre certains problèmes que le modèle polyèdrique laisse en suspend : nous en présenterons quelques-un en conclusion.

1.2. Concepts de Base

1.2.1. *Instructions, Opérations, Ordre d'Exécution*

En sémantique dénotationnelle, un programme est vu comme la représentation textuelle d'une fonction, dont l'argument est l'état de la mémoire au démarrage du programme, et dont le résultat est l'état de cette même mémoire à la terminaison du programme. Je suppose ici que les programme à paralléliser se terminent toujours. Deux programmes sont équivalents si la fonction qu'ils implémentent est la même. De ce point de vue, le découpage du programme est sans importance. Il est indifférent de coder une action à l'aide d'une seule instruction complexe ou de deux instructions plus simples. Il n'en est pas ainsi en programmation parallèle, ne serait-ce que parce qu'on peut trouver du parallélisme dans le second cas, mais surement pas dans le premier. On est donc conduit, pour trouver beaucoup de parallélisme, à découper le programme aussi finement que possible. On arrive ainsi au concept d'opération, qui représente une exécution (ou une instance) d'une instruction. Les opérations sont créées par les instructions de contrôle. Par exemple, une boucle contenant une instruction et qui fait 1000 itérations engendre 1000 opérations.

Mais un programme ne se résume pas à la liste de ses opérations, pas plus qu'un morceau de musique ne se ramène à l'ensemble des notes à jouer. L'ensemble des opérations est ordonné, et changer l'ordre des opérations change le programme, de même que changer l'ordre des notes change la mélodie. En résumé, un programme se décrit par l'ensemble E de ses opérations, et par l'ordre, noté $\prec$, dans lequel ces opérations doivent être jouées.

1.2.2. *Indéterminisme*

Soit u et v deux opérations d'un programme. Si celui-ci est séquentiel, nous pouvons toujours dire laquelle des deux est exécutée la première. En fait, une bonne partie de l'enseignement élémentaire de la programmation est destinée à nous donner une intuition de cet ordre d'exécution. Nous avons tous appris, par exemple, que dans un nid de boucles, la boucle la plus interne tourne « plus vite » que les boucles externes. Nous dirons que pour un programme séquentiel l'ordre $\prec$ est total.

Imaginons maintenant que nous avons réparti les opérations d'un programme entre deux processeurs P1 et P2 qui partagent une même mémoire. Nous pouvons toujours spécifier l'ordre de deux opérations exécutées par le même processeur. Pour de multiples raisons, nous ne pouvons par contre rien dire pour les opérations de deux processeurs différents : les horloges peuvent dériver, il peut y avoir des interférences au niveau de la mémoire ou des intrusions du système d'exploitation. On dira que l'ordre d'exécution est maintenant partiel. Si ce désordre est intolérable, on pourra être amené à introduire des instructions de synchronisation, dont le but est de garantir l'ordre d'opérations excutées sur des processeurs différents. Il est clair que le degré de parallélisme et le degré d'ordre sont liés : un ordre total correspond à un programme séquentiel et un ordre partiel à un programme parallèle. L'ordre vide, correspond aux programmes *embarassingly parallel* où toutes les opérations peuvent être exécutées en même temps.

Il n'en reste pas moins que si l'on pointe les accès à la mémoire lors d'une exécution paralléle, on trouvera qu'ils ont été exécutés dans un certain ordre total. Il existe en effet un dispositif matériel, l'arbitre du bus mémoire, qui interdit les accès simultanés. Cet ordre doit être compatible avec l'ordre des opérations, et si celui-ci est partiel, il existe plusieurs posibilités pour l'ordre des accès. Comme le résultat du programme dépend en général de cet ordre, le résultat du programme parallèle va varier d'une exécution à l'autre : un programme parallèle est souvent non-déterministe. On notera au contraire qu'un programme séquentiel n'a qu'une seule exécution possible : il est donc toujours déterministe.

1.2.3. *Dépendances*

Comme l'objectif de la parallélisation automatique est de trouver un programme parallèle équivalent à un programme séquentiel donné, il faut trouver des conditions qui garantissent qu'un programme paralléle est déterministe. Considérons un programme de deux opérations, $u; v$. Dans sa version parallèle, il y a deux ordres possibles, $u; v$ comme en séquentiel et l'ordre inverse $v; u$. Pour que le programme parallèle soit déterministe, il faut que ces deux exécutions donnent le même résultat, ou encore que u et v commutent. Deux opérations qui ne commutent pas sont dites en dépendance, et on écrit $u \perp v$. On peut montrer que pour qu'un programme parallèle soit déterministe, il suffit que deux opérations en dépendances soient bien ordonnées. Si $\prec_{//}$ est l'ordre d'exécution paralléle, la condition :

$$u \prec v, u \perp v \Rightarrow u \prec_{//} v$$

garantit l'équivalence du programme séquentiel et du programme parallèle. Le graphe dont les sommets sont les opérations, et tel qu'il y ait un arc de u vers v ssi $u \prec v$ et $u \perp v$ est le graphe de dépendance (détaillé) du programme séquentiel. Ce graphe est acyclique, et l'ordre $\prec_{//}$ en est la fermeture transitive. On peut donc considérer que le problème est résolu si nous savons, d'une part calculer le graphe de dépendance, et d'autre part synthétiser le programme parallèle dont il est l'ordre d'exécution. Nous allons maintenant présenter les diverses solutions qui ont été apportées à ces problèmes.

1.3. La parallélisation classique

Il est relativement facile d'écrire la condition pour que deux opérations commutent. On peut utiliser par exemple la méthode de l'exécution symbolique : on exécute le programme en remplaçant les calculs arithmétiques par des calculs algébriques. On doit ensuite prouver – ou réfuter – la formule qui exprime l'égalité des résultats obtenus. Il est facile de se rendre compte que cette preuve peut faire appel à l'ensemble des mathématiques. Or, en l'état actuel de notre science, il n'est pas question d'incorporer un démonstrateur de théorèmes dans un compilateur.

D'où l'idée de se contenter d'une condition *suffisante* de commutation. Celle qui est universellement utilisée est due à Bernstein [BER 66]. On commence par déterminer, pour chaque opération u, l'ensemble $M(u)$ des cellules de mémoire qu'elle modifie, et l'ensemble $R(u)$ des cellules de mémoire qu'elle consulte. Il est alors facile de se convaincre que pour que u et v commutent, il suffit que :

$$M(u) \cap M(v) = M(u) \cap R(v) = R(u) \cap M(v) = \emptyset.$$

On postule que si l'une de ces conditions n'est pas vérifiée, les deux opérations sont en dépendances. On se ramène ainsi à la détermination des ensembles lus et modifiés. Si le programme est un "bloc de base" (pas d'instructions de contrôle) opérant sur des variables scalaires, le calcul de ces ensembles, puis celui de leurs intersections est trivial. Les choses deviennent plus complexes dès que l'on effectue des calculs d'adresses. On sait par exemple que dans certains langages (C, C++) un pointeur peut désigner pratiquement n'importe quelle cellule de mémoire. Le calcul des ensembles lus et modifiés nécéssite une analyse de pointeurs très difficile ; la parallélisation des langages à pointeurs est donc encore un sujet ouvert. Si le calcul d'adresse est une indexation, les choses sont plus simples, surtout si l'on suppose que le programme à paralléliser est correct. Dans ces conditions, deux cellules de tableau ne peuvent coïncider que si elles appartiennent au même tableau et que si leurs indices sont égaux deux à deux.

1.3.1. *Parallélisation d'un nid de boucle*

Les nids de boucles sont la cible préférée des paralléliseur automatiques, car ils sont susceptibles de fournir beaucoup de parallélisme, et parce que la synthèse du programme parallèle est particulièrement simple. Le but du jeu est de prouver que toutes les itérations d'une boucle sont indépendantes, ou au contraire d'exhiber un contre-exemple. Ce contre-exemple doit être constitué de deux itérations *distinctes* qui ne vérifient pas les conditions de Bernstein. Puisque les deux itérations sont distinctes, l'une est exécutée avant l'autre, et nous pouvons choisir arbitrairement laquelle est la première, puisque les conditions de Bernstein sont symétriques. Le corps de la boucle peut être composé de plusieurs instructions, et la dépendance peut être engendrées par deux quelconques d'entre elles, disons S et T. Enfin, toute cellule de mémoire qui figure dans l'intersection non vide des ensembles lus ou modifiés appartient à un tableau A qui est utilisé à la fois par S et par T. L'ensemble de ces conditions se met sous la forme d'un système de contraintes (égalité des indices, légalité des itérations, ordre d'exécution) où les inconnues sont deux valeurs distinctes du compte-tours de la boucle, i et i'.

Le problème est de discuter ce système, c'est-à-dire de savoir s'il existe i, i' entiers satisfaisant toutes les contraintes ci-dessus, ou, dit d'une autre façon, si ce système de contraintes est *faisable*. Ce problème est indécidable en général : il contient le dixième problème de Hilbert. Pour y répondre, on doit soit restreindre la classe des programmes admissibles, soit se contenter de réponses approximatives. Par exemple, si on exige que les fonctions d'indice et les bornes des boucles soient des formes affines, on se ramène à un problème de programmation linéaire en nombres entiers, qui est décidable mais NP-complet. Une réponse approximative est acceptable à condition qu'elle soit pessimiste, c'est-à-dire qu'elle trouve plus de dépendances qu'il n'y en a réellement. On peut par exemple ignorer toutes les contraintes non-linéaires, ce qui ne peut qu'élargir l'espace des solutions.

De très nombreux travaux ont été mené, depuis les années 80, pour trouver des tests de dépendance efficaces. Le test du PGCD, par exemple, vérifie que les contraintes équationelles ont des solutions entières. Les tests de Banerjee [BAN 88], au contraire, vérifient que chaque équation peut avoir, dans l'intervalle donné par les bornes des boucles, une solution rationelle. Ces tests sont très peu précis. De très nombreux travaux ont cherché à augmenter cette précision sans trop faire croitre la complexité. Vers 1985, il est apparu que le plus simple était de traiter les systèmes aux dépendances comme des programmes linéaires en nombre entiers (ou en variables continues) et d'appliquer des méthodes générales. Rémi Triolet [IRI 90] a par exemple retrouvé la méthode de Fourier-Motzkin, tandis que je me suis orienté plutôt vers le Simplex et ses dérivés.

A l'heure actuelle, les performances des ordinateurs ont tellement augmentées que le calcul des dépendances, autrefois sur le chemin critique, prend maintenant un temps négligeable. Les combinaisons de test les plus utilisées sont :

– Le test du PGCD suivi d'un test de Fourier-Motzkin donne de très bonnes performances, au prix d'une perte de précision en général négligeable.

– Le test Omega de Bill Pugh [PUG 91], extension entière du test de Fourier-Motzkin, est exact et a une très bonne efficacité pour les problèmes de petite taille.

– La méthode des coupes de Gomory, extension entière de la méthode du Simplexe, ne devient intéressante que pour des problèmes de taille moyenne.

1.3.2. *Transformations de programmes*

Les chercheurs des années 80 ont été surpris de constater que les méthodes que nous venons de présenter ne trouvaient pas beaucoup de parallélisme dans les programmes usuels. Il a fallu un peu de temps pour comprendre que ce manque de parallélisme est lié aux optimisations que les programmeurs séquentiels effectuent sans y penser. Par exemple, utiliser une variable temporaire scalaire dans une boucle induit des dépendances, et la boucle reste séquentielle. De même, remplacer une multiplication par une incrémentation crée une dépendance. La solution est de transformer le programme initial pour essayer de retrouver la version non optimisée. Ainsi, on peut envisager un schéma de parallélisation en deux étapes :

– Appliquer au programme initial des transformations séquentielles qui n'en changent pas le résultat, mais en augmente le parallélisme.

– Trouver le parallélisme syntaxique du programme transformé.

Le critère d'équivalence peut être ajusté en fonction des besoins. On peut exiger par exemple que la suite des valeurs prises par toutes les variables du programme soit la même pour le programme original et le programme transformé. On peut aussi restreindre l'équivalence à certaines variables particulières dites de sortie. Enfin, s'il s'agit d'un programme de type exploratoire (trouver un minimum ou une limite par exemple), on peut exiger que le test de terminaison ne change pas, sans rien imposer à la méthode de recherche. Ces conditions d'équivalence deviennent de moins en moins contraignantes, mais aussi de plus en plus difficile à administrer automatiquement. On peut classer les transformations de la première espèce de la façon suivante :

– Améliorer le programme séquentiel. On peut soit chercher à simplifier le travail du paralléliseur (éliminer les `gotos`, rendre explicite les indexations) soit effectuer les optimisations "obligatoires" (détecter les calculs invariants dans les boucles). Il est à noter que sortir les calculs invariants d'une boucle diminue le parallélisme ; mais c'est un parallélisme qui ne sert à rien.

– Changer l'ordre des calculs. L'objectif est de regrouper différemment les opérations, de façon à rapprocher les opérations parallèles. C'est une analyse de dépendance « généralisée » qui permet de savoir si une telle transformation est valide ou non.

– Changer les structures de données. On exige toujours de calculer les mêmes valeurs, mais on s'autorise à les ranger différemment en mémoire. Le cas typique est celui de l'expansion de scalaire. Il consiste à ranger les valeurs successives d'une variable scalaire dans des cellules distinctes d'un tableau. Cette transformation suffit pour casser certaines dépendances.

Pour chacune de ces transformations des méthodes ad hoc ont été développées. Elles comportent trois parties. La transformation est elle valide ? Est-elle utile ? Quel est l'algorithme permettant d'exécuter la transformation ? L'emploi de ces transformations peut aboutir rapidement à des explosions combinatoires. Il y a beaucoup de transformations, elles ne sont pas indépendantes, et chacune peut être appliquée en un grand nombre de sites. Une première solution est de guider le choix des transformations par des heuristiques, en général adaptées à la machine cible. Par exemple, il vaut mieux paralléliser les boucles externes pour un multiprocesseur, et les boucles internes pour un ordinateur vectoriel. L'autre solution est de construire des algorithmes de parallélisation intégrés, qui effectuent à la fois les transformations et la recherche du parallélisme syntaxique sur le résultat. Le premier exemple connu d'algorithme intégré est l'algorithme de Kennedy et Allen, que nous allons présenter dans la section suivante.

1.3.3. *L'algorithme de Allen et Kennedy*

La transformation « éclatement de boucle » essaye de remplacer une boucle sur deux instructions par deux boucles sur une instruction :

```
for(i= ....){                          for(i= ....)
   S1;                                    S1;
   S2;              ==>                for(i= ....)
   }                                      S2;
```

Il s'agit d'un changement de l'ordre des calculs, qui a pour but de mieux répartir les dépendances et donc de trouver plus de parallélisme. La transformation « échange de boucle » :

```
for(i= ...)                            for(j= ...)
   for(j= ...)      ==>                   for(i= ...)
      S;                                     S;
```

a pour but de placer le parallélisme à l'endroit le plus favorable.

L'algorithme de Kennedy et Allen [ALL 87] tente de trouver directement le programme parallèle obtenu en appliquant judicieusement ces deux transformations, puis en recherchant le parallélisme syntaxique. Son point de départ est un graphe de dépendance résumé, où les sommets sont les instructions, et où il y a un sommet de S à T si une itération de S au moins est en dépendance avec une opération de T. De plus, on note la profondeur de chaque dépendance.

Le résultat fondamental est que chaque composante fortement connexe (cfc) du graphe de dépendance correspond à une boucle du programme parallèle. Cette boucle est séquentielle si la cfc contient une dépendance de profondeur 0, et parallèle sinon. Enfin, l'ordre des boucles est donné par un tri topologique de la condensation acyclique du graphe de dépendance. Les indications ci-dessus permettent d'écrire les boucles de plus haut niveau du programme parallèle. Pour obtenir les boucles internes, on simplifie le graphe de dépendance en enlevant les arcs qui joignent deux cfcs différentes ainsi que les arcs de profondeur 0. On peut dire que ces arcs ont été « satisfaits » par les boucles externes et la façon de les ordonner. On recommence ensuite en remplaçant la profondeur 0 par la profondeur 1. On poursuit jusqu'à avoir construit toutes les boucles. Il ne reste plus qu'à recopier les instructions du programme à leur place naturelle. Le programme obtenu convient tel quel à un multiprocesseur. Si la cible est une machine vectorielle, on montre qu'il est toujours possible d'échanger les boucles de façon à placer les boucles parallèles à l'intérieur des nids de boucles.

L'algorithme d'Allen et Kennedy est très puissant et de faible complexité, entre autre parce qu'il existe un algorithme – dû à Tarjan – de détermination de cfcs en temps linéaire. Darte et Vivien on montré qu'il est optimal si l'on ne connait rien d'autre que le graphe de dépendance résumé ; mais il est facile de construire des exemples contenant du parallélisme que l'algorithme de Kennedy et Allen échoue à trouver.

1.4. Le modèle polyédrique

Contrairement aux compilateurs ordinaires qui travaillent au niveau des instructions et manipulent des arbres, un paralléliseur passe son temps à manipuler des ensembles ou des relations : par exemple l'ensemble des opérations d'un programme, ou sa relation de dépendance, ou son ordre d'exécution. L'usage en informatique (voir par exemple les opérations ensemblistes de Pascal) est de représenter les ensembles en extension, en donnant soit la liste de leurs éléments, soit le graphe de leur fonction caractéristique. Mais les ensembles ci-dessus sont beaucoup trop gros pour être représentés en extension : un ordinateur d'une puissance d'un MFlops (donc rien suivant les standards actuels) qui exécute un programme durant une seconde engendre un million d'opérations, et le graphe de dépendance est encore plus gros. Il faut donc passer à une représentation en intension, c'est-à-dire donner une formule mathématique pour la condition d'appartenance d'un objet à l'ensemble. Par exemple, les opérations engendrées par la boucle :

```
for(i=0; i<n; i++)
   S;
```

sont représentées par l'ensemble $E = \{\langle S, i\rangle \mid 0 \leq i < n\}$. On notera que la taille de cette formule est la même pour $n = 100$ ou $n = 10^6$. Par contre, pour répondre à la question $E = \emptyset$?, il ne sufft plus d'inspecter l'ensemble : il faut démontrer un théorème, d'ailleurs très simple.

Comme il n'est pas (encore ?) envisageable d'inclure un démonstrateur de théorèmes dans un compilateur, nous devons choisir des représentations d'ensembles où les opérations dont nous avons besoins sont effectives et où les propriétés qui nous intéressent sont décidables. Le modèle polyédrique est l'un des choix possibles. Un polyèdre convexe est l'ensemble des solutions d'un système de contraintes affines :

$$P = \{x \mid Ax \geq b\},$$

où x est un vecteur de dimension n, b un vecteur de dimension m, et A une matrice $m \times n$. En fait, en parallélisation automatique, on est plutôt amené à manipuler des Z-polyèdres, ensemble des solutions *entières* d'un système de contraintes affines. Il est bien facile de voir que l'ensemble des itérations d'un nid de boucle est un Z-polyèdre, à condition que les bornes soient des formes affines. Il en est de même pour la relation de dépendance. Nous avons déja parlé des algorithmes permettant de décider si un polyèdre est vide. Un sous-produit de ces algorithmes donne directement le minimum ou le maximum d'un Z-polyèdre. L'ensemble des Z-polyèdres est clos par intersection et par image inverse par une fonction affine. Il faut légèrement en généraliser la définition pour avoir la cloture par union et image directe. Nous pouvons donc reformuler en terme de polyèdres tous les algorithmes de la section précédente. Mais il est possible d'aller beaucoup plus loin. En particulier, toutes les transformations qui changent l'ordre des exécutions peuvent se représenter comme des changements de base dans l'espace des itérations. Appliquer successivement une transformation caractérisée par la matrice M_1 puis une autre transformation de matrice M_2, c'est appliquer la transformation de matrice $M_2.M_1$. Au lieu de procéder par transformations successives, il est donc envisageable de rechercher directement la matrice de transformation M qui, par exemple, met en évidence le plus de parallélisme. Nous allons décrire quelques algorithmes permettant de construire la matrice de transformation convenable.

1.4.1. *Ordonnancement*

Certains ordinateurs ont un fonctionnement synchrone : à chaque top de l'horloge, tous les processeurs exécutent la même instruction. Il est naturel d'affecter à chaque opération une date d'exécution (en fait, un numéro de top d'horloge). Toutes les opérations exécutées à la même date sont exécutées en parallèle. La fonction qui donne la date d'exécution d'une opération s'appelle un ordonnancement; on la note en général θ. Curieusement, cette idée s'étend sans difficulté aux ordinateurs asynchrones. Il suffit de décider que θ définit l'ordre d'exécution parallèle. L'opération u est exécutée avant v ssi $\theta(u) < \theta(v)$. Il est clair que si $\theta(u) = \theta(v)$ les deux opérations ne sont pas ordonnées, donc s'exécutent en parallèle.

Il n'y a pas de gros effort à faire pour généraliser une dernière fois. Il n'est pas nécessaire que la valeur de θ soit entière ; il suffit qu'elle appartienne à un ensemble ordonné. On peut penser en particulier à l'ensemble des vecteurs à coordonnées entières, ordonné suivant l'ordre lexicographique :

$$x \ll y \equiv \exists k : x[1..k] = y[1..k] \wedge x[k+1] < y[k+1].$$

Il est facile de voir que cet ordre est total : il reproduit l'ordre dans lequel défilent les chiffres d'une montre digitale, ou l'ordre dans lequel les itérations d'un nid de boucles sont exécutées.

Nous savons que deux itérations en dépendances doivent être bien ordonnées. Ceci se traduit par la contrainte de causalité :

$$\forall u, v : u\,\delta\,v \Rightarrow \theta(u) < \theta(v).$$

Cette formule résume un très grand nombre de contraintes. Pour la résoudre, il faut tout d'abord éliminer les quantificateurs, et pour celà supposer que la fonction θ est affine. Nous verrons plus loin que cette hypothèse est indispensable pour une autre raison. Si θ est affine, alors il suffit de vérifier la contrainte de causalité en un nombre fini de points, les sommets du polyèdre des dépendances [QUI 87]. On peut également utiliser le lemme de Farkas, qui assure que la quantité $\theta(v) - \theta(u)$ est combinaison linéaire à coefficients positifs des inégalités définissant le polyèdre des dépendances [FEA 92a, FEA 92b]. Quelque soit la méthode utilisée, on obtient un système de contraintes affines dont les inconnues sont les coefficients de θ. Il est facile de résoudre ce système par les méthodes de la programmation linéaire, en prenant pour fonction objectif, par exemple, la durée totale du programme. Il est clair que si cet algorithme réussit, alors la durée d'exécution est fonction affine de la taille du problème. Or, il est très facile de construire des contre-exemples de complexité parallèle supérieure. Il leur correspond des systèmes de contraintes infaisables. On peut agrandir l'espace des solutions en passant à un ordonnancement à plusieurs dimensions. On montre que la méthode finit toujours par trouver un ordonnancement, dont la dimension maximale est la profondeur du nid de boucles le plus complexe.

Darte et Vivien [DAR 00] ont démontré que presques tous les algorithmes de parallélisation connus, y compris l'algorithme de Allen et Kennedy, sont des variantes de cet algorithme d'ordonnancement. Chaque variante correspond à un agrandissement du polyèdre des dépendances. Comme chaque agrandissement fait perdre des solutions, il n'est pas étonnant que certains algorithmes n'arrivent pas à trouver tout le parallélisme disponible. Plus récemment, Vivien [AéR 03] a démontré que l'algorithme est optimal parmi tous ceux qui construisent des ordonnancements affines. Il suffit d'abandonner cette dernière contrainte, par exemple en autorisant les ordonnancements affines par morceaux, pour voir l'optimalité disparaître.

1.4.2. *Placement*

Certains ordinateurs parallèles sont composés d'un grand nombre d'ordinateurs séquentiels interconnectés par un réseau. Dans ce type d'architecture, chaque processeur a un accès très rapide à sa propre mémoire (en 2004, de l'ordre de 100ns) ; l'accès à la mémoire d'un autre processeur doit passer par le réseau et prend quelques dizaine de microsecondes. Pour obtenir de bonnes performances, il faut donc, dans la mesure du possible, *placer* dans un même processeur une donnée et les opérations qui l'utilisent. Comme, dans un programme normal, toute donnée finit par contribuer à tout résultat, on ne peut pas aller trop loin dans cette direction : on est obligé de tolérer des communications résiduelles sous peine de voir le programme et ses données se regrouper sur un seul processeur.

On peut formaliser le problème de la façon suivante. On numérote les processeurs, et on place l'opération u sur le processeur $\pi(u)$. De la même façon, la cellule de mémoire x est placée sur le processeur $\Pi(x)$. Si l'opération u utilise la cellule de mémoire $f(u)$ (f est une fonction d'indexation), alors on écrit que la donnée et le calcul sont dans le même processeur :

$$\pi(u) = \Pi(f(u)).$$

Si on suppose que les fonctions de placement et que la fonction d'indexation sont affines, on peut déduire de cette contrainte un système linéaire dont les inconnues sont les coefficients des fonctions de placement. Le système obtenu en rassemblant toutes ces équations est homogène, donc il a toujours la solution triviale 0. Celle-ci correspond à un programme dont toutes les données et tous les calculs sont dans le processeur 0 ; il n'a aucun parallélisme.

Il se peut que la matrice du système ne soit pas de rang plein. Elle a donc des solutions non nulles ; celle-ci correspondent à un programme parallèle sans communications. Cette situation est assez rare. Dans le cas où la matrice du système est de rang plein, il faut ignorer certains des accès à la mémoire, jusqu'à ce qu'une solution non triviale existe. Les accès ignorés correspondent à des communications résiduelles. que l'on a intérêt à minimiser. On y parvient en estimant le volume de communication correspondant à chaque accès, et en ignorant de préférence les accès de plus petit volume [FEA 96].

Il reste une dernière difficulté. Le nombre de valeurs prise par une fonction affine des indices de boucles est de l'ordre de grandeur du nombre d'itérations de l'une des boucles. Ce nombre n'a aucune raison de correspondre au nombre de processeurs de la machine cible ; il est en général beaucoup plus grand. On convient donc que les valeurs de la fonction de placement sont des numéros de processeurs *virtuels*. On confie plusieurs processeurs virtuels à un seul processeur réél par une méthode de *tuilage*.

1.4.3. *Génération de code*

Le lecteur attentif s'est sans doute rendu compte que les deux méthodes ci-dessus ne fournissent pas un programme parallèle, mais seulement des indications permettant de le rédiger. Considérons par exemple un programme qui a été ordonnancé à l'aide de la fonction θ. Pour écrire le programme parallèle, nous devons d'abord faire s'écouler le temps, de l'instant de départ du programme, jusqu'à la date de la dernière instruction. On suppose en général que la première opération est exécutée à l'instant 0. La date de la dernière opération, que l'on calcule en résolvant un problème de programmation linéaire, est la durée totale du programme parallèle ou latence, notée L. A chaque instant t, on exécute en parallèle les opérations ordonnancée à la date t. Le programme parallèle peut se schématiser de la façon suivante :

```
for t = 0, L
    forall {u|θ(u) = t}
           do u
```

Le front à la date t, $\{u|\theta(u) = t\}$, est un polyèdre ou une union de polyèdre dont il s'agit d'énumérer les points entiers au moyen de boucles parallèles. On y parvient en calculant pour chaque boucle, sa borne inférieure et sa borne supérieure par des méthodes de programmation linéaire. On se convaincra facilement que la même méthode s'applique à la construction de programmes à partir d'un placement, ou à la transformation "inversion de boucles", ou à l'écriture du code de communication pour une machine à mémoire distribuée. Ce problème a fait l'objet de nombreuses études dans le cadre du modèle polyédrique ([ANC 91, DAR 93, COL 95, KEL 92, AMA 93, GRI 94, XUE 93, QUI 00, BAS] entre autres). La raison en est que le programme transformé par ordonnancement ou placement a des boucles plus complexes que le programme original, et que si ces boucles ne sont pas écrites avec le plus grand soin, elles peuvent faire perdre tout le bénéfice d'une optimisation ou d'une parallélisation.

1.4.4. *Analyse du flot des données*

Tout le monde sait que, contrairement aux mathématiques, dans un programme, les valeurs calculées ne sont jamais nommées directement, mais seulement par l'intermédiaire des cellules de mémoire qui les contiennent. Ainsi, en mathématique, la soi-disant variable x est en réalité une constante dont la valeur, inconnue au début de la résolution d'un problème, devient connue à la fin. Au contraire, la variable informatique `x` représente une cellule de mémoire fixe dont le contenu varie au cours de l'exécution du programme. On devrait, au sens strict, considérer `x` comme une fonction du temps. Il est presque toujours impossible d'expliciter cette fonction. Si on le pouvait, il suffirait de calculer sa valeur à la date de terminaison du programme pour en connaître les résultats sans avoir à l'exécuter ! L'analyse du flot des données a pour but d'analyser cette fonction autant que faire se peut.

Toute valeur utilisée dans un programme provient d'une instruction d'affectation, du moins si l'on veut bien considérer que les initialisations et les instructions de lecture sont des affectations un peu particulières. Chaque fois que l'on utilise une valeur, il est intéressant de savoir quelle est l'opération qui l'a créée. C'est évidemment l'opération d'écriture à la bonne adresse la plus proche dans le passé. Cette opération est unique : on l'appelle la *source* de la valeur lue. Cette source n'est pas fixe : dans un programme ayant des boucles et des tableaux, elle va dépendre des indices du tableau et de l'état d'avancement du programme, c'est-à-dire des compte-tours des boucles englobantes.

Les opérations qui peuvent être la source d'une valeur appartiennent à un Z-polyèdre, la source effective en est le maximum lexicographique, qui peut être trouvé par la programmation linéaire en nombre entiers. Le seul point délicat est que ce calcul de maximum doit être mené paramétriquement par rapport aux compte-tours des boucles englobantes, par exemple à l'aide de logiciels comme PIP [FEA 91] ou Omega [PUG 93].

Le calcul de la fonction source fournit un grand nombre de renseignements utiles sur le programme original. Il permet par exemple de le mettre en forme à assignation unique, ce qui permet ensuite de le traiter comme un système d'équations mathématiques si on souhaite raisonner sur lui. La forme à assignation unique généralise l'expansion de scalaire et permet de trouver beaucoup plus de parallélisme, quitte à procéder ensuite à une compaction de la mémoire en fonction du parallélisme obtenu. L'analyse de la fonction source permet une meilleure estimation des volumes de communications, et simplifie l'écriture du code de communication ou de synchronisation.

1.5. Problèmes émergents en parallélisation automatique

Les paragraphes qui précèdent ont peut être donné au lecteur l'impression que la parallélisation automatique est une science achevée à qui il suffirait d'un effort déterminé d'implémentation pour que tous les problèmes de la programmation parallèle soient résolus. Il n'en est rien. L'essentiel de la difficulté vient du fait que le modèle polyédrique ne s'applique qu'aux programmes à controle statique, lesquels ne recouvrent à peu près que l'algèbre linéaire et le traitement du signal. On s'aperçoit rapidement que les programmes du monde réél ne sont pas (ou pas entièrement) à contrôle statique. Il faut aller au delà, soit en considérant le modèle polyédrique comme une première approximation (mais nous ne savons pas ce que pourrait être la deuxième approximation) soit comme un exemple de ce que pourrait être le soubassement d'une méthode d'analyse de programmes.

D'autre part, on retrouve en parallélisation automatique ce qui se passe en compilation optimisée : les techniques de parallélisation foisonnent, elle peuvent être en concurrence ou en synergie, il devient de plus en plus difficile de savoir par où commencer. La réponse à cette question devra sans doute passer à la fois par un effort d'uniformisation des méthodes de parallélisation et d'optimisation, par des méthodes d'analyse de cas et de planification issues de l'intelligence artificielle, et par l'écriture expérimentale de paralléliseurs aussi complets que possibles.

Au delà de ces questions fondamentales, on peut cependant isoler des problèmes plus ponctuels. En voici quelques uns : il y en a bien d'autres.

1.5.1. *Interprétation abstraite et parallélisation automatique*

L'interprétation abstraite est une méthode d'analyse qui revient en gros à « simuler » un programme en remplaçant les calculs sur des données concrètes par par des calculs dans un espace abstrait qui synthétise ou approxime l'espace des données concrètes. Si cet espace abstrait est bien choisi, d'une part la simulation se termine en temps fini, et d'autre part les résultats abstraits peuvent s'interpréter comme des propriétés des calculs concrets.

L'intérêt de la méthode est de fournir des résultats approximatifs quand un calcul exact est impossible. Un cas d'école est celui de la détermination des ensembles lus et modifiés. En parallélisation classique, ceci n'est possible que si les structures de données sont des tableaux, et si on peut extraire directement du programme les fonctions qui relient les coordonnées d'une opération aux indices d'un tableau. Actuellement, on espère pouvoir lire directement cette relation sur le texte du programme. On pourraît également associer de nouvelles variables aux indices de tableau, puis rechercher les relations de ces variables avec les autres variables du programme par la méthode de Cousot et Halbwachs [COU 78]. Cette approche aurait plusieurs avantages : fournir naturellement des approximations quand un calcul exact est impossible, et unifier un grand nombre de techniques disparates (détection des variables inductives, substitutions, etc.) On pourrait également envisager le calcul direct des dépendances par ce type de méthodes. Je ne connais à ce jour aucun résultat positif dans cette voie. Mais après tout, connaissant les ensembles lus et modifiés, le calcul des dépendances est facile : que pourrait apporter ici l'interprétation abstraite ?

Reste la question de l'analyse du flot des données. On sait qu'il en existe une version pour les variables scalaires, le calcul des chaînes de définition (*use-def chains*), qui se fait par interprétation abstraite. La question de savoir si la méthode se transpose au cas des tableaux est ouverte ; il n'existe à ma connaissance qu'une solution partielle, l'analyse des régions de Béatrice Creusillet.

1.5.2. *Les algorithmes de placement*

On a vu que les algorithmes de parallélisations usuels sont en général des algorithmes d'ordonnancement, mais il est tout aussi facile de se convaincre que les algorithmes de placement sont des algorithmes de parallélisation. En gros, un algorithme d'ordonnancement trouve les opérations qui peuvent être exécutées au même instant. Ce qui reste est séquentiel. Inversement, un algorithme de placement trouve les opérations qui doivent être exécutées sur le même processeur. Ce qui reste est parallèle. Les algorithmes de placement sont algorithmiquement plus simple que les algorithmes d'ordonnancement : il ne font appel qu'à l'agèbre linéaire plutôt qu'à la théorie des inégalités. Il ont aussi un important effet de bord : ils ont tendance à améliorer la localité, processeur par processeur.

Les algorithmes de placement posent cependant des problèmes non encore résolus. La génération du code correspondant à un placement donné peut se traiter par les méthodes de parcours de polyèdres vues plus haut, mais les directions d'optimisation ne sont pas les mêmes. La structure du programme dépend également de la nature de la machine cible. Les méthodes de placement ont été inventées pour les machines à mémoire distribuée. Dans ce cas, il faut écrire un code de communication, qui sert aussi de code de synchronisation. Si la machine est à mémoire partagée, il est possible de simuler une mémoire distribuée, comme le font les fanatiques de MPI. Mais l'efficacité de la méthode diminue quand le volume de communication augmente. Il vaut mieux utiliser directement la mémoire partagée pour éviter les communications, mais il faut écrire des synchronisations, et aussi traiter de délicats problèmes d'expansion de données.

Il semble que les méthodes de placement soient plus souples que les méthodes d'ordonnancement. On peut se poser des questions de réplication des données ou de calculs redondants, qui peuvent dans certains cas améliorer considérablement l'efficacité du programme. On peut également envisager des phases de redistribution des données entre des phases de calcul. La redistribution est un problème bien connu, pour lequel il existe des solutions efficaces. La question du placement optimal des redistributions est entièrement ouverte.

1.5.3. *Parallélisation Modulaire*

Les algorithmes de parallélisation ont en général une complexité plus que linéaire. Par exemple, un simple calcul de dépendance doit examiner tous les couples de références à un même tableau, ce qui demande un temps quadratique en la taille du programme. Un algorithme de placement construit un système de taille proportionelle à la taille du programme, mais ce système doit être résolu par une méthode de Gauss, ce qui prend un temps cubique en la taille du système. On pourrait multiplier les exemples : la conclusion est que les algorithmes de parallélisation ne sont pas facilement extensibles.

On connait la solution usuelle : on découpe le programme en *modules* que l'on traite séparémment. Cette méthode a l'inconvénient de faire perdre du parallélisme : précisémment celui qui existe entre modules. Le découpage naturel est celui que le programmeur a spécifié au moyen de sous-programmes, fonctions ou procédures. A ce propos, il ne faut pas confondre parallélisation interprocédurale et parallélisation modulaire. Dans la première, on suppose disposer de l'ensemble du programme, et on cherche à s'appuyer sur la structure en procédure pour simplifier le travail du paralléliseur. En parallélisation modulaire, au contraire, on parallélise module par module. La question est de décider de ce que l'on doit savoir sur les modules appelés pour paralléliser le module appelant, ou, dans l'autre sens, ce qu'il faut savoir sur le module appelant pour paralléliser le module appelé.

Le problème est analogue à celui que l'on se pose quand on fait de la compilation séparée. La solution est bien connue : d'une part, on normalise la méthode de transmission des données, ce qui dispense de toute information à ce sujet, d'autre part on conserve des informations sur les types des paramètres, pour que le vérifieur de type puisse franchir les limites des procédures. Cette information peut être recopiée, comme en C, ou stockée *hors texte* comme en Ada ou Ocaml. Que doit-on stocker pour que le paralléliseur puisse franchir les limites des procédures ?

La première idée, originalement proposée par Triolet et Irigoin [TRI 86], est d'assimiler une procédure à une super-instruction, et de calculer ses ensembles lus et modifiés ou *régions*. La méthode est assez simple à mettre en œuvre. Elle conduit souvent à des pertes de parallélisme. Par exemple, les méthodes d'ordonnancement sont spécialement efficaces si on les applique à des nids d'au moins deux boucles. Si le programmeur a distribué ces deux boucles entre deux procédures, ce parallélisme sera perdu. La méthode de l'ordonnancement hiérarchique, où l'on calcule et compose un ordonnancement par procédure, souffre du même défaut.

La solution est probablement de ne pas calculer un ordonnancement, mais de stocker les contraintes que doit satisfaire tout ordonnancement légal de la procédure, en se limitant aux contraintes visibles de l'extérieur. On peut ainsi remonter dans l'arbre des appels juqu'à l'ordonnancement du programme principal, puis redescendre pour ordonnancer chaque procédure. Cette méthode pose deux questions. Tout d'abord est-ce que l'on reste en permanence dans le cadre du modèle polyédrique, ou bien est-ce que des non linéarités apparaissent ? Que faire d'autre part des procédures récursives ?

1.5.4. *Ordonnancement sous contrainte de ressources*

Dans son acception générale, une ressource, c'est tout ce qui peut venir à manquer. En informatique, ce sont tous les dispositifs qui n'existent pas en quantité suffisante pour une exécution idéale d'un programme : unités de calcul, registres, mémoire, systèmes de communication, etc.

Quand j'ai présenté les méthodes d'ordonnancement, j'ai ignoré les problèmes de ressources. L'ordonnancement est déterminé uniquement à partir de la contrainte de causalité. Le résultat peut être un programme où les fronts ont beaucoup plus d'opérations qu'il n'y a de processeurs dans la machine cible. On s'en tire en tuilant chaque front : chaque processeur exécute un bloc d'opérations. Pour que le résultat soit efficace, il faut équilibrer la charge des processeurs. On peut le faire soit à la compilation, soit à l'exécution (méthode de l'auto-ordonnancement). Le résultat est bon, à condition qu'il y ait beaucoup plus de parallélisme que de processeurs.

On peut cependant essayer de construire directement des ordonnancements qui respecteraient les contraintes de ressource. Le cas d'école consiste à trouver un ordonnancement satisfaisant la condition :

$$\forall t \ \mathrm{Card}(\{u|\theta(u) = t\}) \leq P,$$

où P est le nombre de processeurs diponibles.

A l'heure actuelle, on connait des techniques de résolution pour le cas où le programme n'a qu'une seule boucle : le pipeline logiciel [LAM 88]. La raison essentielle en est que en une dimension, le calcul du cardinal figurant dans la formule ci-dessus est faisable. Le problème est encore ouvert en plusieurs dimensions. Outre l'heuristique qui consiste à tuiler un ordonnancement non contraint, on peut envisager de simuler les contraintes de ressources par des contraintes de données. Admettons que l'on associe une variable fictive `proc[p]` au processeur p et que toutes les opérations du programme écrivent dans `proc[i%P]`, où i est l'un quelconque des compteurs de boucles englobantes. Comme deux opérations en dépendance ne peuvent appartenir au même front, il est facile de voir que chaque front aura au plus P opérations. L'expérience montre que les programmes construits par cette méthode ne sont pas optimaux. Existe-t-il une approche plus directe ?

On peut également se demander si on peut construire des placements sous contraintes de ressources – je conjecture que oui – et faire de la parallélisation modulaire sous contrainte de ressources. Je pense que dans ce cas la réponse est négative, parce que la contrainte de ressources est une contrainte globale, qu'il est difficile de partitionner suivant les procédures.

1.5.5. *Au delà du modèle polyédrique*

La difficulté essentielle à laquelle doit faire face le modèle polyédrique, c'est que les programmes courants ne sont pas ou ne sont que partiellement polyédriques. Les indices et les bornes des boucles ne sont pas toujours affines. Il est très difficile de caractériser les ensembles lus et modifiés d'une instruction utilisant des pointeurs. Mais la difficulté majeure est l'existence des constructions qui modifient l'ensemble des opérations à exécuter : instructions conditionelles et boucles `while`. Pour traiter de tels programmes, plusieurs techniques ont été proposées.

Tout d'abord, dans certains cas limités, il est possible de mener l'analyse un peu au delà du linéaire. Par exemple, il suffit de quelques connaissances rudimentaires sur les exponentielles pour calculer les dépendances de l'algorithme de la transformée de Fourier rapide. Une autre technique consiste à détecter les fragments réguliers d'un programme irrégulier et à les paralléliser isolémment. Enfin, on peut essayer d'approximer le programme irrégulier par un programme régulier (par exemple, en supposant que les branches d'une conditionnelle sont toutes exécutées). Si l'approximation est bien conduite, le parallélisme trouvé pour l'approximation régulière est valide pour le programme original.

Pour aller plus loin, il faut revenir sur ce qui constitue un programme. On peut y distinguer trois régions, ou mieux encore trois tranches, au sens de l'expression anglaise *program slicing* : le contrôle, le calcul d'adresse et les traitements proprement dits. Les langages de programmation usuels n'imposent aucune séparation entre ces trois régions. Si l'on cherche à déterminer la tranche de programme qui influe sur le calcul d'un indice de tableau, il n'est pas exclu que l'on y trouve tout le programme. Or, ce n'est que si ces trois régions sont découplées que la parallélisation, et plus généralement l'optimisation d'un programme est possible. Le modèle polyédrique correspond au cas d'une séparation maximale : le contrôle et les calculs d'adresses ne dépendent que de quelques *paramètres de structure* : la taille d'une matrice ou la précision d'un échantillonage, par exemple. A l'inverse, un programme qui manipule des données dynamiques (par exemple, un programme de calcul formel), travaille sur des structures qui dépendent entièrement des données du problème ; ils sont donc très difficile à paralléliser.

Les méthodes de compilation dynamique ou d'interprétation partielle sont peut-être applicables à ce type de problème. La parallélisation serait renvoyée à un moment de l'exécution où suffisamment d'informations ont été calculées pour rendre le programme polyédrique ou quasi-polyédrique. On parlerait alors de parallélisation *just-in-time*. Ces méthodes, bien que prometteuses, posent de délicats problèmes d'efficacité : il faut que le temps de la parallélisation puisse être amorti par un calcul d'une durée significative.

1.5.6. *La structure absente*

Toute la parallélisation automatique, telle que nous l'avons vue jusqu'ici, repose sur l'hypothèse que le programme source renferme toutes les informations nécéssaires à sa parallélisation, et qu'il n'y a qu'à creuser assez profondément pour les trouver. Or il est très fréquent que cette hypothèse soit invalide. Un exemple simple est celui des listes. Il est impossible de spécifier en C qu'une structure de donnée est une liste. Tout au plus peut-on décrire un maillon ; il faut une analyse de pointeurs pour avoir une certitude. La situation est un peu meilleure en Java, bien que l'identification des listes y repose en fait sur un jeu de mot. Il faut atteindre des langages comme Lisp ou Ocaml pour voir apparaître les listes comme « citoyens de première classe ».

On a une situation analogue dans le domaine de l'algèbre linéaire sur matrices creuses. Une matrice creuse est représentée par la liste de ses éléments non nuls accompagnés de leurs coordonnées. Mais ces coordonnées ont des propriétés indispensable pour la parallélisation et qui ne figurent pas dans le programme. Par exemple, il n'y a pas de doublons, et les éléments ont été triés, soit par ligne, soit par colonnes.

On pourrait multiplier les exemples de ce type. La question fondamentale est : où trouver l'information manquante ? Une première idée est d'observer une exécution du programme séquentiel. Ceci conduit aux méthodes de parallélisation à l'exécution ou de parallélisation spéculative. Mais l'apprentissage est un exercice difficile, et les résultats de cette approche sont peu probant.

Une autre idée est de demander l'information au programmeur. Mais sous quelle forme ? On a vu apparaître de nombreux systèmes à pragmas (commentaires actifs) où le programmeur signale par exemple les boucles parallèles. Mais si le programmeur ne sait pas ce que c'est qu'une boucle parallèle ? Il s'agit ici d'un problème d'interface homme-machine : comment poser les questions en des termes familiers ?

Il se pourrait que la solution viennent d'une autre discipline, la vérification de programmes. On peut conjecturer que l'information nécéssaire à la parallélisation peut se déduire de la spécification du programme source, ou de la preuve de sa correction. Par exemple, il est difficile de prouver la correction d'un calcul sur matrices creuses sans spécifier la représentation de la matrice, et ceci suffit peut-être pour paralléliser.

1.6. Conclusion

Dans cet exposé, j'ai surtout voulu montrer que la parallélisation automatique, bien que riche d'une longue histoire, est encore un domaine de recherche fructueux, susceptible de nombreuses applications, du grandiose (le calcul à hautes performances) jusqu'au minuscule (les systèmes embarqués). Il est vrai que les problèmes les plus simples sont résolus, et qu'il faut maintenant s'attaquer aux programmes complexes, irréguliers, voire chaotiques. Le domaine n'en devient, à mon avis, que plus intéressant.

1.7. Bibliographie

[ALL 87] ALLEN J. R., KENNEDY K., « Automatic Translation of Fortran Programs to Vector Form », *ACM TOPLAS*, vol. 9, n°4, p. 491–542, octobre 1987.

[AMA 93] AMARASINGHE S. P., ANDERSON J. M., LAM M. S., LIM A. W., « An Overview of a Compiler for Scalable Parallel Machines », *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Springer Verlag, LNCS 768, p. 253–272, août 1993.

[ANC 91] ANCOURT C., IRIGOIN F., « Scanning Polyhedra with DO loops », *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ACM Press, p. 39–50, avril 1991.

[BAN 88] BANERJEE U., *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Boston / Dordrecht / London, 1988.

[BAS 03] BASTOUL C., « Efficient code generation for automatic parallelization and optimization », *ISPDC'03, IEEE International Symposium on Parallel and Distributed Computing*, Ljubljana, p. 23–30, oct 2003.

[BER 66] BERNSTEIN A. J., « Analysis of programs for parallel processing. », *IEEE Trans. on El. Computers*, vol. EC-15, 1966.

[COL 95] COLLARD J.-F., FEAUTRIER P., RISSET T., « Construction of DO loops from systems of affine constraints », *Parallel Programming Letters*, vol. 5, n°3, p. 421–436, 1995.

[COU 78] COUSOT P., HALBWACHS N., « Automatic Discovery of Linear Restraints among Variables of a Program », *Proceedings of the ACM POPL78*, 1978.

[DAR 93] DARTE A., Techniques de parallélisation automatique de nids de boucles, PhD thesis, ENS Lyon, avril 1993.

[DAR 00] DARTE A., ROBERT Y., VIVIEN F., *Scheduling and automatic Parallelization*, Birkhäuser, 2000.

[FEA 91] FEAUTRIER P., « Dataflow Analysis of Scalar and Array References », *Int. J. of Parallel Programming*, vol. 20, n°1, p. 23–53, février 1991.

[FEA 92a] FEAUTRIER P., « Some Efficient Solutions to the Affine Scheduling Problem, I, One Dimensional Time », *Int. J. of Parallel Programming*, vol. 21, n°5, p. 313–348, octobre 1992.

[FEA 92b] FEAUTRIER P., « Some Efficient Solutions to the Affine Scheduling Problem, II, Multidimensional Time », *Int. J. of Parallel Programming*, vol. 21, n°6, p. 389–420, décembre 1992.

[FEA 96] FEAUTRIER P., « Distribution Automatique des Donnés et des calculs », *T.S.I.*, vol. 15, n°5, p. 529–557, 1996.

[GRI 94] GRIEBL M., LENGAUER C., « On the Space-Time Mapping of Nested Loops », *Parallel Processing Letters*, vol. 4, n°3, p. 221–232, septembre 1994.

[IRI 90] IRIGOIN F., JOUVELOT P., TRIOLET R., « Overview of the PIPS Project », FEAUTRIER P., IRIGOIN F., Eds., *Procs of the Int. Workshop on Compiler for Parallel Computers, Paris*, p. 199–212, décembre 1990.

[KEL 92] KELLY W., PUGH W., Generating Schedules and Code within a Unified Reordering Transformation Framework, Rapport n°TR-92-126, Univ. of Maryland, novembre 1992.

[LAM 88] LAM M., « Software Pipelining : An Effective Scheduling Technique for VLIW Machines », *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, Atlanta, p. 318-328, juin 1988.

[PUG 91] PUGH W., « The Omega Test : A Fast and Practical Integer Programming Algorithm for Dependence Analysis », *Supercomputing*, 1991.

[PUG 93] PUGH W., WONNACOTT D., « An evaluation of exact methods for analysis of value-based array data dependences », *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Springer-Verlag LNCS 768, p. 546–566, août 1993.

[QUI 87] QUINTON P., « The systematic design of systolic arrays », FOGELMAN F., ROBERT Y., TSCHUENTE M., Eds., *Automata networks in Computer Science*, Manchester University Press, p. 229–260, décembre 1987.

[QUI 00] QUILLERÉ F., RAJOPADHYE S., WILDE D., « Generation of Efficient Nested Loops from Polyhedra », *International Journal of Parallel Programming*, vol. 28, n°5, p. 469–498, 2000.

[TRI 86] TRIOLET R., IRIGOIN F., FEAUTRIER P., « Automatic Parallelization of FORTRAN Programs in the Presence of Procedure Calls », ROBINET B., WILHELM R., Eds., *ESOP 1986, LNCS 213*, Springer-Verlag, 1986.

[VIV 03] VIVIEN F., « On the Optimality of Feautrier's Scheduling Algorithm », *Concurrency and Computation : Practice and Experience*, vol. 15, n°11–12, p. 1047–1068, septembre 2003.

[XUE 93] XUE J., « An Algorithm to Automate Non-Unimodular Transformations of Loop Nests », *SPDP*, 1993.