Le Modèle Polyédrique

S. Rajopadhye

Nous allons maintenant étudier la construction d'un programme parallèle (selon le modèle PRAM). Plus spécifiquement, nous nous adressons au problème suivant : *étant donné un programme séquentiel, le transformer en programme parallèle.* Ce problème, en toute sa généralité étant extrêmement difficile, on étudie une classe restreinte de programmes séquentiels appelée des *boucles à contrôle affine* (BCA), et ceci pour deux raisons principales :

- Cette classe constitue un enjeu important : la majorité du temps d'exécution de plusieurs programmes est passé dans de tels boucles : non seulement dans des applications numériques (où c'est presque *siné qua non*) mais aussi dans d'autres domaines telle que le traitement d'images.

- Il existe des techniques très performants pour paralléliser cette classe. Issues de la recherche opérationnelle, ces techniques sont connues sous le nom du *modèle polyédrique*, et donnent lieu aux implémentations optimales.

En générale, deux choses empêchent la détection du parallélisme dans un programme séquentiel. La première est la *réutilisation* de mémoire et le problème des fausses dépendances. Le fait que la même casse mémoire est utilisée pour stocker deux valeurs, oblige que *tous* les calculs utilisant la première valeur soient évalués avant la deuxième (bien qu'ils peuvent être indépendants de ce dernier).

La deuxième chose empêchant le parallélisme est la *sérialisation* des réductions (combinaison de plusieurs valeurs avec un opérateur associatif). Le graphe extrait naïvement d'un tel programme séquentiel est une chaîne (disons, de longueur n), et n'admet aucun parallélisme. Or, le fait que l'opérateur soit associatif nous permet de faire le calcul en temps $\lg n$ (sur $n/2$ processeurs) ou bien en temps $\frac{n}{p} + \lg p$ sur p processeurs, qui est de l'ordre $\Theta(\frac{n}{p})$ si $n > p \lg p$. Nous avons alors une implémentation parallèle optimale en travail.

Le modèle polyédrique permet de résoudre ces deux verrous. Tout d'abord, il permet une *analyse exacte de flot de donnés* des BCA, qui permet d'éliminer les fausses dépendances. Le résultat de cette analyse donne une forme intermédiaire du programme qui se formalise en forme de *Systèmes d'Équations Récurrentes Affines* (SERA) dont les variables ont des *domaines polyédriques.* Le modèle polyédrique nous permet de faire une deuxième analyse des SERA pour trouver des réductions dans le programme.

Nous mentionnons ici que le formalisme de SERA est intéressant en soi. Il constitue une classe de programmes équationnels, plus proche aux notions mathématiques à la base des algorithmes, qu'un programme séquentiel. Il est aussi possible d'étendre le formalisme de SERA pour y inclure les opérations de réduction, et de tels programmes sont en principe, encore plus simples et intuitifs à manipuler par des spécialistes d'algorithmique. Parce qu'ils sont équationnels, le développement de ces programmes peut profiter des méthodes formelles telle que la vérification des propriétés, l'interprétation abstraite, etc. Le langage ALPHA développé à l'Irisa souscrit à cette vue, et des outils (le système MMALPHA) pour paralléliser des programmes ALPHA utilise à fond les mêmes techniques que nous allons voir maintenant pour des BCA.

Pour synthétiser un programme parallèle selon le modèle polyédrique on sut des étapes suivantes :

1. Analyser le programme séquentiel pour déterminer le SERA polyédrique correspondant. Dans l'intérêt de breveté, nous ne traitons pas ici la détection des réductions et leur parallélisation mais dirigeons le lecteur intéressé à la discussion bibliographique[1].

2. Un SERA polyédrique nomme, de manière explicite, *chaque valeur* calculée par (i.e., le résultat de *chaque opération* dans) le programme originale. Le deuxième étape consiste alors à déterminer pour le SERA, deux fonctions liées à son implémentation finale :

   - pour chaque opération du SERA, une date d'exécution (i.e., (un ordonnancement) ; et
   - pour chaque opération, une adresse où le résultat sera stockée (une allocation). Cette adresse a deux composants : un processeur, et une adresse locale sur ce processeur. On adopte la convention que le premier composant est *de facto* le processeur responsable pour l'évaluation de l'opération (le "owner-computes rule").

3. On utilise cette information pour construire un SERA équivalente qui respecte les conventions suivantes : certains indices (i.e., dimensions des domaines) de *toutes* les variables du SERA représentent les indices temporels, d'autres représentent les indices de processeurs, et les derniers, l'adresse locale.

4. De ce SERA, on produit du code parallèle qui (i) respecte l'ordonnancement choisi, (ii) ait le parallélisme correspondant aux dimensions de processeurs, et (iii) alloue et utilise la mémoire spécifié.

Nous soulignons une règle d'or que nous souhaitons respecter dans notre démarche. Pendant l'analyse d'un SERA – et, de manière indirecte, son graphe de flot de donnée – il ne faut jamais *construite explicitement* ce graphe. Cette règle a plusieurs motivations

- Ce graphe est trop grand (en comparaison avec la taille du code originale) pour être manipulé facilement par notre compilateur/parallélisateur.

[1] Il est souvent possible que dans un premier temps on évite cette analyse, et d'y revenir que si les prochaines étapes ne donnent pas assez de parallélisme.

- Plus important, le graphe n'est pas (complètement) connu statiquement (au moment de la compilation) dans le cas des programmes paramétrés, et une méthode quelconque qui a besoin de construire le graphe explicitement ne sera pas capable de produire du code qu'après l'instanciation du paramétré de taille !
- Enfin, même si on accepte ces contraintes, le code généré correspondra à un "déroulement" des boucles et aura une taille inacceptablement grande !

L'implication de cette règle d'or est que nous ne pouvons pas utiliser directement les méthodes classiques d'ordonnancement sur multiprocesseurs des graphes de tâches. Il nous faut exploiter la régularité de nos programmes pour travailler sur une représentation réduite. La reste de cette section explique comment le faire à l'aide des polyèdres. Après quelques définitions de base, elle décrit les étapes énoncées ci-dessus, présente des limitations du modèle et des problèmes ouverts, et conclue par des notes bibliographiques.

1 Notations de Base

Dans un programme qui comprisent des boucles à contrôle affine (BCA), la seule construction de contrôle est la boucle `for` dont la borne inférieure (cf. supérieure) est une expression affine (des indices de boucles englobantes et des paramètres) ou le maximum (cf. minimum) de telles expressions. Le corps d'une boucle peut contenir soit une autre boucle, soit une affection aux variables du programme, soit une séquence de ces derniers. Les variables sont des tableaux multidimensionnels, les scalaires étant des tableaux de dimension zéro. La partie droite de l'affectation est une expression arithmétique (que nous supposons élémentaire dans notre analyse) impliquant comme arguments, des variables (tableaux) du programme. L'accès aux éléments d'un tableau (soit dans la partie droite, ou bien dans la partie gauche d'une affectation) se fait par une fonction affine des indices englobants et des paramètres.

Dans une BCA, une affectation S, est exécutée plusieurs fois (une fois par valeur valide des indices englobants). L'ensemble des valeurs valides des indices englobants de S constituent son *domaine d'itération*, D. Une *opération* est alors uniquement identifiée par $\langle S, z\rangle$ où S_i est une affectation, et $z \in D$ est un vecteur entier, appelé le *vecteur d'itération*. On note par $O_1 \ll O_2$, le fait que l'opération O_1 ***précède*** O_2 ***dans l'exécution séquentielle*** de la BCA.

Rappelons quelques définitions concernant les polyèdres. Un ***polyèdre rationnel*** est un ensemble de la forme $\{z \in \mathcal{Q}^n \mid Qz \geq q\}$, où Q (cf. q) est une matrice entière (cf. un vecteur entier). Un ***domaine polyédrique*** (un polyèdre tout court, dans la suite) est l'ensemble de points *entiers* appartenant à un polyèdre rationnel :

$$D \equiv \{z \in \mathcal{Z}^n \mid Qz \geq q\} \tag{1}$$

Une famille ***paramétrée*** de polyèdres corresponde au cas ou le q inclue une fonction linéaire de m *paramètres de taille* p, i.e., $\{z \in \mathcal{Z}^n \mid Qz \geq q - Pp\}$, ou bien

$$\left\{ \begin{pmatrix} z \\ p \end{pmatrix} \in \mathcal{Z}^{n+m} \mid \begin{bmatrix} Q & P \end{bmatrix} \begin{pmatrix} z \\ p \end{pmatrix} \geq q \right\}$$

nous pouvons alors voir un polyèdre n-dimensionnel, paramétré par m paramètres comme un seul polyèdre de $n+m$ dimensions.

Des polyèdres paramétrés admettent aussi une définition équivalente duale en forme de *générateurs* (ses *sommets* par éxemple) qui sont des fonctions *affines par morceaux* des paramètres.

Exemple 1: Voyons $\{i,j,n \mid 0 \leq j \leq i; j \leq n; 1 \leq n\}$ comme un polyèdre 3-dimensionnel. Il a deux sommets : $[0,0,1]$ et $[0,1,1]$. Ses rayons sont donné par l'ensemble $\{[1,0,0],[0,0,1],[0,1,1]\}$. Le même polyèdre, vue comme un polyèdre 2-dimensionnel (paramétré par n) a deux sommets, $\{[0,0],[n,0]\}$ et un rayon, $[1,0]$.

Exemple 2: Considérons $\mathcal{P}_1 = \{i,j,n,m \mid 0 \leq j \leq i \leq n; j \leq m; 1 \leq n,m\}$ comme un polyèdre 2-dimensionnel. En fonction des valeurs relatives de ces deux paramètres, n et m, il est soit un triangle (si $n \leq m$) soit un trapèze (sinon). Il n'a pas de rayons, et l'ensemble de ses sommets est une fonction affine par morceaux des paramètres :

$$\left\{ \begin{array}{lcl} \{1 \leq n \leq m\} & \Rightarrow & \{[0,0],[n,0],[n,n]\} \\ \{1 \leq m < n\} & \Rightarrow & \{[0,0],[n,0],[n,m],[m,m]\} \end{array} \right.$$

On dénote par $z_1 \prec z_2$, l'***ordre lexicographique*** entre deux vecteurs, z_1 et z_2. Notons que c'est un ordre total, et il n'est pas nécessaire que les vecteurs soient de même diménsion (comme les mots d'une dictionnaire, l'origine de cette ordre). On peut voir que l'ordre d'exécution séquentielle est lié à l'ordre lexicographique :

$$\langle S_1, z_1 \rangle \ll \langle S_2, z_2 \rangle \equiv \left\{ \begin{array}{ll} z_1 \preceq z_2 & \text{si } S_1 \text{ apparaisse } \textbf{avant } S_2 \text{ dans le texte de la BCA} \\ z_1 \prec z_2 & \text{sinon} \end{array} \right.$$

Lmax (cf. Lmin) dénote le maximum (cf. minimum) lexicographique de deux (ou plusieurs) vecterus. On remarque que le Lmax (ou le Lmin) de tous les points d'un polyèdre est forcément l'un de ses sommets, et que le Lmax (cf. Lmin) de deux (ou plusieurs) fonctions affines par morceaux est aussi une telle fonction. Alors, le Lmax (cf. Lmin) de tous les points d'un ensemble de polyèdres est une fonction affine par morceaux de leurs paramètres.

Une ***équation récurrente*** (ER) est une équation de la forme suivante, qui définisse une variable X à toutes les points, z, dans un domaine, D^X

$$X(z) = D^X \quad : \quad g(\ldots X(f(z)) \ldots) \tag{2}$$

où

- $f(z)$ est une fonction de dépendances, $f : \mathcal{Z}^n \to \mathcal{Z}^n$;
- g est une fonction élémentaire ;
- les "…" indiquent qu'il peut y avoir d'autres arguments à g.

Nous pouvons avoir plusieurs équations qui définissent X comme ci-dessous :

$$X(z) = \begin{cases} & \vdots \\ D_i & : \quad g_i(\ldots X(f(z))\ldots) \\ & \vdots \end{cases} \tag{3}$$

Chaque ligne est un *cas* de l'équation, et le domaine de X est l'union (disjoint) des domaines des cas $D^X = \bigcup_i D_i$. Les ER peuvent avoir aussi des opérateurs de réduction, mais cet aspect ne sera pas abordé ici.

Une ER est appelée une ***équation récurrente affine*** (ERA) si chaque fonction de dépendance a la forme, $f(z) = Az + Bp + a$, où A (cf. B) est une matrice $n \times n$ (cf. $n \times l$), et a est un n-vecteur.

Un ***système*** d'équations récurrentes (SER) est un ensemble de m telles équations, mutuellement récurrentes. Chacune des variables $X_1 \ldots X_m$, a son propre domaine (D^{X_i} de dimension n_i pour X_i), et les fonctions de dépendance sont en générale de fonctions de $\mathcal{Z}^{n_i} \to \mathcal{Z}^{n_j}$. Dans un systèmes d'ERA (SERA) les fonctions de dépendance sont toutes affines (les matrices A ne sont pas nécessairement carrées)

2 Analyse exacte de flot de données d'un bca

Nous cherchons maintenant, à indentifier les vrais dépendances de notre BCA. Pour cela, nous allons expliciter toutes les opérations du programme et construire un SERA qui ait la même semantique que la BCA original. Pour spécifier *toutes* les opérations, les variables de notre SERA, et leurs domaines, doivent être respectivement, les *affectations* de la BCA, et leurs *domaines d'itération.* Il reste alors à déterminer la partie droite des équations de notre SERA. Il est évident que la fonction g_i n'est rien d'autre que l'expression qui correspond à la partie droite de l'affectation S_i. Mais pour determiner ses arguments, il faut répondre à la question suivante :

Pour une lecture, `X`$[f(z)]$ *d'une variable (tableau)* `Y`*, avec un fonction d'acces f, dans la partie droite de S_i, quelle opération a écrit la valeur lue ?*

La réponse doit forcément être un instance z' d'une affectation, S_j qui a la variable `Y` en partie gauche. Nous considérons de telles affectations un par un. Pour chaque candidat S_j (qui a une fonction d'acces f_j dans sa partie gauche), la solution est l'instance z' qui satisfait (au moins) les conditions suivantes :

$$\left\| \begin{array}{lcl} z & \in & D_i \\ z' & \in & D_j \\ f(z,p) & = & f_i(z',p) \\ \langle S_j, z' \rangle & \ll & \langle S_i, z \rangle \end{array} \right. \tag{4}$$

Les premières deux conditions nous assurent qu'on parle bien des opérations valides, la troisième vérifie que ces opérations accèdent toutes les deux à la même adresse mémoire, et la dernière dit que l'opération $\langle S_j, z' \rangle$ *précède* l'opération $\langle S_i, z \rangle$ dans l'ordre d'exécution du programme originale. Notons, que ces conditions (4) ne sont pas encore assez précises – il y a plusieurs instances valides, z' de S_j qui précèdent $\langle S_i, z\rangle$ et qui ont écrit dans la même adresse mémoire qu'accède cette dernière. Il nous faut trouver le plus récent.

Avant de le fair, remarquons que la dernière contrainte de (4) n'est pas une simple inégalité affine. Néanmoins, elle pourrait être exprimée en forme d'une disjonction (union) d'un nombre fini de telles contraintes. Alors, l'ensemble de solutions que nous sommes amené à considérer est une union de polyèdres. Chaque polyèdre est de dimension $n_i + n_j + m$ (les indices qui y sont mentionnés sont ceux de S_i (i.e., z), ceux de S_j (i.e., z'), et des paramètres du programme). Voyons le plutôt comme un polyèdre de dimension n_j, mais *paramétré* par les $n_i + m$ autres indices. Il est alors claire que l'ensemble de ses sommets est une fonction affine par morceaux de z et p.

Maintenant, revenons au problème de déterminer la *plus récente* écriture parmi les candidats. Pour chaque polyèdre (issus de la décomposition de $\ll$ en disjonction d'inégalités), ceci n'est rien que le point qui est le "plus tard" dans l'ordre d'exécution séquentielle, i.e., son Lmax, qui doit forcément être un de ses sommets – une fonction affine par morceaux de z et p. Parmi les différents polyèdres le plus récent est alors le Lmax de leurs sommets. Enfin, la comparaison de telles solutions pour des différents candidats S_j (rappelons qu'il peut y avoir plusieurs affectations avec la variable `Y` en partie gauche) nous amène à conclure que la source de notre opérande est une fonction affine par morceaux de z et p. Elle pourrait être trouvée de manière automatique, soit avec un outil de manipulation de polyèdres paramétrés, soit un outil de programmation linéaire entiers paramétrée.

3 Ordonnancement selon le modèle polyédrique

Nous allons maintenant résoudre le problème d'ordonnancement : déterminer une date d'exécution pour chaque instance de chaque variable de notre SERA. Nous allons travailler avec un (multi) graphe *réduit* de dépendances (GRD), plutôt que le graphe de flot de données (qui sera appelé graphe de *étendu* de dépendances (GED) dans la suite). Le GRDa un nœud pour chaque variable du SERA, et un arc orienté de S_i à S_j pour chaque instance que S_j apparaisse dans la partie droite de l'équation de S_i. L'arc est étiqueté par la fonction affine de dépendance, et le (sous) domaine de S_i où la dépendance est valide (dû à cause des branches de cas, par exemple). L'arc est alors complètement specifié par un quadruple, $\langle S_i, S_j, D, f\rangle$, et dénote, en effet, plusieurs arcs du GED(un pour chaque point de D).

3.1 Ordonnancement affine unidimensionnel

Notre règle d'or nous oblige également, d'interdire que l'ordonnancement soit spécifié par une *énumération* des dates de chaque opération : il nous faut aussi une représentation compacte de l'ordonnancement. L'instant de temps auquel est exécutée l'opération $O = \langle S, z\rangle$ est noté par une fonction $\tau(O)$. Pour des raisons qui deviendront bientôt claires, nous choisissons τ de la forme suivante :

$$\tau(\langle S_i, z\rangle) = \lambda_i \cdot z + \alpha_i + \lambda'_i \cdot p$$

où λ_i et λ'_i sont des vecteurs constants, et α_i est un scalaire (un tel ordonnancement est dite de *partie linéaire unique*) si $\lambda_i = \lambda_j = \lambda$ pour tout i, j). Notre ordonnancement est complètement spécifié de manière *compacte* : deux vecteurs et un scalaire par variable, indépendamment de la taille des domaines ou les paramètres.

Nous interprétons cette class d'ordonnancements de la manière suivante. Pour un instance de problème, $\alpha_i + \lambda'_i \cdot p$ est fixe, et l'équation $\lambda_i \cdot z = \text{cst}$ défine une famille d'hyperplans dont le vecteur normale est λ_i. Les points sur un hyperplan sont tous executés au même temps en parallèle. Le degrée de parallélisme est $n-1$ (pour un domaine de dimension n, les hyperplans constituent un sous-éspace de dimension $n-1$). Dans le cas d'ordonnancements à partie linéaire unique, il n'y a qu'une famille d'hyperplans. Ils peuvent être visualisés comme une série de "vagues de temps".

Pour que l'ordonnancement soit valide, il doit respecter la causalité des dépendances, i.e., que pour tout arc $\langle S_i, S_j, D, f\rangle$ du GRD, pour tout $z \in D$, $\tau(\langle S_i, z\rangle) > \tau(\langle S_j, f(z)\rangle$, ce qui donne

$$\lambda_i z - \lambda_j f(z) > (\lambda_j - \lambda_i)p + \alpha_j - \alpha_i \tag{5}$$

Ces contraintes (de nombre potentiellement non-borné, car elles doivent être respectées par tout $z \in D$ et pour tous les arcs du GRD) peuvent être simplifiées, grace au modèle polyédrique – les domaines étant des polyèdres, les fontions concernées étant affines, et les polyèdres ayant une représentation compacte. On obtient ainsi un nombre fini des contraintes affines. L'éspace d'ordonnancements valides est alors, lui même un polyèdre (mais avec des coéfficients des λ et des α comme "indices"), non-borné (en fait il est un cône). Nous pouvons encore utiliser des méthodes issues de la recherche opérationnelle pour *choisir* un ordonnancement, typiquement un qui minimise minimise le temps d'exécution.

3.2 Ordonnancement affine multidimensionnel

Il n'est pas toujours possible de trouver un ordonnancement unidimensionnel, i.e., de trouevr une solution au contraintes (5), par exemple dans le cas où le programme n'a pas un degré de parallélisme suffisamment grand. Dans ce cas, nous sommes amenés aux ordonnancements affines multidimensionnels, qui sont une simple généralisation des ordonnancements unidimensionnels, sauf que des λ_i sont des *matrices* $k \times n_i$, des λ'_i sont des *matrices* $k \times mn_i$, et des α_i sont des k*-vecteurs*. Le temps est vue comme un k-vecteur, plutôt qu'un scalaire, et la notion d'ordre totale entre ces "instants de temps" est l'ordre lexicographique. Les contraintes d'ordonnancement restent identique aux (5), sauf qu l'inégalité ($>$) est remplacé par $\succ$, la succession (stricte) dans l'ordre lexicographique.

Nous pouvons visualier ces ordonnancements (toujours dans le cas de partie linéaire unique) avec la même notion de vagues des hyperplans, ou comme (à peu près) la sous division de temps normale en heures, minutes et secondes.

Parce que l'ordre lexicographique peut être exprimé comme la disjonction d'un nombre fini des inégalités affines, nous avons réduit notre problème au quelques instances d'ordonnancement unidimensionnels, et les méthodes se généralisent bien. Pour choisir l'ordonnancement, il a été démontré qu'un "algorithme de glouton" est optimale, dans le sens qu'il trouve de degré maximale de parallélisme.

4 Allocation de processeurs et transformation de sera

Le prochaine étape de notre démarche énnoncée sérait de déterminer une fonction de mémoire pour chaque opération du SERA. Cependant, nous allons pour le moment, supposer que cette fonction est l'identité. Ceci nous permet de voir les détails de la génération de code (d'assignation unique) et d'ensuite revenir sur (i) l'analyse nécessaire pour la réutilisation de mémoire, et (ii) son exploitation pour la génération de code (d'assignation multiple).

Pour générer du code parallèle à partir de notre SERA, il faut d'abord le transformer telle que (i) toutes les variables partagent le même éspace d'indices, et (ii) interpréter certains de ces indices (pour *toutes les variables*) comme le temps. Ceci se fait à l'aide d'une transformation appelé le changement de base (CDB. Considérons le SER :

$$\begin{array}{lcll} X[z] & = & z \in D^X & : \; g_X(X[f_{XX}(z)], Y[f_{XY}(z)], \ldots) \\ Y[z] & = & z \in D^Y & : \; g_Y(X[f_{YX}(z)], Y[f_{YY}(z)], \ldots) \end{array} \tag{6}$$

Nous souhaitons construire un SER équivalent où le domaine de X est transformé par une fonction, $\mathcal{T}$. Ceci est donné par

$$\begin{array}{lcll} X[z] & = & z \in \mathcal{T}(D^X) & : \; g_X(X[\mathcal{T} \circ f_{XX} \circ \mathcal{T}'(z)], Y[f_{XY} \circ \mathcal{T}'(z)], \ldots) \\ Y[z] & = & z \in D^Y & : \; g_Y(X[\mathcal{T} \circ f_{YX}(z)], Y[f_{YY}(z)], \ldots) \end{array} \tag{7}$$

à condition que $\mathcal{T}$ admet un inverse à droite dans le contexte D^X, i.e., il existe une fonction $\mathcal{T}'$ tel que $\forall z \in D^X,\ \mathcal{T} \circ \mathcal{T}'(z) = z$. Il faut souligner que la classe de SERA est clôs sous les CDB par des transformations affines : si $\mathcal{T}$ est une fonction affine, $\mathcal{T}(z) = Tz + t$, le SER qu'on obtient en applicant unt CDB *reste* un SERA.

Pour revenir au problème initiale, nous avons un SERA et un ordonnancement (disons, de dimension k). Pour chaque variable, S_i, nous construisons un CDB $\mathcal{T}$ dont les premières k lignes correspondent à l'ordonnancement : $[\Lambda_i, \Lambda'_i, A_i]$. Les autres lignes peuvent être quelconques à condition qu'il y a le même nombre de lignes pour toutes les $\mathcal{T}_i$. Il est toujours possible d'assurer cette condition, en fait il existe un infinitude de choix possibles. Bien que le choix influence le nombre de processeurs et leur efficacité (à un facteur constant près) et les communications (d'une manière profonde), nous n'allons pas le regarder ici : on cible une machine PRAM.

5 Génération de code en assignation unique

Maintenant, nous avons un SERA dont tous les domaines partagent l'éspace d'indices, et avec une notion unique des indices temporels. Il nous reste de générer du code correspondant à une évaluation du SERA selon l'ordonnancement choisi. Ce code comprise deux parties :

- Un préambule dans lequel nous déclarons la mémoire nécessaire pour stocker nos résultats intermédiares. Parce qu'il s'agit, pour le moment, du code à assignation unique, le préambule consite en une déclaration des tableaux multidimensionnels, identiques aux domaines de nos variables (ou leurs boites rectangulaires englobants).

- Un corps qui comprise un ensemble de boucles qui parcourt l'ensemble de ses tableaux (i.e., l'union des domaines de variables) dans l'ordre lexicographique des premiers k indices (et d'autre dans un ordre quelconque). Les premiers k boucles sont alors des boucles `for` et les autres sont les boucles `forall`. Le problème de génération de ces boucles est résolu grace aux outils de manipulation de polyèdres. Les corps des boucles ne sont rien que les corps des équations correspondants.

6 Reutilisation de mémoire et génération de code en assignation multiple

Nous sommes maintenant au dernière étape de notre parallélisation, celle de la transformation du code d'assignation unique en assignation multiple. En principe l'idée est simple. Nous faisons un analyse pour determiner une fonction d'allocation de mémoire valide, et ensuite modifions systématiquement le code.

Traitons d'abord le deuxième aspect. Si nous avons, pour chaque variable du SERA, une fonction, $\mathcal{M}_i$ qui dit que l'adresse ou $S_i[z]$ est stockée est $\mathcal{M}_i(z)$ (pour le moment supposons qu'un oracle nous assure que c'est une allocation valide). Alors, pour générer du code d'assignation multiple, il nous faut simplement modifier la démarche ci-dessus en :

- Produire un préambule qui alloue des tableaux pour $\mathcal{M}_i(D_i)$ plutôt que pour des D_i.

- Systématiquement remplacer (pour tout i) tous les accès au tableau S_i (qui est forcément de la forme $S_i[f(z)]$) par $S_i[\mathcal{M}_i(f(z))]$.

L'analyse nécessaire pour assurer la validité de ces fonctions $\mathcal{M}_i$ cosniste en effet, de développer des contraintes qui garantie qu'une valeur n'est jamais écrasée avant que toutes les opérations qui l'utilisent sont exécutées. Ces conditions, qui correspondent à une analyse de durée de vie des variables pourrait être réduites en un nombre fini des contraintes affines, grace aux polyèdres. Nous pouvons ainsi développer des allocations mémoire optimales qui minimisent la mémoire pour un ordonnancement donné.

Globalement, notre démarche de parallélisation d'une BCA est le réverse d'une démarche "normale" de la construction manuelle d'un programme parallel. Commencant par un programme qui explicite chaque opération (et alors horriblement naive en son utilisation de mémoire), nous avons choisi un ordonnancement (et alors un degré de parallélisme), et à partir de cet ordonnancement, nous avons repliée la mémoire. La chose importante est que nous avons le fait de manière systématique et modulaire (séparation des concernes), et en plus automatique et optimale !

7 Notes bibliographiques

La section 0.2 est basé sur les travaux de Feautrier [9]. L'ordonnancement a toujours été un sujet de recherche très actif. Dans un papier pionnier [12] Karp, Miller et Winograd ont étudié l'ordonnancement d'une sous-classe (propre) des SERA polyédriques, telle qu'on a défini ici, appelé des systèmes d'équations récurrentes *uniformes*. Ces méthodes ont été adoptées et étendues dans le cadre de la synthèse d'architectures systoliques d'abord à partir des programmes à dépendances uniformes – par Quinton [19], Rao et Kailath [22, 23], et d'autres – ensuite à partir des programmes à dépendances affines, pour lesquels les SERA polyédriques ont été défini et les premiers resultats sur l'ordonnancement "polyédriques" ont été développés par Delosme et Ipsen [8], Rajopadhye et. al [21, 16], Quinton et Van Dongen [20], Yaacoby et Cappello [25] et d'autres. Ces résutats on été étendus pour le cas d'ordonnancement multidimensionnel par Feautrier [10, 11]. Bien que le problème n'a pas été posé de ce façon, l'ordonnancement a été aussi un aspect important dans la parallélisation classique des boucles (voir par exemple, les travaux de Allen et Kennedy [1, 2], de Lamport [14] et de Wolf et Lam[24]. Un livre sur ce sujet ést sortie récemment [6]. Le problème clé de génération de code – comment parcourir des polyèdres à l'aide des boucles `for` – a été resolu par Ancourt et Irigoin [4] en utilisant la méthode d'élimination de Fourier et Motzkin, et a été utilisé dans plusieurs parallélisateurs comme SUIF [3], PIPS de l'École des Mines de Paris, LOOPO de l'Université de Passau, etc. La génération de code à partir d'un SERA nécessite le parcours d'un *union de polyèdres*, comme traité par Kelly et. al [13], et Quilleré et. al [18]. Le problème de la réutilisation de mémoire à été traité par Chamski [5], De Greef et Catthoor [7], Lefebvre et Feautrier [15] et Quilleré et Rajopadhye [17]. Plusieurs bibliothèques de manipulation de polyèdres existent maintenant, notamment Polylib, PIP, Omega, etc.

References

[1] R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In *IBM Conference on Parallel Processing and Scientific Computing*, 1982.

[2] R. Allen and K. Kennedy. Automatic translation of Fortran programs to parallel form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] S. P. Amarasinghe. *Parallelizing Compiler Techniques based on Linear Inequalities.* PhD thesis, Stanford University, Computer Science Department, 1997.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Third Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM SIGPLAN, ACM Press, 1991.

[5] Z. Chamski. Generating memory efficient imperative data structures from systolic programs. Technical Report PI-621, IRISA, Rennes, France, December 1991.

[6] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[7] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. In *Parallel Processing and Multimedia*, Geneva, Switzerland, July 1997.

[8] Jean-Marc Delosme and Ilse C. F. Ipsen. Systolic array synthesis: Computability and time cones. In M. Cosnard, P. Quinton, Y. Robert, and M. Tchuente, editors, *Int. Workshop on Parallel Algorithms and Architectures*, pages 295–312. Elsevier Science, North Holland, April 1986.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb 1991.

[10] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I, one-dimensional time. Technical Report 28, Labaratoire MASI, Institut Blaise Pascal, April 1992.

[11] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, multidimensional time. Technical Report 78, Labaratoire MASI, Institut Blaise Pascal, October 1992.

[12] R. M. Karp, R. E. Miller, and S. V. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[13] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers 95: 5th Symposium on the Frontiers of Massivelly Parallel Computation*, McLean, VA, 1995.

[14] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, pages 83–93, February 1974.

[15] V. Lefebvre and P. Feautrier. Optimizing storage size for static control programs in automatic parallelizers. In Lengauer, Griebl, and Gorlatch, editors, *Euro-Par'97*, volume 1300. Springer-Verlag, 1997.

[16] Christophe Mauras, Patrice Quinton, Sanjay V. Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S. Y. Kung and E. Swartzlander, editors, *International Conference on*

Application Specific Array Processing, pages 100–110, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[17] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 2000. accepted, in press.

[18] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):(in press), October 2000.

[19] Patrice Quinton. *The Systematic Design of Systolic Arrays*, chapter 9, Automata Networks in Computer Science, pages 229–260. Princeton University Press, 1987. Preliminary versions appear as IRISA Tech Reports 193 and 216, 1983, and in the proceedings of the IEEE Symposium on Computer Architecture, 1984.

[20] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[21] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503, New Delhi, India, December 1986. Springer Verlag, LNCS 241. Later appeared in Parallel Computing, June 1990.

[22] Sailesh Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, Information Systems Lab., Stanford, Ca, October 1985.

[23] Sailesh Rao and Thomas Kailath. What is a systolic algorithm. In *Proceedings, Highly Parallel Signal Processing Architectures*, pages 34–48, Los Angeles, Ca, Jan 1986. SPIE.

[24] M. Wolf and M. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.

[25] Yoav Yaacoby and Peter R. Cappello. Scheduling a system of nonsingular affine recurrence equations onto a processor array. *Journal of VLSI Signal Processing*, 1(2):115–125, 1989.