Analyse de la Granularité des Applications Régulières

Jean-Paul Stromboni

Laboratoire I3S, URA 1376 CNRS,
Université de Nice Sophia-Antipolis
Valbonne, FRANCE, FR 06560
e-mail: strombon@essi.fr, Tel. 04 92 96 51 64

RÉSUMÉ

Les applications de taille importante dans le domaine du Traitement du Signal qui sont composées de nids de boucles traitant systématiquement des tableaux de données, présentent en conséquence une structure de dépendance régulière (c'est à dire répétitive). La détection de sous-graphes répétitifs dans le graphe des tâches associé suggère un partitionnement de l'application par agglomération de ces sous-graphes en tâches de grain supérieur. Dans cette optique, un modèle des dépendances des applications régulières est d'abord défini, et des conditions de partitionnement en sont déduites, avec l'effet sur la taille du graphe associé. Ces éléments sont implémentés dans un heuristique capable de suggérer au concepteur d'une l'application des regroupements de tâches adéquats à partir d'une analyse a priori de la structure de l'application.

ABSTRACT

Some huge applications in the field of Signal Processing are mainly composed with sequences of loop nests for array processing, then potentially endowed with a regular structure of dependence. The detection of repetitive subgraphs in the associated task graph could allow to partition efficiently the application by means of period clustering with corresponding granularity adjustment. With this purpose, an adequate model must be defined for the application, the conditions and methods for graph compaction must be derived, with their effect on graph size. From the results, a compile-time heuristic is proposed, that provides advices for task aggregation from a priori program structure analysis.

1 Présentation de l'approche

Le graphe des tâches est un formalisme répandu pour décrire les dépendances des programmes en Informatique et en particulier pour les applications du Traitement du Signal. Arrangé en niveaux, un tel graphe révèle le parallélisme potentiel d'une application et permet à un compilateur de décider de son implémentation sur une architecture parallèle. Cependant, la complexité des décisions de placement et d'ordonnancement croît exponentiellement avec la taille du graphe et selon le niveau de détail adopté pour la programmation. Cela peut poser un problème dans les environnements d'aide à l'implantation pour décrire, représenter les applications massives (par exemple en traitement d'image bas niveau) et proposer une implantation de manière à optimiser l'Adéquation Algorithme Architecture, comme avec SYNDEX [1].

1.1 Partitionnement sans information sur la structure des applications

Pour réduire les dimensions d'un graphe de tâches, des méthodes générales qui n'utilisent pas d'information sur les propriétés structurelles du graphe peuvent être envisagées. Les heuristiques déterministes ou stochastiques de cette catégorie cherchent à minimiser le nombre d'arcs du graphe au moyen de regroupements des sommets. Dans [3] par exemple, ce problème dit *Graph Compaction Problem* est complètement formalisé et plusieurs heuristiques déterministes sont comparés avec une méthode de recuit simulé.

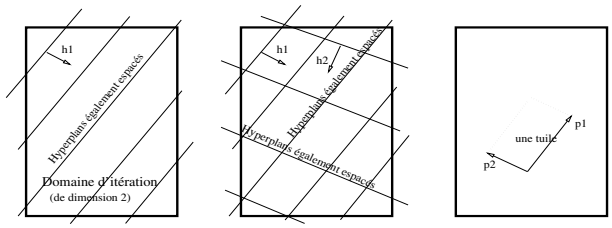

FIG. 1 — Supernodes, tiling

1.2 Découpages du domaine d'itération

Pourtant, les applications même massives peuvent avoir une formulation simple et brève, comme une séquence de nids de boucles (boucles imbriquées), ou un système d'équations récurrentes. Pour l'implantation des systèmes d'équations récurrentes uniformes dans le domaine des algorithmes systoliques [5], on utilise alors la caractéristique d'uniformité des dépendances de données : pour ces applications en effet, les vecteurs de dépendance qui représentent les échanges de données nécessaires entre les instructions du programme se répètent uniformément sur une ou plusieurs dimensions dans le domaine des itérations. Certains auteurs créent des supernodes, comme [2], en découpant ce domaine à l'aide de familles d'hyperplans régulièrement espacés et convenablement orientés (vecteurs h_1 et h_2, illustration Figure 1) en fonction de la structure de l'application. D'autres comme [4] créent des tuiles (tiling) dont ils étudient l'épaisseur et la forme pour optimiser un compromis entre calcul et communication.

1.3 Principe retenu

Le principe retenu ici consiste à rechercher les périodicités de la structure des dépendances de données dès la formulation du programme, à la différence de [3] qui doit considérer l'ensemble du graphe, et à agglomérer ou à superposer les répétitions détectées pour réduire le graphe des tâches et par conséquent la taille de l'architecture cible adaptée. On augmentera en conséquence le grain des tâches et des dépendances. Les dépendances de l'application sont périodiques selon la dimension des données (largeur du graphe) mais a priori non uniformes, à la différence de [2] et [4], bien qu'elles puissent le devenir pour le graphe réduit.

2 Modèle de l'Application

2.1 Définitions et paramètres du problème

Les applications envisagées sont constituées de nids de boucles uniformes opérant sur des tableaux de **données**.
A l'intérieur d'un tel nid, une instruction met en relation un ou plusieurs tableaux opérandes et un tableau résultat. Chaque relation de ce type est une **contrainte de dépendance**. Dans le modèle, tous les **tableaux** multidimensionnels sont linéarisés, et représentés par des vecteurs. Le calcul d'un tableau résultat est constitué de la répétition d'un opérateur, ou **tâche** calculant des sous-tableaux, ou **paquets** de données, à partir de sous-tableaux opérandes, ce qui détermine des **échanges** entre tâches. La **granularité** ou grain d'un paquet est le nombre de données ou valeurs qu'il contient.
Les tâches peuvent ainsi être représentées sur les tableaux par le groupe des valeurs qu'elles engendrent. La **granularité** d'une tâche ou d'un échange est celle du paquet de données calculé, respectivement transmis.
La **règle d'accès** ou d'adressage des données est restreinte à une loi affine d'ordre $N+1$ où N est la profondeur du nid de boucles : Pour chaque tableau opérande ou résultat impliqué dans une instruction, la boucle de niveau i du nid qui l'inclut se voit associer 3 paramètres entiers $n_i, o_i, e_i, i = 1 \cdots N$ pour les accès aux données : le nombre d'itérations, la première valeur de l'indice, et le pas des itérations. On range dans le vecteur d'itération $j = j_1, j_2, \cdots j_N$ les indices des N boucles. Les 3 paramètres supplémentaires n_0, o_0, e_0 décrivent un paquet de n_0 données.
Avec cette règle d'accès, un paquet de valeurs d'un tableau tab linéarisé englobe à l'itération j les n_0 composantes *tab[v]* suivantes :

$$(tab, j) = \{tab[v], v = \sum_{i=0}^{N} (o_i + j_i)e_i, j_0 = 0 \cdots n_0 - 1$$

$$j = j_0, j_1, \cdots j_N \text{ tel que } 0 \leqslant j_i < n_i,\ i = 1 \cdots N$$

2.2 Agglomération et Superposition

On analyse les périodicités créées par la boucle de niveau i d'un nid pour une contrainte de dépendance reliant deux tableaux.

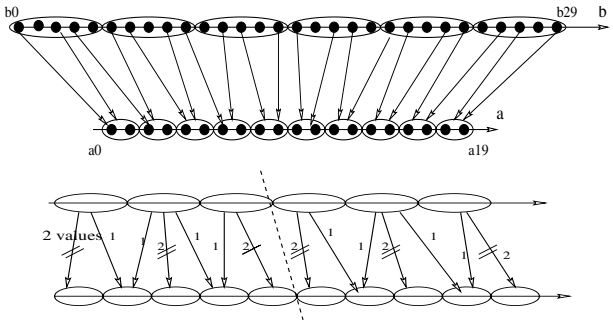

FIG. 2 — Exemple de dépendance périodique

Supposons les opérandes organisés en n_{op} paquets (ou tâches) de grain g_o, les résultats en n_r paquets de grain g_r, c'est à dire que le tableau opérande est calculé par n_{op} tâches calculant chacune g_o valeurs, et le résultat par n_r tâches engendrant g_r données chacune. Notons T_o et T_r les périodes trouvées (en nombre de tâches). Les tableaux opérande et résultat sont décomposables par division euclidienne :

$$n_{op} = T_o \times x_o + m_o, \quad n_r = T_r \times x_r + m_r$$

Si $m_o = m_r = 0$, on peut agglomérer parfaitement les périodes pour obtenir x_0 et x_r tâches de granularités $T_o \times g_o$ et $T_r \times g_r$, ou les superposer pour obtenir T_o et T_r tâches de grains $x_o \times g_o$ et $x_r \times g_r$. On gagne $(n_{op} - x_o) + (n_r - x_r)$ tâches par agglomération ou $(n_{op} - T_o) + (n_r - T_r)$ par superposition.
Ainsi, le graphe de la Figure 2 représente deux instructions `i1` et `i2` d'opérateurs `initb` et `tsk` engendrant les tableaux b[30] puis a[20] fonction de b :

```
i1 : for(j1=0;j1<6;j1++)
       b[j0+5*j1,j0=0..4] = initb();
i2 : for(j1=0;j1<10;j1++)
       a[j0+2*j1,j0=0..1] =
              tsk(b[2*j0+3*j1,j0=0..1]);
```

`a[j0+2*j1,j0=0..1]` représente deux valeurs calculées par l'itération `j1` de l'opérateur `tsk`, 20 valeurs de a sont calculées en tout par groupes de deux à partir de 20 valeurs de b (sur 30 valeurs de b en tout). Les paramètres d'accès des données s'en déduisent :

```
# parametres d'acces : [n1,o1,e1][n0,o0,e0]b
i1 : [6,0,5][5,0,1]b = initb();
i2 : [10,0,2][2,0,1]a =
              tsk([10,0,3][2,0,2]b);
```

On recense 16 sommets, 14 arcs, soit une taille du graphe définie comme la somme $16 + 14 = 30$.
Une régularité apparaît visuellement (cf. pointillé Figure 2), reproduction en double exemplaire d'une relation entre 5 tâches de a (de grain 2) et 3 de b (de grain 5), avec 7 vecteurs de dépendance de grains 1 et 2 (le grain est défini comme le nombre de données impliquées). Une analyse des congruences ci-dessous retrouve cette régularité, et suggère une agglomération en 4 tâches de grains 15 et 10 avec deux dépendances de grain 10 :

$$T_o = \frac{ppcm(e_1, g_o)}{g_o} = \frac{ppcm(3,5)}{5} = 3,$$

$$T_r = \frac{ppcm(e_1, g_o)}{e_1} = 5 \rightarrow x_o = x_r = 2,\ m_o = m_r = 0$$

La Figure 3 représente les graphes obtenus par agglomération et superposition avec ces valeurs. On note la possibilité de fusionner verticalement les tâches après agglomération.
Remarque : la superposition implique une permutation des tâches et des données dans les tableaux, avec une incidence sur l'ordre chronologique des traitements.

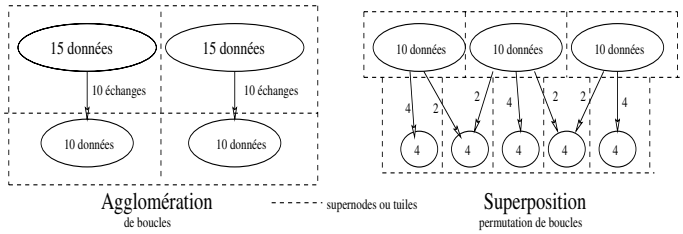

FIG. 3 — Agglomération et Superposition

On étend sans difficulté cette technique aux cas particuliers de la réduction totale où $n_{op} = 1$ et de la diffusion totale où $n_{res} = 1$.

2.3 Restrictions du modèle

Principalement, le modèle ci-dessus ne décrit pas les nids de boucle affines où l'indice des boucles interieures dépend linéairement ou non des indices des boucles englobantes [6]. Egalement, il n'y a pas de structures conditionnelles, les dépendances de données sont statiques, c'est à dire connues à la compilation, seuls les schémas répétitifs périodiques sont considérés.

3 Analyse en boucle ouverte de l'Algorithme R.P.G.E.

3.1 Application étudiée

Afin d'illustrer la méthode, on considère un exemple plus complexe constitué de plusieurs niveaux de précédence inspiré d'une itération de l'algorithme de rétropropagation du gradient (stochastique) de l'erreur (RPGE), pour l'apprentissage d'un réseau neuromimétique à trois couches. En dépit des dimensions faibles du réseau ($4 \times 3 \times 2$) retenues pour la représentation du graphe initial (Figure 4), on recense déjà 19 instructions calculant 19 tableaux ou vecteurs, 89 tâches et 156 échanges, soit une taille de $89 + 156 = 245$, du fait de la granularité fine (unité) des tâches et des échanges. Supposons des seuils fixés à 50 sommets et 100 arcs dans un environnement d'aide à l'implantation pour la taille du graphe : ce graphe trop gros à deux titres sera rejeté, l'analyse des répétitivités de la structure des dépendances peut alors permettre de le réduire jusqu'à une taille acceptable.

3.2 Méthode proposée

L'heuristique suivant construit à partir des considérations précédentes, sera appliqué à l'exemple :

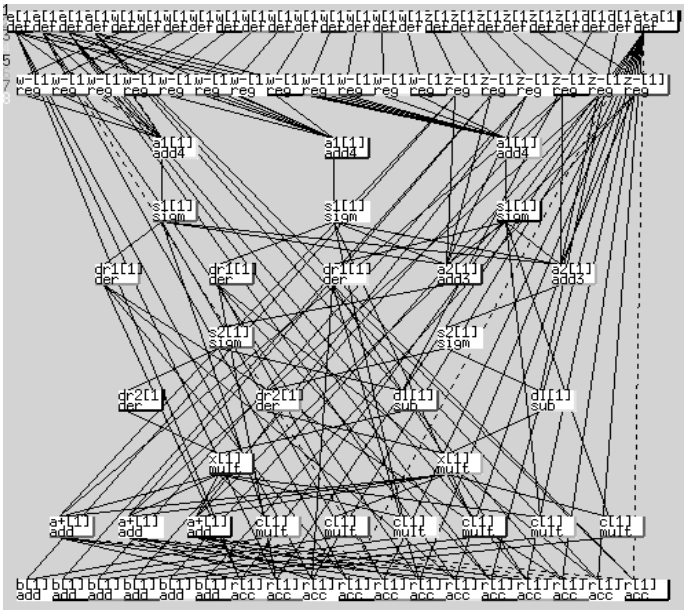

FIG. 4 — Graphe des tâches initial de l'algorithme RPGE

- La période initiale T des entrées est forcée à 1.
- Pour chaque instruction, pour chaque tableau opérande ou résultat, les périodicités T_o et T_r des dépendances sont calculées.
- En cas d'utilisation multiple d'un tableau, on retient le maximum des périodes candidates (ou leur plus petit commun multiple si le nombre de périodes est entier).
- A partir du résultat, on propose pour chaque tableau les agglomérations possibles, soit horizontales (suivant la dimension des données) soit dans certains cas verticales (agglomération de niveaux successifs voir l'exemple de la Figure 3), avec une évaluation du gain résultant. La décision reste en définitive au concepteur de l'application.

3.3 Résultats

3.3.1 Réduction du graphe des tâches

Le graphe des tâches de l'algorithme étudié, organisé en niveaux selon les prédécesseurs, est représenté Figure 4. La largeur mesure le parallélisme potentiel, la hauteur indique la durée d'exécution. Ce graphe peut être compacté (avant même d'être représenté), suivant les directives de l'heuristique et jusqu'à ce que ses dimensions deviennent acceptables.
Sur la Figure 5 par exemple, le graphe des tâches aggloméré ne comporte plus que 15 tableaux, 37 sommets (< 50) et 92 arêtes (< 100). Les granularités des tâches et des dépendances notées entre crochets $[g_i]$ sur la Figure ont augmenté en conséquence, la taille est ramenée de 245 au départ à $37+92 = 129$.

3.3.2 Architecture cible

On a cependant préservé le parallélisme potentiel de l'application, en évitant la solution triviale séquentielle du pro-

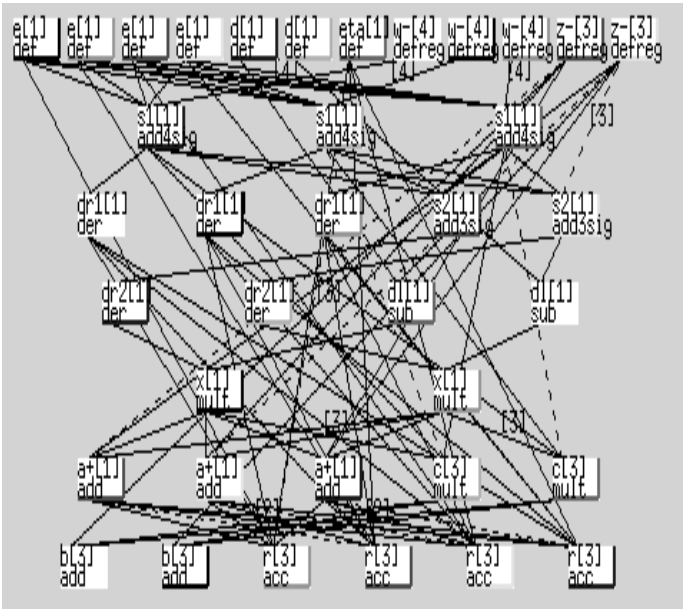

FIG. 5 — Algorithme après agglomération)

blème, qui consiste à réduire à une tâche le calcul de chaque tableau. La présentation en niveaux de précédence de ce graphe compacté suggère pour l'exécution une architecture idéale de 12 sites de traitement, soit la largeur maximale du graphe réduit. Un réseau d'interconnexion minimum adapté au traitement de l'application peut être déduit de la liste des échanges nécessaires en partant d'une architecture entièrement connectée et en supprimant les liens de communications inutiles.
Inversement, dans une variante *en boucle fermée* du procédé de partitionnement non encore implémentée, le grain initial des tâches serait choisi en fonction de la taille donnée pour l' architecture cible, par exemple en répartissant équitablement les données entre les processeurs.

4 Conclusion

L'heuristique proposé pour l'analyse de la granularité, inclus dans un environnement d'aide à l'implantation, permet d'évaluer une technique pour le partitionnement des applications régulières. La méthode suggère des questions théoriques, pratiques, et méthodologiques non encore résolues comme :

- détecter des régularités autres que la périodicité, analyser la dimension temporelle,
- étendre aux équations récurrentes, adapter aux nids de boucles affines,
- interfacer la méthode avec un langage évolué,
- calculer rigoureusement l'effet d'une variation de grain sur la taille et la structure du graphe, et prévoir entre autres la pénalité résultante à l'exécution.
- comment ce type de méthode peut-il être appliqué dynamiquement, instruction par instruction, au cours de l'exécution ?

L'objectif est qu'un compilateur futur puisse sur ces prémisses adopter automatiquement le grain Adéquat compte tenu de l'Architecture cible et de la structure de l'Application à implémenter.

Références

[1] Y. Sorel, C. Lavarenne, "Performance Optimization of Multiprocessor Real Time Applications by Graph Transformations" *Conference ParCo'93*, pp. 89-98, September 1993, Grenoble, FRANCE.

[2] F. Irigoin, R. Triolet "Supernode Partitioning" *Proc. 15th ACM SIGACT-SIGPLAN Principles of Programming Languages*, San Diego, California, January 1988, pp. 319-329

[3] Dino Karabeg "Process Partitioning through Graph Compaction" *J. Parallel and Distributed Computing*, 25, 115-125 (1995)

[4] P. Boulet, A. Darte, T. Risset, Y. Robert "(Pen)-ultimate tiling ?" *INTEGRATION, the VLSI Journal*, 17, 33-51 (1994)

[5] X. Chen, G.M. Megson A General Methodology of Partitioning and Mapping for Given Regular Arrays. IEEE Trans. on Parallel and Distributed Systems, 6(10), 1100–1107, October 1995.

[6] A. Darte, Y. Robert Affine by Statement Scheduling of Uniform and Affine Loop Nests over Parametric Domains. J. Parallel and Distr. Computing, 29, 43–59, 1995