Parallélisation Automatique

Histoire et Perspectives

Paul Feautrier

Paul.Feautrier@inria.fr.

INRIA

Pourquoi le parallélisme

Tous les ordinateurs modernes sont parallèles pour des raisons technologiques :

- Utiliser toutes les portes logiques que l'on peut graver sur un *chip*.
- Masquer la latence d'accès à la mémoire.

Ce parallélisme a longtemps été caché, mais il devient explicite : il n'est plus possible d'alimenter un processeur moderne sans aide du programmeur ou du compilateur.

Types de parallélisme

Type	caché/visible	Extraction
Pipeline d'exécution,	caché	matériel
Superscalaire	caché	matériel + compilateur
VLIW-EPIC	visible	compilateur
vectoriel	visible	compilateur
multithreading	visible	compilateur ou système
grappes	visible	manuel + système
grilles	visible	manuel + système

Avantages et Inconvénients

- Parallélisation matérielle.
  - Le paralléliseur est inclu dans le matériel.
  - Il ne voit à chaque instant qu'une petite fraction du programme et il prend de la place sur le silicium.
- Compilateur Paralléliseur
  - Le paralléliseur est inclu dans le compilateur.
  - Il voit tout le programme, mais ni les données ni les spécifications.
- Parallélisation Manuelle
  - Le programmeur trouve le parallélisme et en informe le matériel à travers un langage (occam, F90, OpenMP) ou un système (Unix) ou des bibliothèques (MPI, PVM).
  - On s'appuie en général sur la structure du problème et on trouve du parallélisme à gros grain.

Ecrire un programme parallèle est difficile

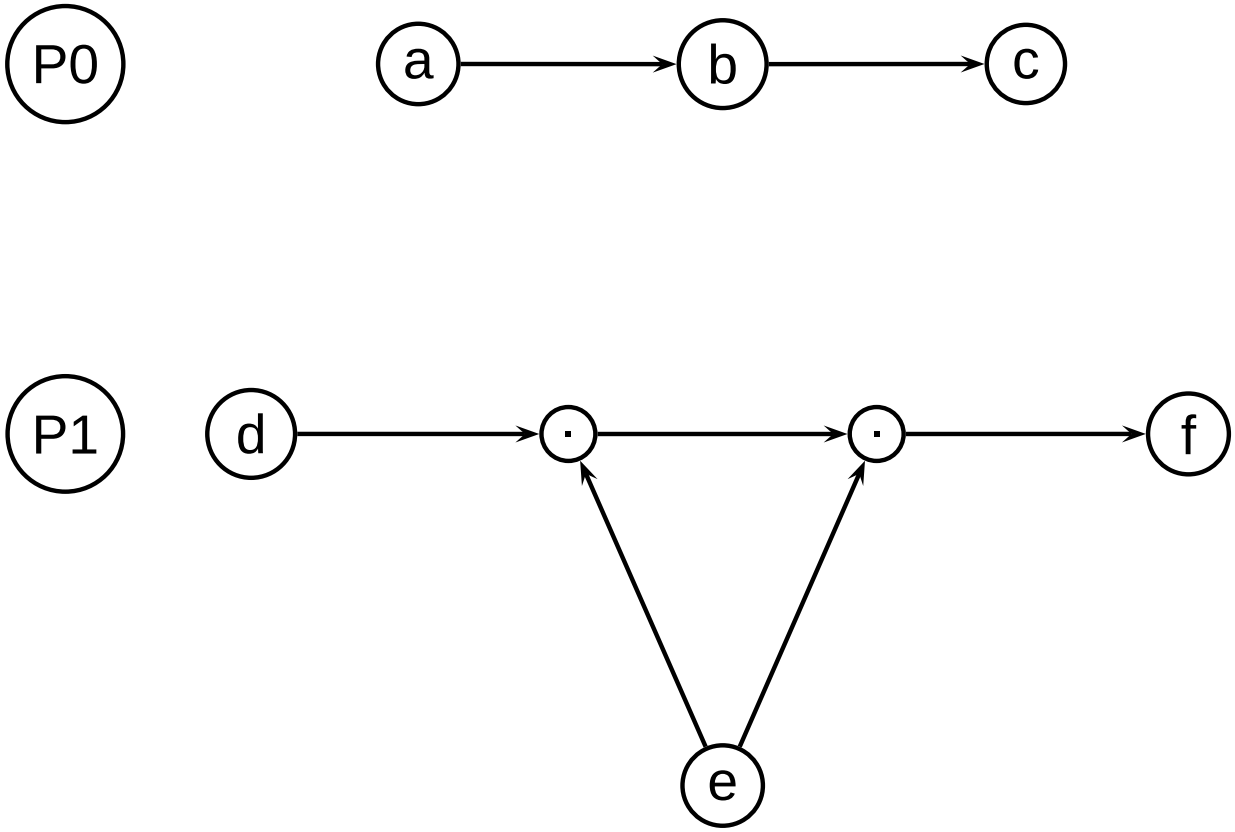
- Il faut trouver le parallélisme ...
- ... mais ne pas en trouver trop :
  - Si on introduit trop de parallélisme, le fonctionnement du programme devient non reproductible!
- On ne peut pas faire l'impasse sur les performances ...
  - ... et les performances dépendent fortement de l'architecture cible;
  - Les programmes parallèles sont peu portables.

Le remède habituel : faire faire le plus possible du travail par l'ordinateur. Jusqu'où peut on aller dans cette direction?

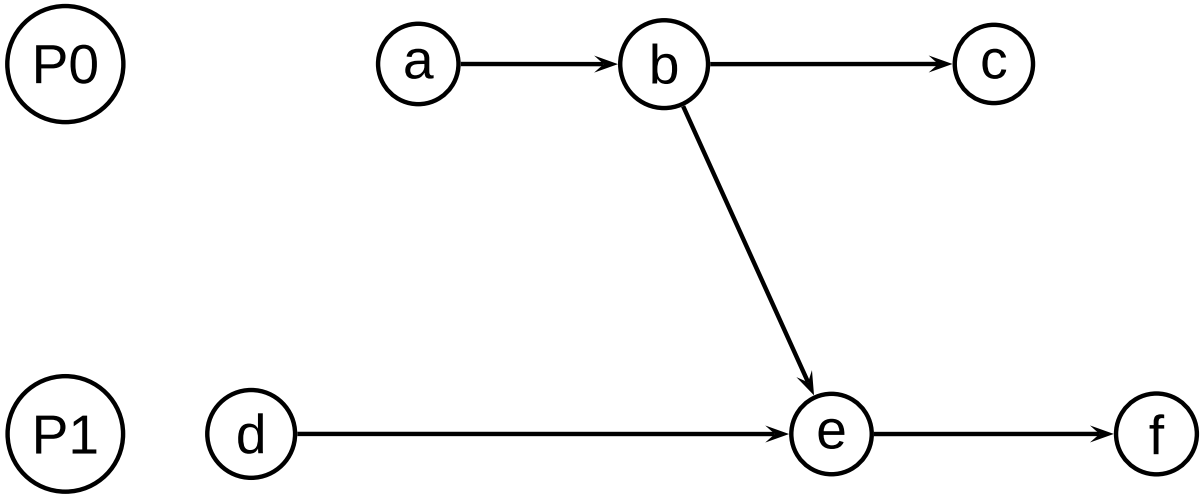
Concepts de base

- Instruction : le choix de ce qu'est une instruction permet le réglage du grain de l'analyse.
- Opération : une itération d'une instruction.
- Ordre d'exécution : u avant v où u et v sont des opérations.
- Caractérisation des ordres d'exécution:
  - Ordre total = programme séquentiel,
  - Ordre partiel = programme paralléle,
  - Ordre vide = programme totalement parallèle.

Indéterminisme



Synchronisation



- Si on veut que le programme soit déterministe, il faut que l'ordre d'exécution de $(b, c), (b, d), (c, e), (c, f), (d, a), (d, b)$ et (d, e) soit indifférent.
- Réciproquement, si on sait quelles sont les opéraions dont l'ordre est indifférent, on peut construire le programme parallèle.

Graphe de dépendance

- Deux opérations u et v sont en dépendance si $u;v \neq v;u$. On écrit $u \perp v$.
- Les sommets du GD sont les opérations;
- Il y a un arc de u vers v si u est exécuté avant v dans le programme séquentiel et si $u \perp v$.
- Le programme dont l'ordre d'exécution est la fermeture transitive du GD est équivalent au programme initial, et il peut être parallèle.
- Problèmes:
  - Il est impossible de dessiner le GD.
  - Comment synthétiser le programme parallèle spécifié par le GD?

Histoire

Condition de Bernstein (1966)

- Soit $M(u)$ l'ensemble des cellules de mémoire modifiées par u,
- soit $R(u)$ l'ensemble des cellules de mémoire lues par u.
- Si u est exécutée avant v et si:

$$M(u) \cap M(v) = \emptyset,$$
$$M(u) \cap R(v) = \emptyset,$$
$$R(u) \cup M(v) = \emptyset,$$

  alors u et v ne sont pas en dépendance.
- Si l'une des conditions ci-dessus n'est pas remplie on a respectivement une dépendance de sortie (PP ou WAW), une dépendance de flot (PC ou RAW), une anti-dépendance (CP ou WAR).

Calcul des Dépendances

- Le calcul des ensembles lus et modifiés et celui des intersections est trivial pour un programme sans calcul d'adresses.
- Le problème n'est pas encore bien résolu si le calcul d'adresse se fait par *pointeur*.
- Si le calcul est une indexation, il faut comparer les valeurs courantes des indices. On suppose que le programme est correct, c'est à dire qu'il n'y a jamais de débordement d'indice.
- Dans ces conditions: $a[e_1, \ldots, e_n] \equiv a'[e'_1, \ldots, e'_m]$ si et seulement si;

$$a \equiv a',\ n = m,\ e_k = e'_k,\ k = 1, n$$

Recherche des boucles parallèles

- On cherche à prouver qu'aucun couple d'opérations de la boucle n'est en dépendance.
- Ceci revient à construire un GD *résumé*, où toutes les opérations instances d'une même instruction sont regroupées en une seule.
- Pour savoir s'il existe une dépendance, il faut discuter les solutions d'un système de contraintes que l'on espère affines.
- Dans un nid de boucles, le concept de profondeur permet de distinguer les boucles parallèles et les boucles séquentielles.

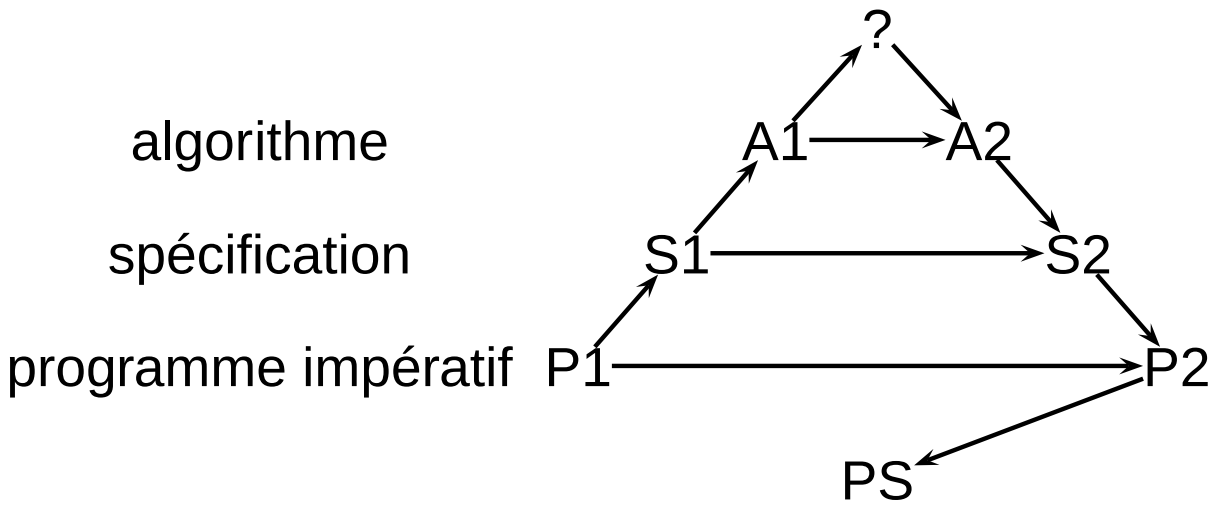
Tests de dépendance

- Le *Principe de pessimisme* permet de construire des test approchés.
- Test du PGCD: identifier les cas ou il n'y a pas de solution en nombres entiers.
- Tests de Banerjee : traiter les équations une par une et en variables continues.
- Lambda test, Power test, etc...
- Test de Fourier Motzkin, Simplexe : méthodes exactes en variables continues.
- Omega test, coupes de Gomory : méthodes exactes en variables entières.

A la vitesse des ordinateurs actuels, la performance du test de dépendance n'est plus un problème.

Transformations de programme I

- Le calcul des dépendances permet de trouver le *parallélisme syntaxique* d' un programme.
- Il est fréquent que l'on ne trouve rien par cette méthode, mais:
- On peut transformer le programme initial en un programme équivalent qui a plus de parallélisme syntaxique.



Transformations de programme II

Esquisse de classification.

- Optimisation du programme séquentiel:
  - Elimination des `gotos`.
  - Elimination des calculs redondants.
- Changement de l'ordre des calculs.
  - Distribution de boucles.
  - Inversion de boucles.
  - Le test de validité se ramène à un test de dépendance.
- Changement des structures de données.
  - Expansion de scalaire ou de tableau.
  - Permet de trouver plus de parallélisme en supprimant les dépendances de sortie.
  - Nécéssite l'analyse du flot de données.

L’algorithme de Allen et Kennedy

- Il y a beaucoup de transformations, elles peuvent être appliquées en de nombreux sites, elles interagissent de façon compliquée : problème de combinatoire.
- D’où la recherche d’algorithmes incorporant une ou plusieurs transformations plus la recherche du parallélisme syntaxique.
- L’algorithme de Allen et Kennedy utilise la distributions et l’inversion de boucles.
  - Point de départ : le GD résumé + les profondeurs de dépendance.
  - Boucles = composantes fortement connexes (cfc) du GD.
  - Ordre des boucles = Graphe Quotient.
  - Boucle séquentielle si la cfc contient un arc de profondeur 0.
  - Boucles internes = application récursive de l’algorithme.
  - Il est toujours possible de faire “descendre” les boucles parallèles.
- Le résultat est un programme vectoriel.

Le Modèle Polyédrique

Polyèdres

- Un compilateur paralléliseur (ou un optimiseur) doit manipuler des ensembles:
  - Ensemble des opérations, ensembles des opérations qui écrivent dans une méoire donnée, ensemble des cases d'un tableau qui ont été modifiées, etc.
  - Un compilateur ordinaire ne manipule que des arbres.
- Ces ensembles sont trop complexes pour être manipulés en extension.

```
for(i=0; i<n; i++)
  for(j=i+1; i<n; i++)
    a[i,j]=...;
```

$S = \{i, j | 0 \leq i < n, i + 1 \leq j < n\}$.

- Si le programme est à controle statique, ces ensembles sont l'ensemble des solutions entières d'un système de contraintes affines ou Z-polyèdre. Les algorithmes de manipulation des Z-polyèdres viennent de la Recherche Opérationnelle et sont bien développés.

Ordonnancement

- Principe: on attribue à chaque opération une date de lancement. Toutes les opérations qui ont la même date de lancement sont exécutées en parallèle.

$$u\delta v \Rightarrow \theta(u) + \partial(u) \leq \theta(v).$$

- $\theta(u)$ fonction affine des compte-tours des boucles englobant u.
- La fonction affine $\theta(v) - \theta(u) - \partial(u)$ doit être non-négative dans le polyèdre $u\delta v$.
- Résolution par le lemme de Farkas ou la méthode des sommets.
- Tous les algorithmes de parallélisation connus (y compris Kennedy et Allen) sont des variantes de l'algorithme d'ordonnancement obtenues en « agrandissant » le polyèdre des dépendances [Darte et Vivien 1999]

Placement

- Principe: on attribue à chaque opération un « processeur virtuel ». Toutes les opérations assignées au même processeur sont éxécutées en séquence.
- De même, chaque donnée est attribuée à (la mémoire d') un processeur. Pour éviter des communications, chaque calcul doit être effectué par le processeur qui en détient les données.

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

- Système d'équations linéaires et homogènes à résoudre par une méthode de Gauss incrémentale.
- Certaines contraintes ne peuvent pas être satisfaites et engendrent des communications (synchronisations) résiduelles.

Génération de code

- Les algorithmes d'ordonnancement et de placement ne donnent pas directement le programme parallèle.
- Exemple : programme ordonnancé:

  for $t = 0, L$
  forall $\{u|\theta(u) = t\}$
  do u

- Il faut parcourir dans un ordre quelconque le polyèdre $\{u|\theta(u) = t\}$ (le front à la date t).
- On détermine les bornes inférieures et supérieures de chaque boucle par des méthodes de programmation linéaire [Ancourt et Irigoin, Darte, Collard et Risset, Pugh et. al., Lam et. al., Lengauer et. al., Xue, Quilleré].

Autres applications

- Analyse du flot des données.
  - Trouver la plus récente écriture dans une cellule de mémoire donnée (ou encore, la *source* de la valeur contenue dans la cellule de mémoire).
  - Problème de maximisation paramétrique. Les paramètres sont l'instant où on veut calculer la source et les indices de la cellule (si elle appartient à un tableau).
- Comptages des points entiers contenus dans un polyèdre [Ehrhardt, Clauss].
  - Le nombre de points entiers d'un polyèdre peut représenter un nombre d'opérations, la taille d'un tableau ou d'une section de tableau.
  - Application en équilibrage de charge, gestion de la mémoire, gestion du cache.
  - Le nombre de points est la valeur d'un polynome dont la forme est donnée par des théorèmes d'Ehrhardt complétés par Clauss, Wilde, Rajopadhye, Loechner.
  - Les coefficients du polynomes s'obtiennent par interpolation.

Problèmes émergents

Que peut apporter l'interprétation abstraite?

- Faire de l'interprétation abstraite, c'est « simuler » un programme :
  - On remplace les calculs réels par des calculs approximatifs,
- Intérêt : permet de trouver des approximations quand le calcul exact est impossible.
- Calcul des ensembles lus et modifiés :
  - On remplace les expressions d'indice par de nouvelles variables.
  - On détermine les relations de ces variables avec les autres variables du programme [Cousot Halbwach].
- Calcul direct des dépendances : aucun résultat à ce jour.
- Analyse du flot des données :
  - Cas scalaire : analyse des *use-def chains*. Pas de transposition connue au cas des tableaux.
  - Analyse des régions [Creusillet].

Et les algorithmes de placement?

- Les algorithmes de parallélisation sont en général des algorithmes d'ordonnancement ...
- mais les algorithmes de placement sont aussi des algorithmes de parallélisation.
  - Ils sont plus simples et moins complexes (pas de calcul de dépendance, Gaus vs. Simplexe).
  - Ils optimisent naturellement la localité.
- Problèmes :
  - Génération du code.
  - Gestion de la mémoire.
  - Optimisation des communications ou des synchronisations.

Parallélisation Modulaire

- La parallélisation modulaire n'est pas la même chose que la parallélisation interprocédurale.
  - En compilation séparée, le problème est de savoir ce qu'il faut stocker « à coté » d'un module pour pouvoir compiler le reste du programme.
  - En général, on note, pour chaque fonction, son type et le type de ses arguments. C'est insuffisant pour paralléliser.
- Que faut-il stocker pour poursuivre la parallélisation?
  - Les ensembles lus et modifiés [Triolet, Irigoin]? Cela suppose une exécution atomique du module et demande une estimation de sa durée d'exécution.
  - Les contraintes d'ordonnancement visibles de l'extérieur? Permet une parallélisation simultanée de l'intérieur et de l'extérieur du module. Est-ce possible dans le cadre du modèle polyédrique?

Ordonnancement sous contraintes de ressourc

- Ressources : tout ce qui peut venir à manquer : opérateurs, registres, mémoire, interconnexions, etc.
- Cas d'école :
  - chaque opération occupe une certaine ressource (e.g. un processeur) pendant un certain temps (e.g. un cycle). Il y a P processeurs. Trouver un ordonnancement vérifiant la contrainte:

$$\forall t \ \ |\{u|\theta(u) = t\}| \leq P.$$

- Etat de l'art :
  - Le problème est résolu pour le cas unidimensionnel (une seule boucle) parce que le calcul du cardinal est faisable :*software pipelining*.
  - dans le cas multidimensionnel (nid de boucle) il existe plusieurs heuristiques :
    - Ignorer les contraintes de ressource puis « replier » l'ordonnancement;
    - Simuler les contraintes de ressources par des contraintes de données.
- Existe-t-il une solution directe?

Existe-t-il d’autres modèles de programme?

- Représentation des ensembles d’opérations et des structures de données.
- Algorithmes effectifs d’union, intersection, complémentation, projection, etc.
- Proposition : langages réguliers (= automates finis)
  - doivent être étendus (= transducteurs) pour représenter des fonctions.
  - l’intersection devient indécidable.
  - ne permettent de représenter que les arbres.
- Y a-t-il d’autres modèles? Peut-on étendre les méthodes d’ordonnancement ou de placement?

La structure absente

- De nombreux programmes ne contiennent pas l'information nécéssaire à leur parallélisation.
  - Comment reconnaitre une liste?
  - Comment paralléliser un calcul sur matrices creuses?
  - Comment paralléliser un *branch-and-bound* ou un algorithme itératif convergent?
- Observer une exécution du programme?
- Demander l'information absente au programmeur? Dans quel langage?
- Partir d'une spécification ou d'une preuve du programme?

Conclusion

Les problèmes liés au parallélisme ne vont pas disparaître,
bien au contraire,

Retroussons nos manches ....