Parallélisation Automatique

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

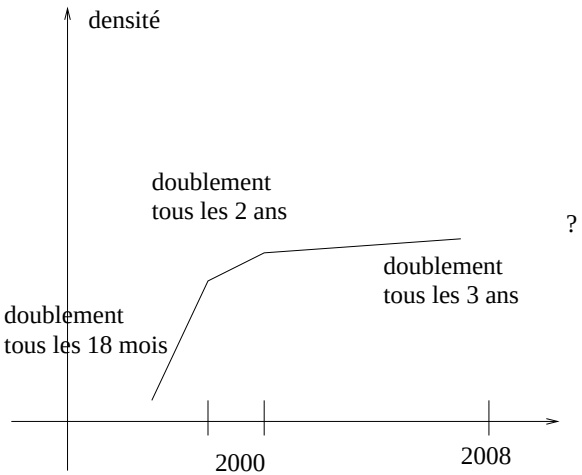
8 septembre 2008

Pourquoi la parallélisation automatique ?

- Les gains de performances dus à la technologie s'amenuisent pour des raisons physiques (effets quantiques, dissipation)
- Les applications nouvelles (3D et 4D, analyse et reconnaissance, sémantique) demandent de plus en plus de puissance
- Une seule solution, le parallélisme
- Pour se faciliter la vie : écrire un compilateur qui traduit un programme séquentiel en un programme parallèle.

Evolution de la technologie, I

La loi de Moore n'est plus ce qu'elle était :



- La finesse de gravure se rapproche dangeureusement des tailles atomiques. Des effets quantiques et de courants de fuite apparaissent. La densité ne va plus augmenter à la même vitesse
- La distribution d'horloge sur fils conducteurs devient de plus en plus difficle (biais) et de plus en plus gourmande en énergie (30% de la consommation sur un processeur moderne). La fréquence d'horloge plafonne vers 3 Ghz.

Evolution de la technologie, II

- Par contre, il n'y a pas d'obstacle *théorique* à augmenter la taille du chip. Que faire de ces portes supplémentaires ?
- Réaliser un processeur plus complexe, exploitant le parallélisme implicite. Il semble qu'une limite soit atteinte (diminishing return)
- Implanter plusieurs processeurs simples sur une seule puce. Excellent pour la consommation électrique, mais difficile à programmer
- Ajouter des co-processeurs, parce que les processeurs généralistes sont peu efficaces (exemple : processeurs pour le traitement du signal, pour la cryptographie).

Evolution des Applications

- Plus de puissance ⇒ applications plus complexes.
- Course à la performance (simulation physique, mathématique, sociale, sémantique, reconnaissance de la parole, des images ...)
- Généralisation des systèmes embarqués : ordinateurs pas ou peu programmables, cachés à l'intérieur d'un produit grand public :
  - minimiser la taille du programme et des données
  - minimiser la consommation électrique (durée de vie des batteries)
  - algorithmes figés, donc possibilité d'utiliser des accélérateurs matériels et des méthodes de compilation coûteuses.

Revue d'architecture

Critères de classification :

- Classification de Flynn : SISD, SIMD, MISD (n'existe pas), MIMD
- Ma classification : deux critères
  - une horloge / plusieurs horloges, synchrone / asynchrone, déterministe / non déterministe
  - mémoire globale / mémoire distribuée
- la difficulté de programmation croit de "gauche à droite" ;
- la difficulté de réalisation **décroit** de gauche à droite !

Une horloge, une mémoire

- Processeur séquentiel
  - Exploitation du parallélisme niveau instruction (ILP)
  - Goulot d'étranglement pour l'accès à la mémoire (*memory bottleneck*), caches multiples.
- Processeur VLIW
- Processeur vectoriel
- Accélérateurs matériels (ASIC, FPGA)

Une horloge, plusieurs mémoires

- *Single Instruction, Multiple Data, SIMD* Plusieurs processeurs appliquent la même instruction à des données différentes
- Chaque processeur possède sa propre mémoire
- Le fonctionnement est synchrone
- Il faut prévoir un réseau de communication et un ordinateur "hôte".
- Les applications doivent être très régulières (algèbre linéaire, traitement d'image)
- Le nombre de processeurs n'est pas en général égal à la taille de l'objet, d'où l'introduction du concept de *processeur virtuel*, qui a donné ensuite naissance aux *threads*.

Plusieurs horloges, une mémoire

- SMP (*Symetric MultiProcessor*) : plusieurs processeurs sur une même carte mère
- Multicore : plusieurs processeurs sur une seule puce
- SMT (*Simultaneous Multi threading*) : un seul processeur capable de passer d'un thread à un autre à chaque défaut de cache (typiquement en un cycle).

Problème essentiel, la bande passante mémoire :

- Chaque processeur a un cache indépendant
- Problème de la cohérence
- Le nombre de processeurs est limité à quelques dizaines
- Il est cependant possible d'aller très au delà (SGI Origin : 700 processeurs), à l'aide de multiples niveaux de caches et de la mémoire virtuelle distribuée (SDM).

Plusieurs horloges, plusieurs mémoires

- Ordinateurs à mémoire distribuée. Processeurs complets interconnectés par un réseau. Exemple : un ensemble de PC connectés par Ethernet.
  - Performances limitées par le débit du réseau : réalisation de réseaux plus performants qu'Ethernet : Myrinet, Quadrics, SCI ...
  - Une (ou deux) tâche par processeur échangeant des messages avec les autres tâches. Assure à la fois les échanges de données et les synchronisation. Les échanges doivent être aussi rare que possible.
  - Evolution vers les Grilles de Calcul.
  - Il existe des bibliothèques d'échange de messages (PVM, MPI, logiciels pour la grille).

Circuits spécialisés, I

- Il arrive que l'on puisse intégrer toute une application sur un seul chip (exemple : un modem). Comme le nombre de transistors augmente, il est même possible de réaliser un système multiprocesseur sur un chip (SoC) parfois muni d'un réseau (NoC)
- Il est plus fréquent que l'on isole le "noyau" d'une application et que l'on réalise un accélérateur couplé plus ou moins étroitement à un processeur généraliste
- Exemple bien connu : la carte graphique
- Autres exemples : transformateur de Fourier, analyseur de séquences génétiques, etc
- Ces circuits se connectent en général au bus PCI, qui est souvent le goulot d'étranglement de la configuration. (Une carte graphique se connecte sur un bus plus proche du processeur)

Circuits spécialisés, II

Deux grand types :

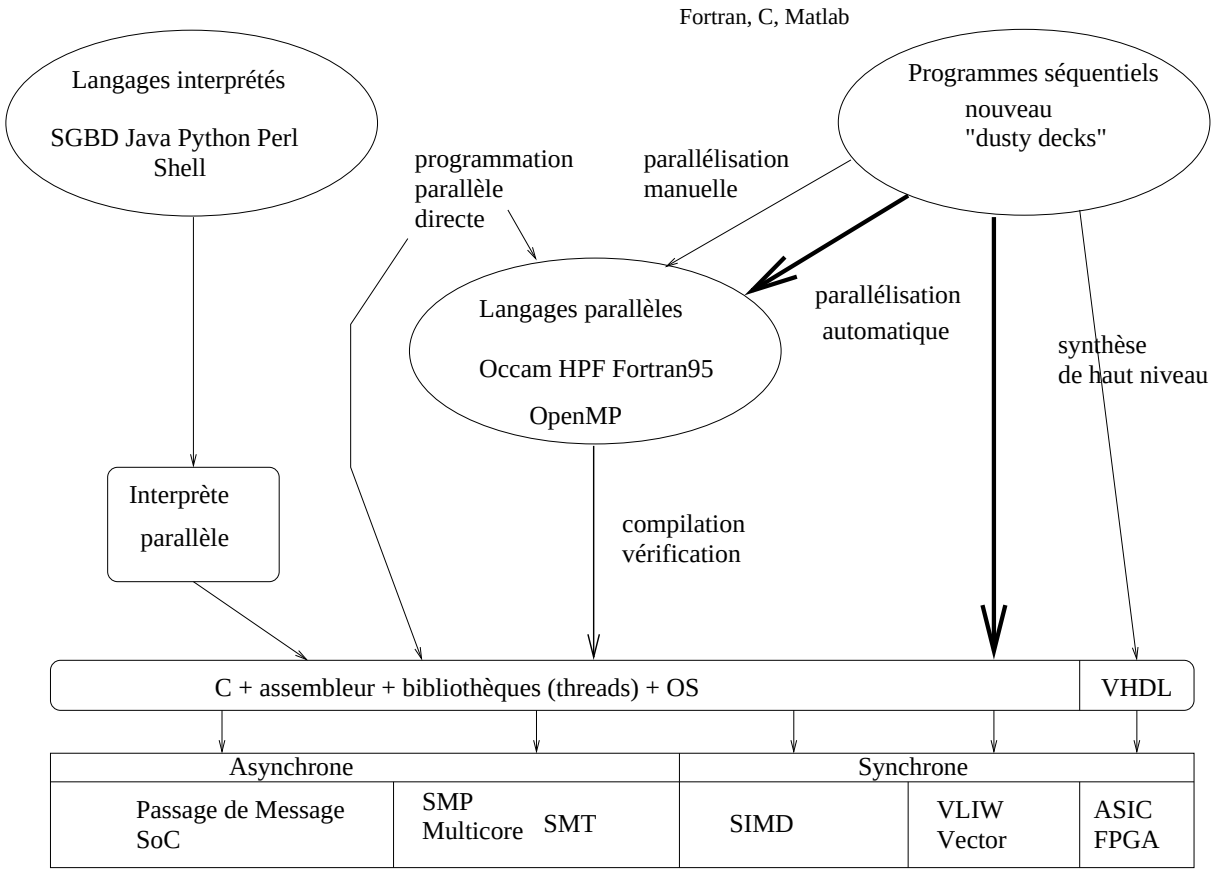
- Circuits classiques : structure analogue à celle d'un processeur :
  - Un *chemin de données* où les calculs sont effectués : registres, accès à la mémoire, opérateurs arithmétiques et logiques, et les connexions nécessaires.
  - Un contrôleur, qui est un automate fini, qui assure l'enchaînement des opérations, configure le chemin de données et interprète les signaux d'état.
- Circuits systoliques : un grand nombre d'opérateurs identiques interconnectés régulièrement par une grille 1D ou 2D.
  - Les opérateurs peuvent contenir des données propres et recevoir des signaux de contrôle.
  - Les applications doivent être très régulières.
  - Avantage : il existe des méthodes de synthèse automatique.
  - Exemple : le filtrage.

Circuits spécialisés, III

Quelques problèmes :

- La taille du circuit : le prix du circuit augmente avec sa taille. Ajouter de la mémoire ou augmenter le parallélisme fait augmenter la taille, mais améliore les performance. Il y a un compromis à trouver.
- La prédictabilité : beaucoup de circuits spécialisés travaillent en temps réél – téléphonie, télévision, pilotage, etc. Certains dispositifs ont des comportements difficiles à prédire (caches, réseaux).
  - Remplacer les caches par des mémoires "scratchpad".
  - Utiliser des réseaux à garantie de service, ou ordonnancer les communications.
- la consommation électrique : parce que les système embarqués sont souvents autonomes. Le parallélisme réduit la consommation, la localité réduit la consommation, etc.

La programmation parallèle
Vue générale



Programmation parallèle, I

- Programmation bas niveau : C (ou assembleur) + bibliothèque parallèle (ou système d'exploitation). Exemples :
  - C + primitives Unix (fork/wait)
  - threads POSIX, threads Java
  - C + MPI
- Un langage interprété peut avoir un interprète parallèle : Shell, SGBD, etc
- Il existe (ou il existera) des logiciels parallèlisés (jeux, tableurs, etc)
- Il existe de nombreux langages parallèles, en général extensions de langages séquentiels : Fortan 95 (vecteurs), HPF (Fortran + distribution) OpenMP (Fortran ou C + boucles parallèles), ADA.

Programmation parallèle
Où se place la parallélisation automatique ?

- Un nouvel algorithme est en général mis au point en séquentiel (exemple de l'utilisation de Matlab en traitement du signal)
- Il faut ensuite le paralléliser, soit à la main, soit automatiquement
- Les compilateurs pour langage parallèles utilisent des techniques issues de la parallélisation automatique
- En l'état de la recherche, ce n'est possible que si le programme est régulier (voir plus loin)
- Directions de recherche :
  - Augmenter la puissance des paralléliseurs automatiques
  - Concevoir des compilateurs hybrides, qui font le maximum, et s'en remettent au programmeur dans les cas difficiles.

Une étude de cas : le produit matrice vecteur

Il existe de multiples façons d'écrire le produit matrice * vecteur.
Version naïve :

```
for(i=0; i<n; i++){
  s = 0.0;
  for(j=0; j<n; j++)
    s += a[i][j]*b[j];
  c[i] = s;
}
```

Il n'y a pas de parallélisme, parce qu'il n'y a qu'un seul accumulateur, s.
Mais on peut accumuler directement dans c.

Produit matrice vecteur, II

```
for(i=0; i<n; i++){
  c[i] = 0.0;
  for(j=0; j<n; j++)
    c[i] += a[i][j]*b[j];
}
```

- Les deux programmes sont-ils équivalents ? Pourquoi ?
- La boucle sur `i` est parallèle. Pourquoi ?

Produit matrice vecteur, III

- On peut faire toutes les initialisations "en bloc"
- On reconnait une opération vectorielle.

```
c[0..n-1] = 0.0;
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    c[i] += a[i][j]*b[j];
```

Produit matrice vecteur, IV

- On devine que l'on a le droit d'échanger les deux boucles
- On reconnait une autre opération vectorielle

```
c[0..n-1] = 0.0.
for(j=0; j<n; j++)
  c[0..n-1] +=  a[0..n-1][j]*b[j];
```

De quel droit ?

Pour que tout ceci soit légitime, il faut que le programme final donne les mêmes résultats (au bit près) que le programme initial. Il faut **justifier** la parallélisation (ici, la vectorisation). Deux idées :

- Procéder étape par étape
- Mettre les preuves "en facteur"

Pour l'exemple :

- Expansion de scalaire : montrer que le flot des données n'est pas modifié
- Modification de la structure des boucles : montrer que l'histoire des valeurs successives prise par l'un des `c[i]` ne change pas
- Le passage au vectoriel n'est que de la reconnaissance de forme.

Conclusion provisoire

- La parallélisation d'un programme se fait par succession de transformations.
- Avant d'appliquer une transformation, il faut savoir si elle est légale ; c'est l'analyse de programme qui permet de répondre à la question.
- L'information essentielle est la description du *flot des données* : quelle est la source de chaque valeur utilisée dans les calculs.
- Du flot des données on déduit les contraintes d'ordre d'exécution.
- La reconnaissance de forme (boucles parallèles, opérations vectorielles, détection de récurrences) joue un role important
- La programmation parallèle est un mal nécessaire
- La parallélisation automatique peut simplifier le travail, au moins partiellement.

Plan du Cours

- Introduction
- La géométrie des programmes
- Tests de dépendance
- Ordonnancement
- Techniques élémentaires de parallélisation
- Génération du code parallèle
- Au delà des programmes réguliers