Le parallélisme : pourquoi et comment?

Jérôme Lelong

Ensimag

Année 2013-2014

Organisation du cours

- **Horaires** : le lundi matin 8h15 — 11h45.
  En général un CM puis un TP.
- **Evaluation** :
  - Mini-projet sur les 2 dernières séances de TP
  - Contrôle écrit

  Note finale = (Projet + contrôle écrit) / 2.
- Quid des TPs...

Contexte

- Réduire le temps à attendre pour obtenir le résultat
- Résoudre des problèmes de plus grande taille
- Mutualiser les ressources de plusieurs machines : plus de mémoire et plus de puissance
- Tirer profits des architectures multi–cœurs
- Mieux utiliser les machines de bureau largement sous exploitées : les plusieurs PC reliés en réseau (très utilisé dans les banques)
- Effectuer un nombre de calculs toujours plus importants en un temps fixé : calculs d'exposition au risque (batchs nocturnes), valorisation de produits très sophistiqués (dérivés de crédit par par exemple)
- Les limites du séquentiel
  - La vitesse d'accès à la mémoire (optimisations de code possibles)
  - Economiquement plus viable de multiplier le nombre de coeurs que d'augmenter la fréquence (problème de surchauffe et miniaturisation)

Objectifs

- Se familiariser avec les différents concepts.
- Comprendre l'intérêt des concepts vus en cours pour les applications pratiques.
- Mettre en œuvre les différents concepts sur des exemples simples et en comprendre les difficultés.
- Réaliser un pricer parallèle.

Plan du cours

- Les cours magistraux
  - Pourquoi le parallélisme ? Les différentes architectures.
  - OpenMP
  - OpenMPI : messages simples
  - Les générateurs de nombres aléatoires
  - OpenMPI : manipulation de structures complexes
- Les TP
  - Prise en main de la libraire *PNL* pour la partie calculs mathématiques.
  - D'abord des exemples simples : le Monte Carlo européen (OpenMP puis OpenMPI)
  - Transporter des objets complexes (matrices, vecteurs, structures quelconques) avec OpenMPI
  - Un pricer parallèle

Les différents types de parallélisme (classification de Flynn)

- architecture séquentielle : SISD (Single Instruction Single Data)
- parallélisme de données : réaliser les mêmes opérations sur des données différentes (machines vectorielles, GPU). Fonctionnement de type SIMD (Single Instruction Multiple Data).
  Exemple type : calcul matriciel
  $\rightsquigarrow$ diviser les données en blocs
- parallélisme d'instructions : réaliser des opérations différentes sur des données différentes (plusieurs processeurs indépendants).
  Fonctionnement de type MIMD (Multiple Instructions Multiple Data)
  $\rightsquigarrow$ diviser pour régner, résolution de plusieurs sous problèmes.

Pourquoi le parallélisme est-il inéluctable ?

- Plafonnement de la vitesse des processeurs
  - Loi de Moore (1965) et limites physiques : ce n'est pas la puissance mais la densité de transistors qui double tous les 18 mois.
  - Patterson (2007) : si la fréquence continue à augmenter, on ne pourra plus allimenter tout ce que l'on sait mettre sur une puce.
- Comment améliorer les performances :
  - hausse de la fréquence $\Longrightarrow$ hausse consommation et température
  - ré-ordonnancement : plus de concurrence dans l'ordre des instructions.
  - caches plus grands et plus rapides

  Les 2 premiers points s'épuisent.
- Plus de cœurs = gain d'énergie
- La mémoire est un goulet

La mémoire est un goulet

- Les accès mémoire limitent souvent les performances **Example** : $a[i] = b[i] + c[i]$. Les 3 accès prennent plus de temps que l'addition
  - 2 lectures et une écriture
  - l'addition se fait en un cycle
  - Core 2 duo P8800 (FSB=1033 Mhz, CPU=2.66 Ghz) $\rightsquigarrow$ 2.5 cycles de CPU pour transférer 64 bits.
- localité de la mémoire : prises en compte des caches, contiguité des données à accéder, ...

Caches et multi–cœurs

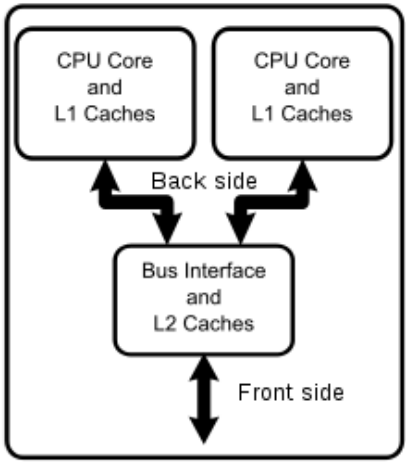


FIGURE: partage des caches (source : Wikipédia)

Caches et multi–cœurs

- Avantages
  - identique à un cache simple
  - pas de problème liée à la cohérence
- Inconvénients
  - bande passante limitée
  - taille réduite
  - maintien en cohérence ?

Les architectures I

- Les machines mutli-coeurs : mémoire partagée (centralisée). Toutes les tâches ont la même image de la mémoire
  - mauvais passage à l'échelle
  - programmation assez facile (OpenMP)
  - pas de problèmes de transmission de messages mais des accès concurrents
- Les clusters de PC : mémoire distribuée. Chaque tâche a sa mémoire locale. L'accès à la mémoire d'une autre tâche passe par un accès réseau
  - bon passage à l'échelle
  - programmation délicate.
  - pas d'accès concurrents mais besoin d'un protocole de communication.
  - rapport calcul/communication : choix du grain de parallélisation ?

Les architectures II

- clusters de machine multi-coeurs : parallélisation à 2 niveaux avec un premier niveau en mémoire distribuée puis un second en mémoire partagée, typiquement au sein de chaque noeud.
  - Programmation délicate avec deux niveaux de parallélisme
  - L'optimisation est délicate
  - Les gains potentiels de performance sont meilleurs
- Nouvelles architectures de type grille : unités de calculs hétérogènes connectées sur un réseau classique.

Mémoire partagée

- Même espace d'adressage global pour toutes les tâches mais une mémoire locale propre
- Chaque processeur/coeur travaille indépendamment des autres mais les modifications faites par l'un sont immédiatement visibles par tous les autres.
  $\rightsquigarrow$ protection des accès mémoire
- 2 types de mémoire partagée :
  - Uniform Memory Access (UNA). Même temps d'accès mémoire pour tous les processeurs : Symmetric MultiProcessor (SMP).
  - Non Uniform Memory Access (NUNA). Conçues pour pallier les accès concurrents via un bus unique. Tous les processeurs peuvent accéder à toute la mémoire mais pas au même coût. Présence de mémoire cache locale.

Mémoire distribuée

- Chaque processeur a sa propre mémoire locale, pas d'adressage globale.
- Chaque processeur travaille indépendamment des autres et ne voit pas les modifications faites par les autres.
- L'accès à de la mémoire d'un autre processeur se fait par la mise en oeuvre explicite d'une communication
  $\rightsquigarrow$ temps d'accès réseau pour la mémoire non locale.
- A priori, peu adaptée aux problèmes mathématiques non locaux

Choix du modèle de parallélisme

- Plusieurs paramètres influencent ce choix :
  - Flexibilité : support de différentes architectures.
  - Efficacité : bon passage à l'échelle.
  - Complexité : contenir le coût de maintenance du code.
- Il faut analyser les éléments suivants :
  - Regroupement de données : indépendance ou dépendance temporelle.
  - Ordonnancement : identifier quelles tâches créent quelles données et quelles données sont requises par quelles tâches.
- Partage des données :
  - Quelles données sont partagées entre plusieurs tâches.
  - Garantir l'accès aux données.

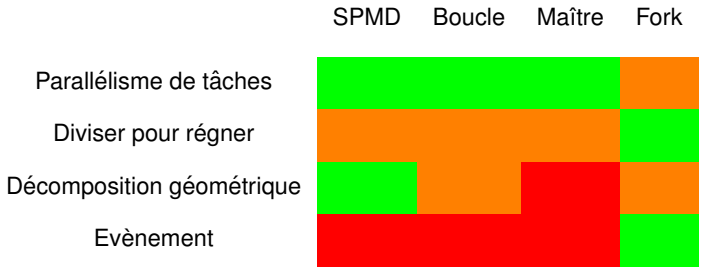
Décomposition du code

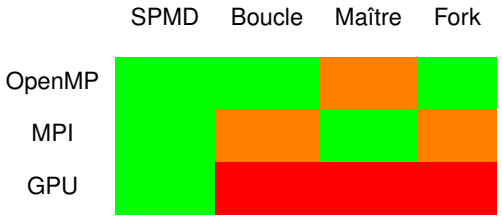
- Identifier les éléments qui autorisent un traitement parallèle.
- Comment décomposer le code pour obtenir le meilleur degré de parallélisme ?
- Parallélisme de tâches : le code séquentiel se découpe en plusieurs sous-tâches qui peuvent être exécutées indépendamment.
- Parallélisme de données : le problème peut se découper en plusieurs problèmes sur des sous-parties indépendantes des données.
- Parallélisme de flux : les données arrivent de manière séquentielle. Principe du travail à la chaine : pipeline.

Les modèles de parallélisme

- SPMD (Programme unique, données multiples)
- Maître/Esclave
- Manager/Travailleur
- Fork/Join
- Boucle
- Flux

Modèles et algorithmes





Et la finance dans tout ça ? I

Méthodes Monte–Carlo Européen. $\mathbb{E}(f(X)) \approx \frac{1}{n}\sum_{i=1}^{n} f(X_i)$.

- Vitesse de convergence de l'ordre de $\sqrt{n}\,\mathrm{Var}(f(X))$.
- Une bonne précision implique n grand
  $\rightsquigarrow$ temps de calcul très long.
- Pas de lien entre les itérations, facile à mettre en œuvre sur une architecture parallèle
  $\rightsquigarrow$ besoin de générateurs aléatoires parallèle,
  i.e. chaque processus doit pouvoir générer une suite de nombres statiquement indépendants de celles utilisées par les autres processus.

Et la finance dans tout ça ? II

Valorisation d'un gros portefeuille de produits financiers : cas typique de l'évaluation des risques imposée aux établissements financiers par les règles de Bâle

- Les calculs sont indépendants :
  $\rightsquigarrow$ une valorisation se déroule sur un seul processeur, pas de communication entre les processeurs.
- Complexité des calculs très inégale : bcp de calculs très courts et peu de calculs vraiment coûteux
  $\rightsquigarrow$ répartition des calculs difficile : engorgement rapide du maître dans une stratégie maître/esclave.
- Besoin de pouvoir estimer la complexité d'un calcul pour que le calcul parallèle soit efficace :
  $\rightsquigarrow$ Minimiser le nombre de communications (il est plus rapide d'envoyer un *gros* paquet que d'envoyer une *multitude de petits* paquets).

Et la finance dans tout ça ? III

Les problèmes délicats.

- Pricing d'options américaines : résolution de l'équation de programmation dynamique

$$\begin{cases} u_T &= \phi(S_T) \\ u_t &= \max(\mathbb{E}(u_{t+1}|\mathcal{F}_t), \phi(S_t)) \end{cases}$$

- Résolution d'EDP : typiquement un algorithme itératif en temps

$$(\partial_t + \mathcal{L})u = f$$

- Possibilité d'utiliser localement un code parallèle : recours à des librairies spécifiques (*Scalapack*, *FFTW*, ...)

Mesure de performances I

- Un calcul séquentiel se décompose en 3 phases : une phase d'initialisation, une phase de calcul, une phase de finalisation

$$T_{total} = T_{init} + T_{calcul} + T_{final}$$

- Si le calcul se réalise sur P processeurs / cœurs

$$T_{total}(P) = T_{init} + \frac{T_{calcul}}{P} + T_{final}$$

Deux notions sont essentielles pour mesurer la performance d'un code parallèle

Definition

Le speedup

$$S(P) = \frac{T_{total}}{T_{total}(P)}$$

L'efficacité

$$E(P) = \frac{S(P)}{P}$$

Mesure de performances II

- En règle général, on a toujours $S(P) \leq P$ et $E(P) \leq 1$.
- Dans de rares cas, on peut légèrement dépasser ces bornes supérieures mais il faut alors soigneusement argumenter les résultats obtenus. Une explication possible :
  - changement de hiérarchie mémoire.
  - taille des caches mémoire : toutes les données peuvent rester en cache.

Loi de Amdhal I

- Les parties du code qui ne peuvent s'exécuter simultanément représentent une certaine fraction du temps total d'exécution

$$\gamma = \frac{T_{init} + T_{final}}{T_{total}(1)}$$

Definition (loi de Amdhal)

$$S(P) = \frac{T_{total}(1)}{\left(\gamma + \frac{1-\gamma}{P}\right) T_{total}(1)} = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

- C'est un speedup idéal qui ne prend pas en compte les aléas de programmation : gestion des threads, étranglement.
- Cette loi suppose qu'il n'y a pas de surcoût lié au parallélisme

Loi de Amdhal II

- $1 - \gamma$ représente la part de code parfaitement parallélisable
- Le borne supérieure du speedup est donnée par

$$S(P) < \frac{1}{\gamma}$$

- A charge constante, il est inutile de continuer à augmenter le nombre de processeurs.

Loi de Gufstason

- On normalise γ par rapport au nombre d'unités de calcul utilisées

$$\gamma(P) = \frac{T_{init} + T_{final}}{T_{total}(P)}$$

Definition (loi de Gufstason)

$$S(P) = P(1 - \gamma(P)) + \gamma(P)$$

- Cette version permet d'étudier l'influence du nombre d'unités de calcul sur le speedup.