Modèles et techniques en programmation parallèle hybride et multi-cœurs

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

December 24, 2017

Plan (2 premières séances)

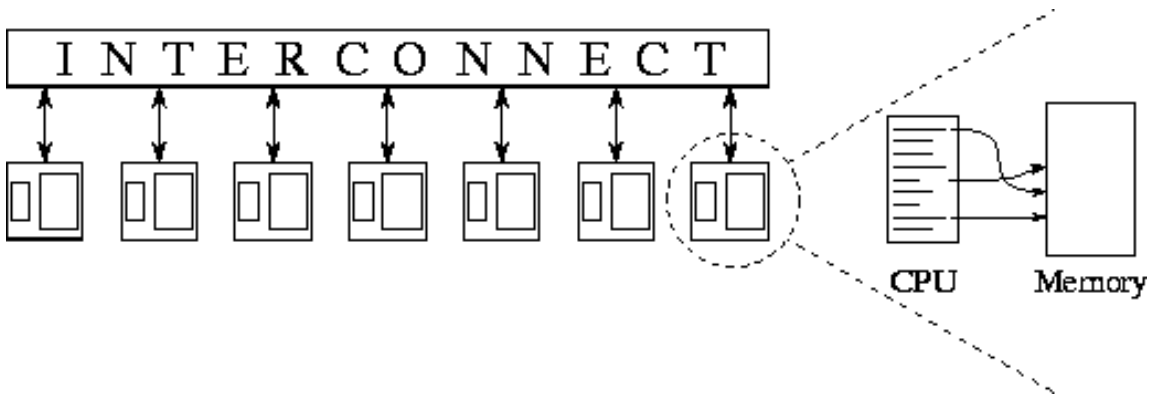
Rappels sur l'architecture matérielle

Programmation séquentielle efficace

Rappels sur l'architecture matérielle

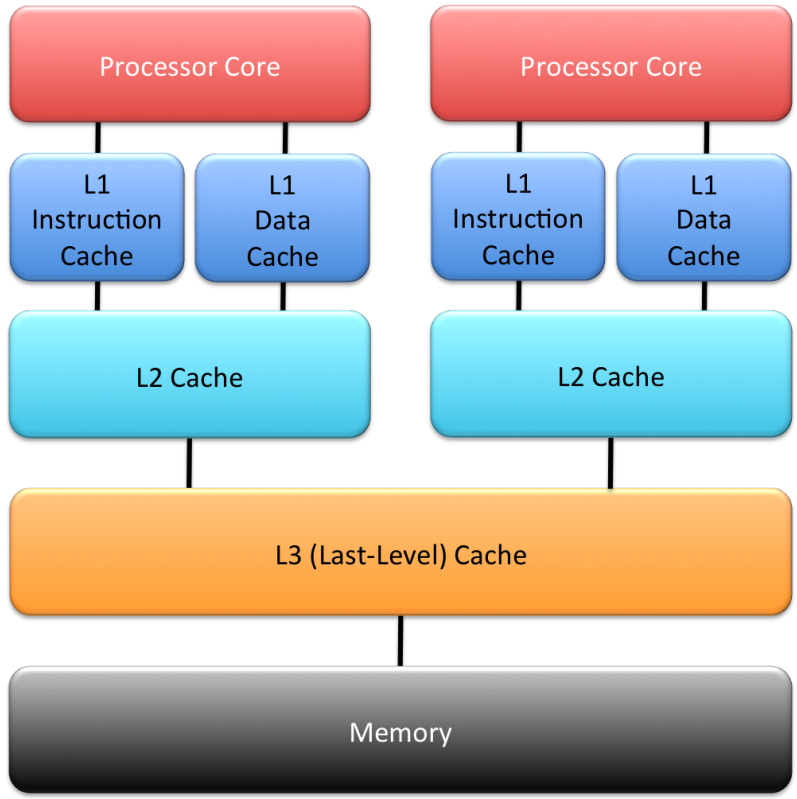
On peut voir la structure d'une machine de calcul à différentes échelles:

Vue globale: un ensemble de nœuds de calcul chacun contenant un ou plusieurs processeurs et de la mémoire, les nœuds sont connectés par un réseau:

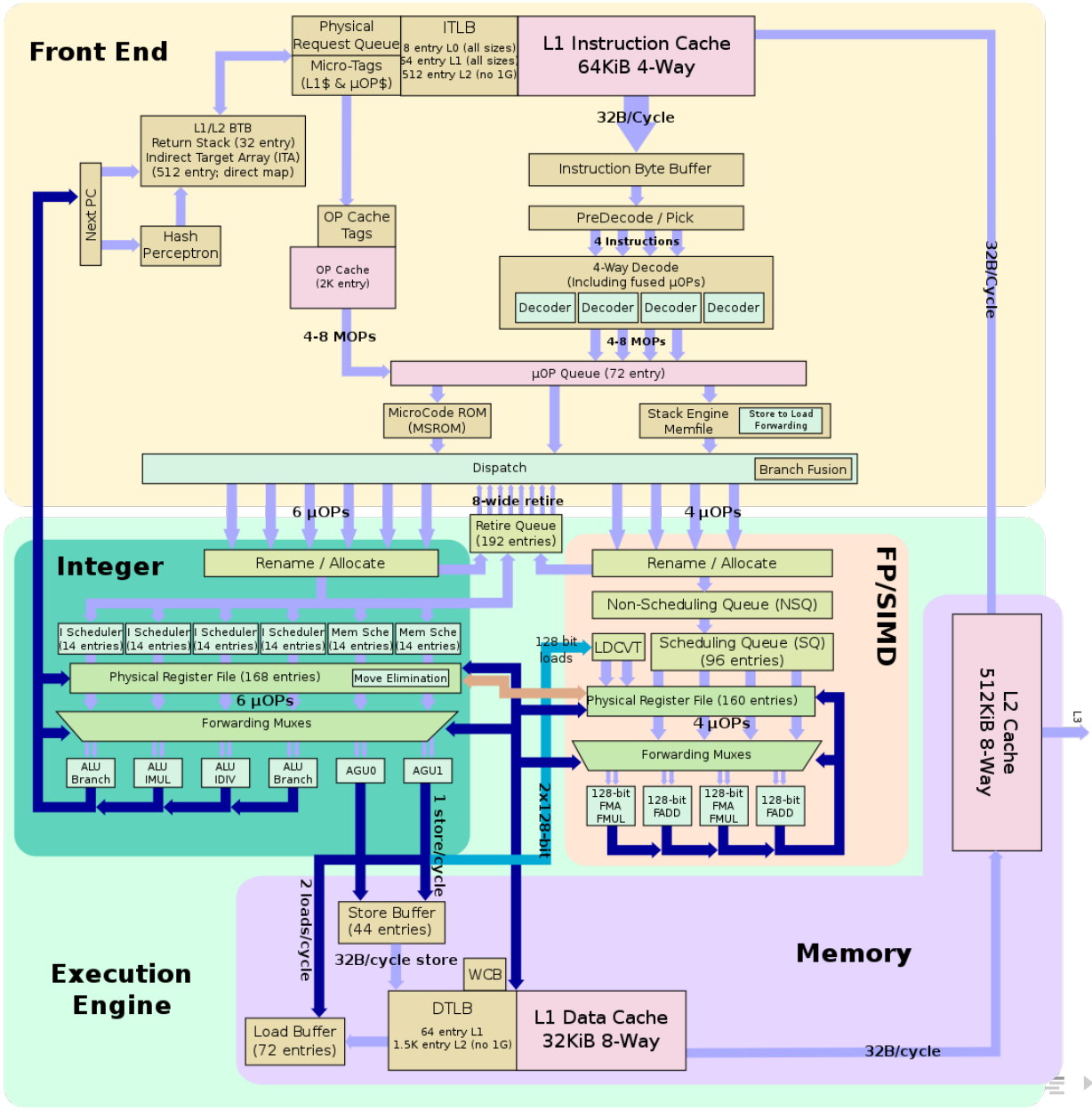


(les machines les plus puissantes actuellement contiennent plusieurs centaines de milliers de nœuds)

Vue interne d'un nœud (variable suivant le modèle de processeur et la génération utilisée):



Vue interne d'un cœur (variable suivant le modèle de processeur et la génération utilisée):



Situation actuelle (et encore pour plusieurs années):

vitesse de calcul (d'un processeur)

- $\approx$ *vitesse de la mémoire interne (registres) du processeur*
- $>$ *vitesse des différentes mémoires cache, intermédiaires entre le processeur et la mémoire centrale du nœud (vitesse L1 $>$ L2 $>$ L3, taille L1 $<$ L2 $<$ L3)*
- $\gg$ *vitesse de la mémoire centrale d'un nœud*
- $\gg$ *vitesse du réseau qui connecte les nœuds*

Fonctionnement :

- Les données sont envoyées dans les registres du processeur qui va les utiliser, en laissant des copies dans les différents niveaux de cache de ce processeur.

- Les zones mémoires sont copiées par bloc entre la mémoire centrale et les mémoires cache. Donc si une donnée est proche d'une autre qui vient d'être utilisée, elle a peut-être déjà été copiée (et l'utilisation de cette 2nde donnée est plus rapide)

- Si une donnée est utilisée n fois par le même processeur dans des instructions proches, les $n-1$ dernières utilisations seront plus rapides (parce que la donnée sera peut-être encore dans une mémoire cache).
- Si un processeur a modifié une donnée dans une de ses mémoires, il faut répercuter cette modification dans les différentes copies de cette donnée.

Cette gestion peut représenter une partie importante du temps d'exécution (pour assurer la cohérence des différentes parties de la mémoire et leurs mises à jour correctes).

Pour obtenir un code efficace (en temps d'exécution), il faut:

- utiliser les algorithmes les plus efficaces possible (pas couvert par ce cours)
- organiser le placement des données (améliorer la localité spatiale)
- organiser la séquence d'instructions (améliorer la localité temporelle)
- écrire les instructions pour qu'elles soient les plus rapides (utilisation du // interne des processeurs, éviter si possible les tests)

Programmation séquentielle efficace

Localité spatiale

Règle: autant que possible, utiliser des zones mémoires proches les unes des autres dans une séquence d'instructions

Le transfert entre mémoire centrale et mémoire cache se fait par bloc.
Donc, si une donnée est á côté d'une donnée qui vient d'être utilisée (et donc transférée en mémoire cache), un nouveau transfert sera peut-être inutile.

Exemple: voir TP 1

Localité temporelle

Règle: autant que possible, pour une zone mémoire, les instructions qui l'utilisent doivent s'exécuter de façon rapprochée dans le temps

La mémoire cache étant de petite taille, le gestionnaire mémoire, s'il a besoin de place, effacera dans la mémoire cache les données les plus anciennes.
Si une donnée est utilisée par plusieurs instructions proches dans le temps, elle sera maintenue plus longtemps en mémoire cache (et donc nécessitera moins de transferts)

Exemple: voir TP 1

Utilisation du // interne au processeur

Règles:

- essayer de rassembler plusieurs instructions simples en une seule (quand cela a un sens),

  utiliser au maximum la présence de plusieurs unités de calcul (additionneurs, multiplicateurs, etc) dans un cœur

- essayer d'éviter les tests

  simplifier le travail du processeur (enchaînement des instructions le plus déterministe possible).

Exemple: remplacer

```
for (i=0; i<n; i++) {
   u[i] = u0[i];
   if (terme1) u[i] += a*v[i];
   if (terme2) u[i] += b*w[i];
}
```

par:

```
if (terme1) aa = a; else aa = 0.0;
if (terme2) bb = b; else bb = 0.0;
for (i=0; i<n; i++) {
    u[i] = u0[i] + aa*v[i] + bb*w[i];
}
```