Parallélisme

Résumé des cours dispensés par Jens Gustedt à l'Université de Strasbourg

Session de printemps 2018

L'USAGE DE CE DOCUMENT NE PEUT ÊTRE QU'ACADÉMIQUE

Mise en forme par Marek Felšöci

4 février 2018

Crédits

Ce résumé s'appuie sur les notes notes du cours de Parallélisme dispensé par Jens GUSTEDT à l'Université de Strasbourg.

1 Introduction

Une **machine parallèle** est une collection d'éléments de calcul capables de communiquer et de coopérer dans le but de **résoudre rapidement des problèmes de grande taille**.

1.1 Motivations

Le parallélisme apporte de la **rapidité** lorsqu'il y a un grand nombre de calculs à effectuer comme par exemple :

- simulations physiques (*météo, jeux*)
- exploration d'état symbolique (*théories mathématiques, cryptanalyse*)
- visualisations (*jeux*)

Il peut également aider lors de traitement et de stockage de grands volumes de données (*Large Hydron Collider au CERN*) :

- moteurs de recherche sur Internet
- génération de séquences d'images (*films*)

La sûreté de fonctionnement peut être renforcée par les redondances matérielle (*avions, fusées*) et logicielle (*systèmes d'exploitation, processeurs, serveurs Web*).
Enfin, le parallélisme est outil aussi dans le domaine de sécurité car il permet, par exemple, l'encapsulation d'état (*voiture, smartphone*).

2 Machine parallèle

2.1 Capacité

La capacité d'une machine parallèle se mesure en *flops (floating point operations per second)*.
Pour mettre en avant l'avancé dans la capacité de machines de calcul listons quelques dates clés :

- 1941 : ZUSE Z3 avec 2 *flops*
- 1947 : ENIAC avec 1000 *flops*
- 1990 : PC avec 1 *Mflops*
- 2009 : PC avec 3 *Gflops*
- 2016 : *Core*^{TM} *i7* à quatre cœurs avec 90 *Gflops*

Des grandes machines parallèle offrent encore plus de puissance. En voici quelques unes :

- **Taihu light** : 10 millions de cœurs, 93 (*Pflops*), 15 MW
- **Tanhe-2** : 3 millions de cœurs, 33 *Pflops*, 17 MW
- **Titan** : 500 mille cœurs, 17 *Pflops*, 8 MW

L'énérgie utilisée par ces machines croît avec la fréquence mais pas linéairement. Cette croissance suit une fonction superlinéaire.
Un autre problème est le fait que la vitesse de la lumière soit bornée. De plus la miniaturisation augmente les effets quantiques. C'est pourquoi la fréquence d'horloge raisonnable est de 4 GHz. Cependant, normalement on se situe entre 2 et 3 GHz.

2.1.1 Loi de Moore

Cette loi nous dit que le nombre de transistors sur une puce double tous les 18 mois. Le seul moyen alors d'accélérer les calculs c'est de faire plusieurs choses à la fois. Néanmoins il faut comprendre que les architectures parallèles nécessitent également des logiciels parallèles. Ils le sont presque tous de nos jours.

2.2 Programmation

Comment programmer une machine parallèle ?

- **correctement** : problèmes de cohérence de données (écrasement), de blocage et d'ordonnancement des instructions (ordre numérique)
- **efficacement** : trouver une façon d'assurer une accélération et un coût raisonnable
- **à large échelle** : problème de répartition de données et de recollecte des résultats

2.3 Architectures

2.3.1 Classification

On utilise la classification de **Flynn** élaborée en 1966. Celle-ci repose sur deux critères pour caractériser les modes de calcul qui sont les flots de données et d'instructions :

	Simple	*Multiple*
Single	SISD	SIMD
Multiple	MISD	MIMD

Table 1: Classification de Flynn

SISD correspond à une machine séquentielle comme spécifiée par Von Neumann.

SIMD signifie de faire une même opération sur plusieurs données à la fois (*vector processor*) mais n'existe plus aujourd'hui à l'exception des GPU et des instructions de processeur comme SSE ou AVX.

MISD représente les unités de traitement en *pipelines*effectuant plusieurs instructions à la fois (simultanément).

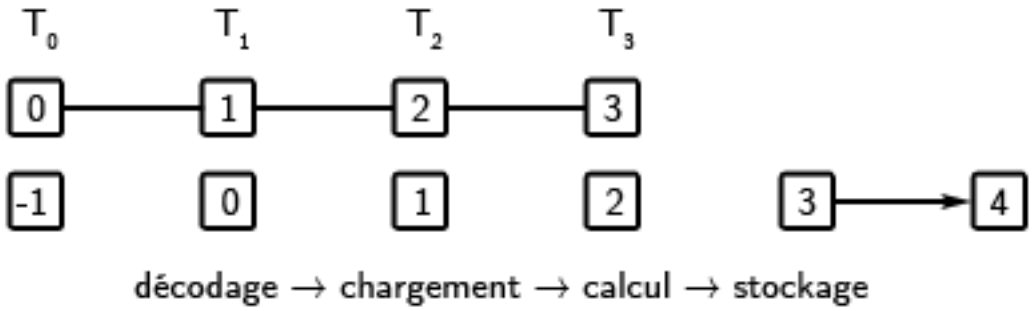

Figure 1: Illustration d'une unité de traitement en *pipelines*

MIMD est la plus importante de nos jours. Exemples :

- multicœurs
- machines parallèles spécialisées HPC (*Blue Gene*)
- grappe (*cluster*) : 10 à 1000 PC connectés via le réseau
- grille de calcul (grappes reparties dans le monde en plusieurs milles en distribué)
- CLOUD : service de calcul virtualisé à grande échelle

2.3.2 Caractérisation

Mémoire partagée Dans ce cas chaque processeur accède à la mémoire partagée en écrivant sur le bus (étranglement) et un autre processeur lit le mot écrit.

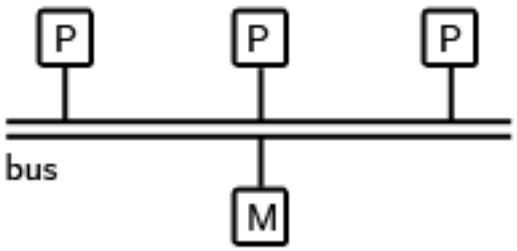

Figure 2: *Simetric Multi Processor*

- Les adresses sont les mêmes pour tous les processeurs.
- Un seul accès est permis à la fois.

La mémoire partagée sert aux processeurs comme moyen de communication mais elle est plus lente que la fréquence des processeurs. On introduit alors l'architecture à cache (Core™ Duo, AMD DualCore avec L2 séparée):

Mémoire distribuée (voir la figure ??) En revanche en distribué chaque processeur a sa propre mémoire et la communication se fait par échange de messages via un réseau d'interconnexion (commutateur *Myrinet* spécialisé, *Gigabit Ethernet*, Internet ou NoC : *Network on chip*). Les avantages de ce modèle sont l'indépendance des composantes, la modularité et la scalabilité de l'architecture.

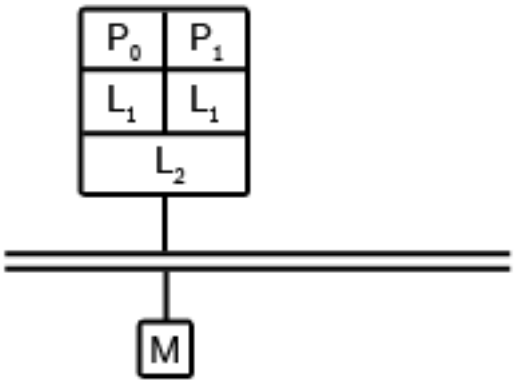

Figure 3: *Exemple d'architecture à cache*

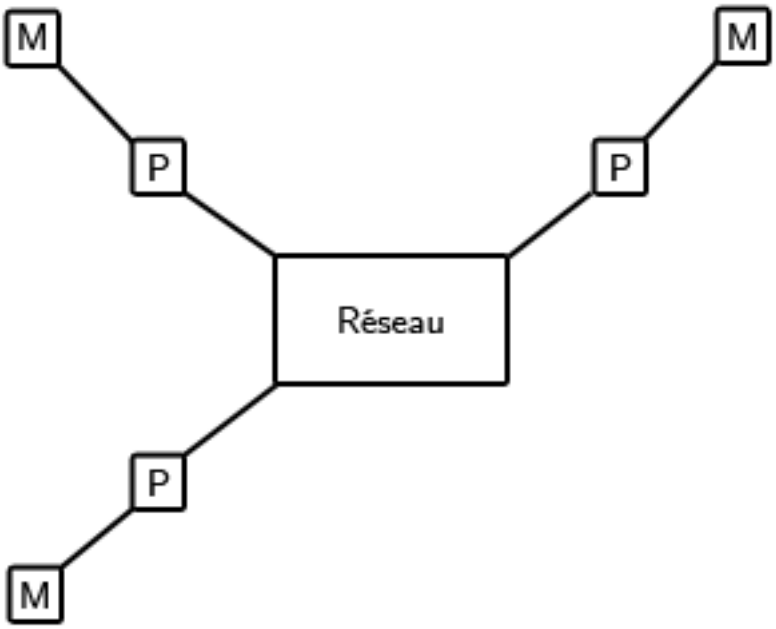

Figure 4: *Modèle de mémoire distribuée*

2.4 Performance

Définitions :

- **accélération** (*speed-up*) : $S(n,p) = \frac{T_1(n)}{T_p(n)}$ où $T_1(n)$ est le temps séquentiel d'un programme spécifique pour une donnée de taille n et $T_p(n)$ le temps parallèle avec p processus
  - si $S < p$ alors il s'agit d'accélération sous-linéaire (normal)
  - si $S = p$ alors il s'agit d'accélération linéaire (rare)
  - si $S > p$ alors il s'agit d'accélération sur-linéaire (exceptionnel)
- **éfficacité** : $E(n,p) = \frac{S(n,p)}{p} = \frac{T_1(n)}{p \times T_p(n)}$
- **coût total** : $C(n,p) = p \times T_p(n)$

2.4.1 Loi d'Amdahl (1967)

Suppositions :

- $T_1(n) = T_s(n) + T_p(n)$ où $T_s(n)$ est la partie non-parallélisable et $T_p(n)$ la partie parallélisable
- $T_s(n) = f(n) \times T_1(n)$ où $f(n)$ est la fraction séquentielle
- $T_p(n) = (1 - f(n)) \times T_1(n)$

Le meilleur temps parallèle possible est $T_p(n) = T_s(n) + \frac{T_p(n)}{p} = (f(n) + \frac{1-f(n)}{p}) \times T_1(n)$ d'où $lim_{p\to\infty} S(n,p) = lim_{p\to\infty} \frac{1}{f(n)+\frac{1-f(n)}{p}} = \frac{1}{f(n)}$.

Exemple : $\forall\ n,\ f(n) = 10\%$ donne l'accélération de 10 et l'éfficacité nulle.

2.5 Programmation

Étudions la technique de programmation parallèle sur un exemple :

Évaluation d'un polynôme $P(x) = a + b \times x + c \times x^2 + d \times x^3$
En séquentiel :

```
for (i = 0; i < n; i++)
  res[i] = a + b * v[i] + c * v[i] ^ 2 + d *
     v[i] ^ 3;
```

En parallèle, on a trois possibilités :

1. **parallélisme des données** (s'assurer de l'indépendance des données) :

   ```
   parallel for (i = 0; i < (n - 1); i++)
     res[i] = P(v[i]);
   ```

   Un compilateur parallèle produit un exécutable parallèle. Mais généralement le nombre de processeurs est inférieur à N. Comment répartir alors les itérations sur les processeurs ?

   (a) **distribution** cyclique : le processeur r prend tous les i avec $i \,\%\, p = r$

   (b) **distribution par bloc** : $r \times \frac{N}{p} \leq i < (r+1) \times \frac{N}{p}$

   (c) **distribution par blocs cycliques** : pour b une taille de blocs on a $\lfloor \frac{i}{b} \rfloor \,\%\, p = r$

2. **parallélisme de tâches** (assurer la synchronisation) : $R = a + b \times v(i)$, $S = c + d \times v(i)$, $T = v(i) \times v(i)$, $res(i) = R + S \times T$. Néanmoins l'ordonnancement des systèmes de tâches est difficile.

3. **parallélisme de flux** (*pipelines*) : reformulation du calcul avec le schéma de Horner ressemble à $res(i) = a + v(i) \times (b + v(i) \times (c + v(i) \times d))$

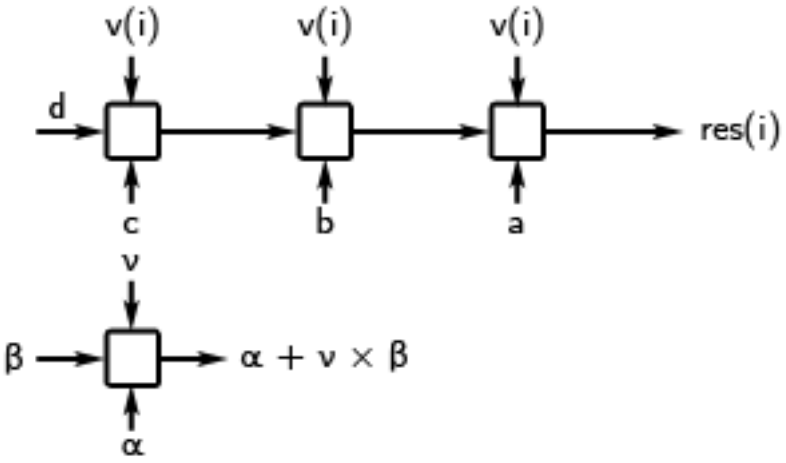

Figure 5: *Floating Point Multiply Add*

4. **parallélisme de la mémoire vive** : implique un nombre arbitraire de processeurs synchronisés disposant des instruction habituelles (addition, multiplication, etc.) ainsi que l'utilisation d'une mémoire partagée.
   Dans ce cas on analyse deux types de ressources :

   - le temps total d'une exécution
   - le coût exprimé en nombre total d'instructions

   Exemple : *opération OU (algorithmes)*
   La donnée en entrée est un vecteur $a[]$ de base. La question qu'on se pose est de savoir s'il y a une seule valeur qui soit vraie.

```
result = false;
parallel for (i = 0; i < (n - 1); i++)
```

```
    if(a[i]) result = true;
  return result;
```

Le coût est de $O(n)$ et le temps est constant $O(1)$.
Le problème est que plusieurs processeurs écrivent dans la même variable . On distingue alors *Concurrent Read and Concurrent Write* dans la RAM parallèle ce qui n'est pas réaliste. Il faudrait donc mettre en place *Concurrent Read and Exclusive Write* :

```
parallel for (i = 0; i < (n - 1); i++)
  b[i] = a[i]; // tout acces est exclusif
while (n > 1) {
  r = n % 2;
  m = floor(n / 2); // partie entiere
     inferieure
  n = m + r;
  parallel for (i = 0; i < (m - 1); i++)
    b[i] = (b[i] || b[i + n]);
}
return b[0];
```

Dans ce cas le temps est de $O(log(n))$ et le coût $O(n * log(n))$ le but étant d'avoir le coût de $O(n)$ avec $\frac{n}{log(n)}$ processeurs.
Toujours est-il que l'accès concurrent à la mémoire est primordial. Il faut donc assurer la **cohérence** et la **performance**.
Exemple :

```
tmp0 = a;
tmp1 = a;
tmp0 = tmp0 + 1;
tmp1 = tmp1 + 1;
a = tmp0;
a = tmp1;
```

Ici la variable a est augmentée deux fois : $2 \times op(a+1)$!

3 Communication MPI

3.1 Instructions atomiques

Ce sont des instructions spécialisées caractérisées comme étant **indivisibles**, **linéarisables** et **non-interruptibles**.
Pour déclarer une variable comme atomique on utilise la syntaxe suivante : *unsigned _Atomic int counter = 0*
Ainsi toutes les opérations effectuées sur celle-ci sont atomiques et donc effectuées en une seule opération. Mais le coût en est énorme.

3.2 Passage de messages

Chaque acteur (processeur ou processus) dispose d'une adresse et d'un état séparés. Ils communiquent en mode *peer-to-peer*. Ainsi on évite l'accès concurrent.
Pour communiquer on utilise les fonctions *send* (envoi de message) et *recv* (réception de message).
Plusieurs types de communication sont possibles :

- **fiable**
- **ordnonnée** (sans déséquencement)
- **synchrone** (aller-retours en direct)
- **asynchrone** (sans réponse instantannée)
- **bloquante**

3.3 Autres primitives (macro communication)

1. **synchronisation** [tous, pas de donnée] : attendre que tous les acteurs aient fini l'opération concernée pour pouvoir synchroniser
2. **diffusion** (*broadcast*) [1 à tous, le même message]
3. **distribution** (*scatter*) [1 à tous, n messages différents]
4. **rassemblement** (*gather*) [tous à 1, n messages différents]
5. **commérage** (*all-to-all*) [tous à tous, n messages différents]
6. **transposition** (*multiscatter*) [tous à tous, n^2 messages différents]
7. **réduction** : pour une opération $\otimes$ on a $d = \bigotimes_{i=0}^{n-1} d_i$
8. **traduction** : diffusion

Les six premières opérations sont inefficaces si elles sont réalisées en mode *peer-to-peer*.