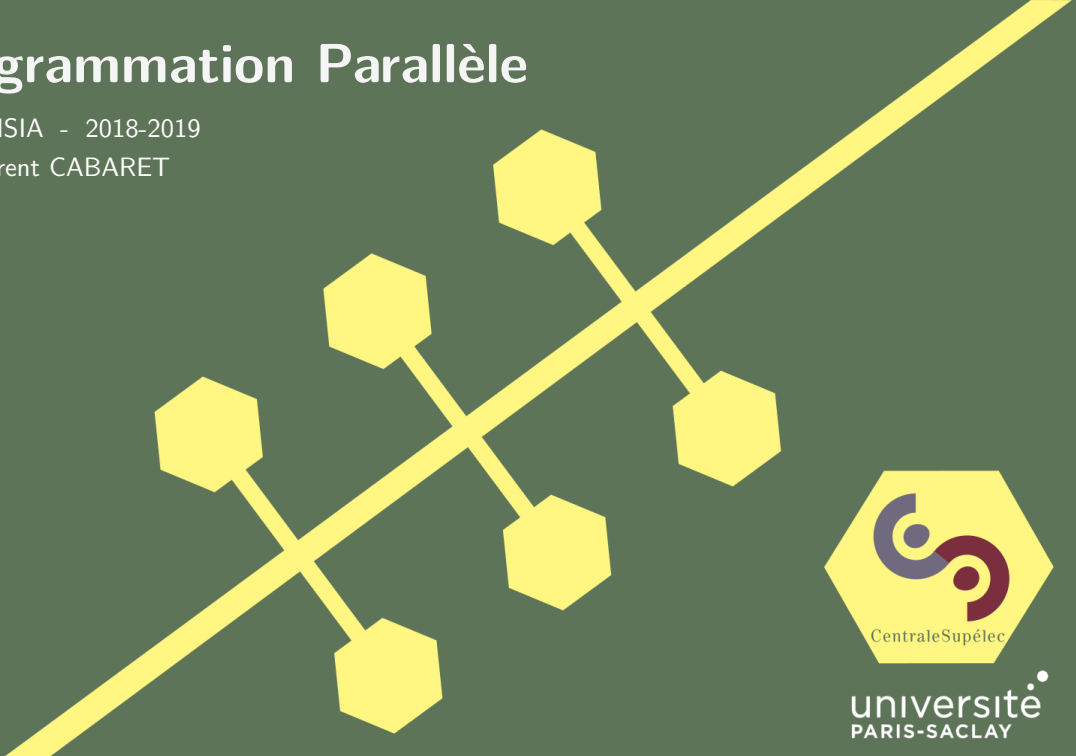
Programmation Parallèle

Option ISIA - 2018-2019

par Laurent CABARET



CentraleSupélec

université PARIS-SACLAY

Plan

- **Présentation** : Objectifs du cours, déroulement, évaluation, illustrations
- **Ab uno disce omnes** : Architecture séquentielle et mur de la chaleur
- **Paradigme parallèle**
- **Parallélisme d'instruction** : Vectorisation et SIMD
- **Parallélisme multi-cœur** : Mémoire partagée, OpenMP, Race conditions
- **Parallélisme massif** : cas du GPU, programmation CUDA

Présentation (1/2)

Objectifs du cours

- À l'issue de ce cours vous serez en mesure de concevoir des algorithmes parallèles et de les implémenter sur CPU en utilisant OpenMP ou les Intrisics SIMD ou sur GPU en utilisant CUDA

Déroulement du cours

- Le cours sera composé d'une alternance de séances de cours/TD, de TPs, de séances de travail en équipe sur une étude de cas de parallélisation.

Evaluation

- L'évaluation sera basée sur les compte-rendus de TP et sur le résultat de l'étude de cas (soutenance orale, code et analyse).

Présentation (2/2)

Ce que nous ne ferons pas :

- Programmation Multi-tâches
- Découvrir des architectures historiques ou exotiques
- Preuve de programmes parallèles
- Programmation distribuée MPI

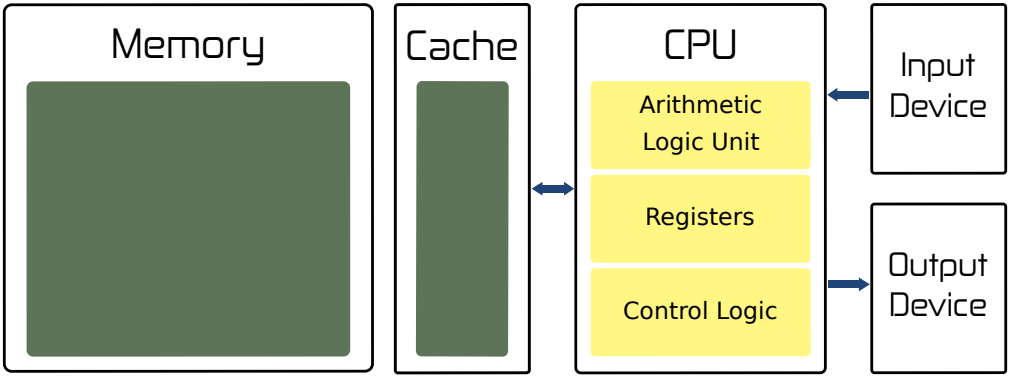
Quelques illustrations

- $\forall i \in [0, 10^6], S[i] = A[i] + B[i]$
- $s = \sum_{n=0}^{10^6} A[i]$
- $\forall (x, y) \in [1, 2046]^2, G_x[x, y] = I[x + 1, y] - I[x - 1, y]$
- $\forall (x, y) \in [1, 2046]^2, S[x, y] = I[x, y] + I[x, y + 1]$

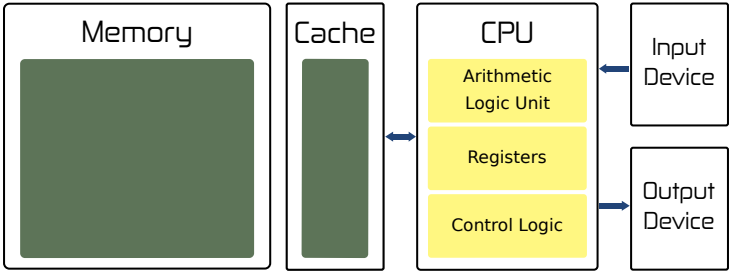
Ab uno disce omnes*

* Et qu'un seul vous apprenne à les connaître tous

Architecture d'un processeur mono-cœur

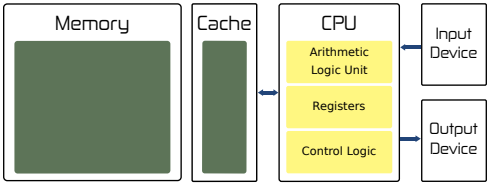


Cycle d'instructions idéalisé



Le fonctionnement basique d'un processeur est composé de 3 actions principales.

Cycle d'instructions augmenté



Assembleur

Commande pour obtenir un assembleur un peu lisible

```
# gcc (-O0 or O1) -S -fno-asynchronous-unwind-tables -fno-dwarf2-cfi-asm -
fno-exceptions -fomit-frame-pointer file.c
```

C code

```
int main() {
  int a=1;
  int b=7;
  a=a+3;
  b=b+2;
  a=a+b;
  return a;
}
```

Asm code (-O0)

```
movl    $1, -8(%rsp)
movl    $7, -4(%rsp)
addl    $3, -8(%rsp)
addl    $2, -4(%rsp)
movl    -4(%rsp), %eax
addl    %eax, -8(%rsp)
movl    -8(%rsp), %eax
ret
```

Asm code (-O1)

```
movl    $13, %eax
ret
```

Facteur déterminants pour le temps d'exécution (1/2)

$$T_{exec} = NI \times CPI \times TC \iff T_{exec} = \frac{NI}{IPC \times F}$$

avec

- T_{exec} : le temps total d'exécution
- NI : Le nombre d'instruction du programme
- CPI : Le nombre de cycles machine par instruction
- IPC : Le nombre d'instructions par cycle
- TC : Le temps d'un cycle
- F : La fréquence d'exécution

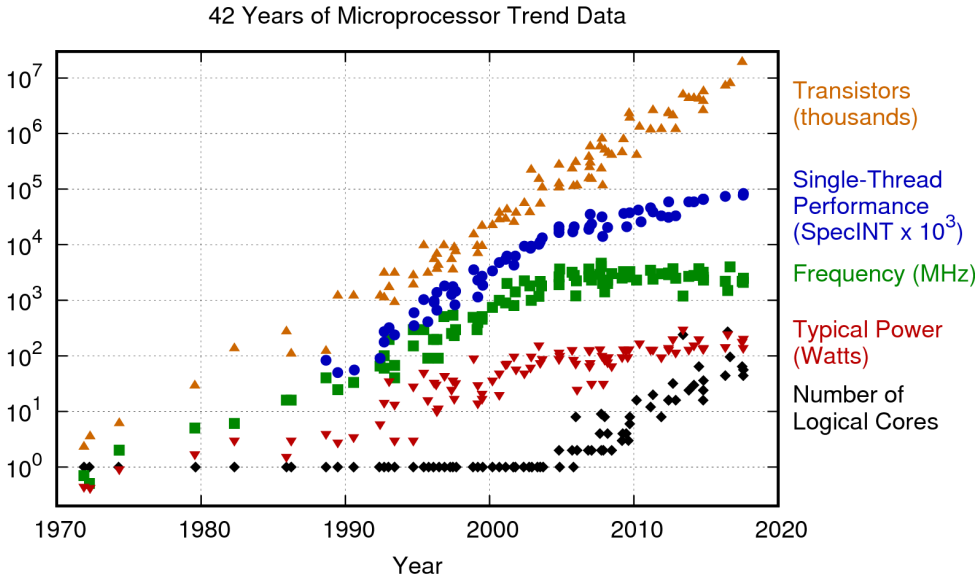
Facteur déterminants pour le temps d'exécution (2/2)

$$T_{exec} = \frac{NI}{IPC \times F}$$

3 options pour accélérer

- F : La fréquence d'exécution $\rightarrow$ progrès matériels régulier jusqu'à ...
- NI : Diminuer le nombre d'instructions du programme $\rightarrow$ travail sur l'algorithmie et travail du compilateur
- IPC : Le nombre d'instructions par cycle $\rightarrow$ travail sur l'architecture

La fréquence est limitée par le « mur de la chaleur »

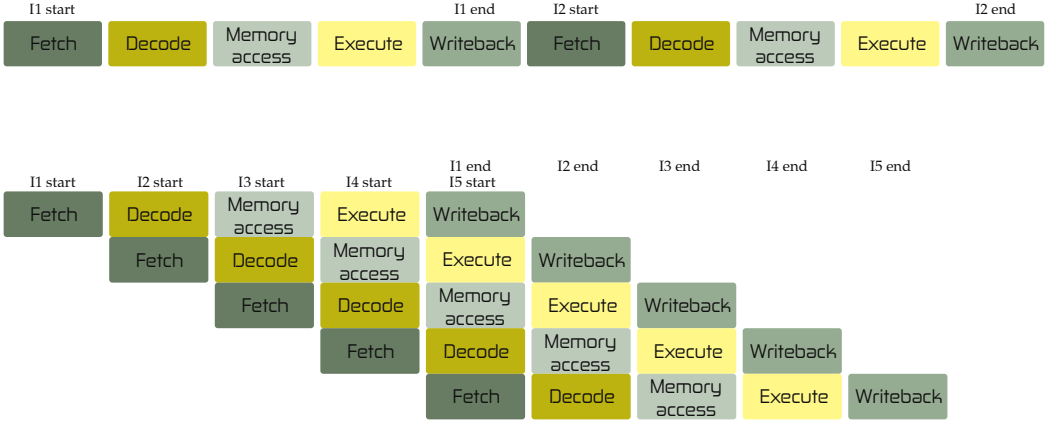


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Il faut améliorer l'architecture

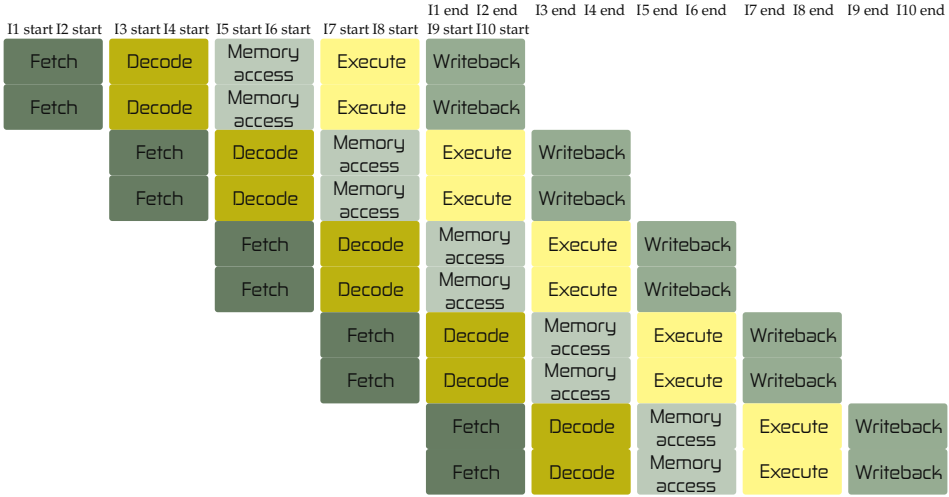
- Pipeline
- Superscalaire
- ... ?

Principe du pipeline



l'IPC est multiplié par 5 !

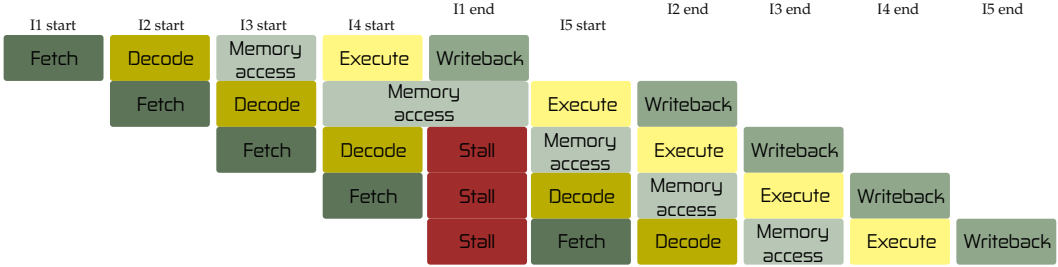
Principe du superscalaire



l'IPC est encore multiplié par 2 !

Dépendance structurelle : pipeline stall

L'introduction du pipeline nous oblige à examiner attentivement les dépendances entre les données et entre les instructions.



Dépendance de données

Il faut garantir la reproductibilité du programme dans tous les cas

Accès en lecture ou en écriture au même registre ou à la même adresse mémoire

- Read after read (RAR) : Pas d'incidence. **On peut changer l'ordre**.
- Read after write (RAW) : L'instruction I_A doit avoir écrit son résultat dans `var` avant que I_B ne le lise.
- Write after read (WAR) : L'instruction I_A doit avoir lu la donnée dans `var` avant que I_B ne la modifie.
- Write after write (WAW) : L'instruction I_A doit avoir écrit son résultat dans `var` avant que I_B ne la modifie. **Il y a dans tous les cas un problème de concurrence, mais l'impératif est de garantir l'ordre d'exécution.**

Dépendances de contrôle (1/2)

Le pipeline a permis d'augmenter le nombre d'instructions par cycles mais est très sensible aux branchements (If/Else)

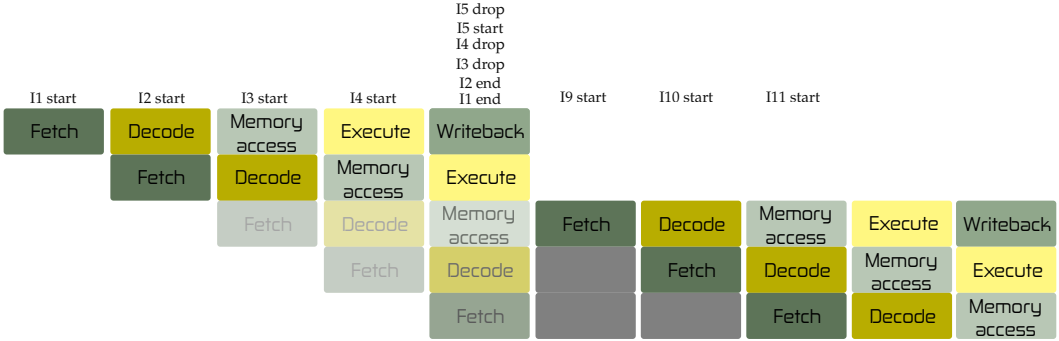
C code

```
int main() {
  int a=1;
  if (a>3) {
      a=a+7;
      return a;
  }
  else {
      return a;
  }
}
```

Asm code (-O0)

```
main:
        movl    $1, -4(%rsp)
        cmpl    $3, -4(%rsp)
        jle     .L2
        addl    $7, -4(%rsp)
        movl    -4(%rsp), %eax
        ret
.L2:
        movl    -4(%rsp), %eax
        ret
```

Dépendances de contrôle (2/2)



Toutes les instructions démarrées sont perdues et le pipeline doit être «rechargé» pour redevenir efficace.

Quelques solutions

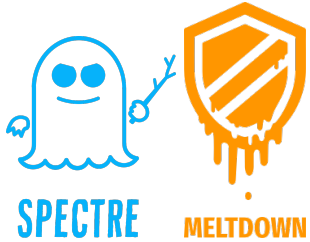
Out of order execution

L'exécution dans le désordre peut permettre de réordonnancer les instructions pour minimiser : les dépendances structurelles et les dépendances de données.

Prédiction de branchements ou exécution spéculative

Cela revient à parier sur le résultat d'un test et de charger le pipeline avec la branche la plus probable.

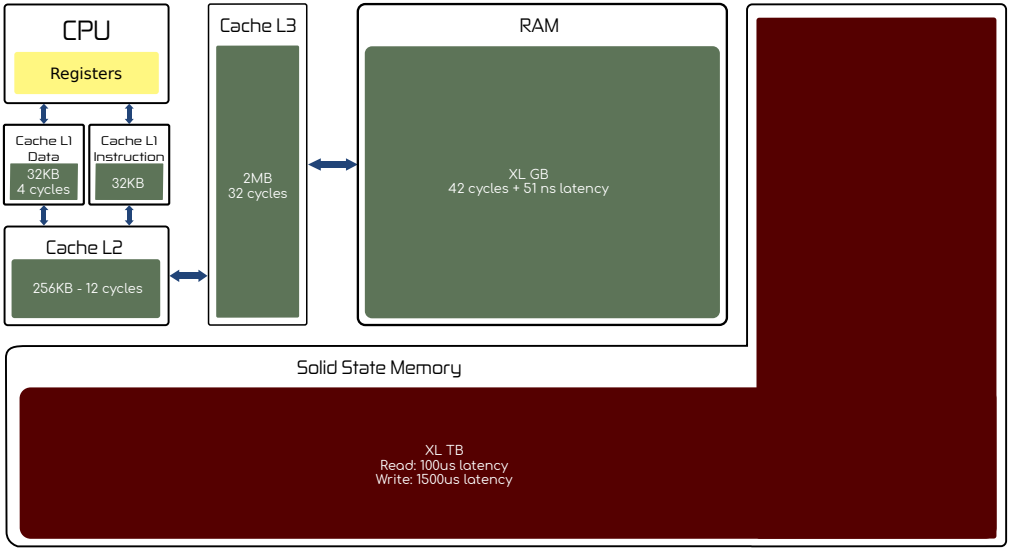
Légér inconvénients !



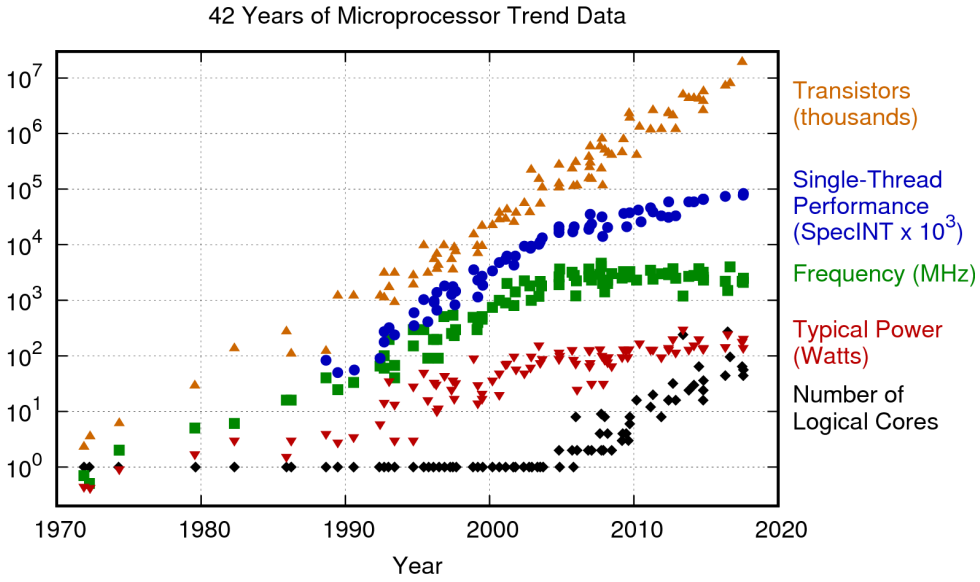
Pour en savoir plus sur ces mécanismes [3] et sur les inconvénients [4]

Et la hiérarchie mémoire ?

Exemple du Coffee Lake (4 cycles = 1ns)

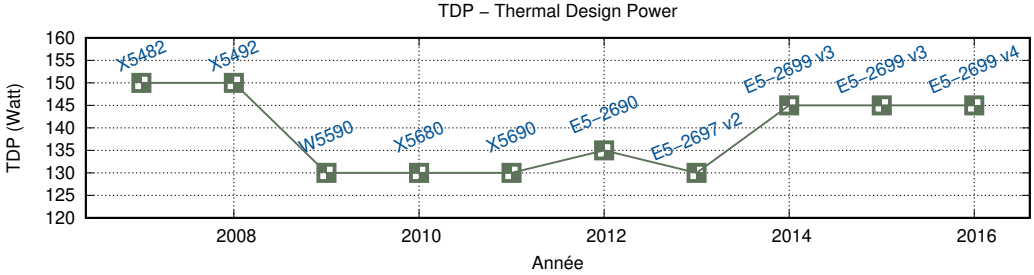


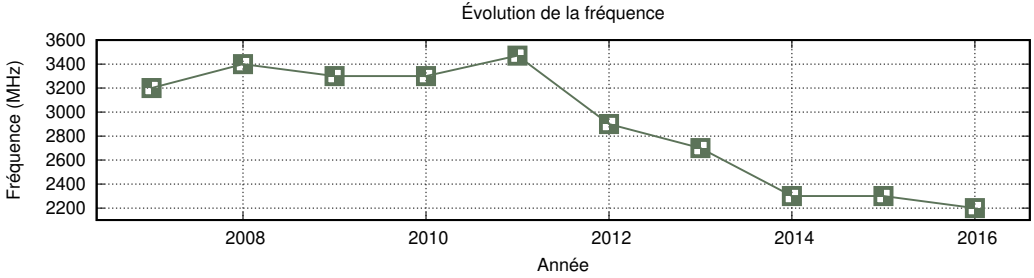
La fréquence est limitée par le « mur de la chaleur »



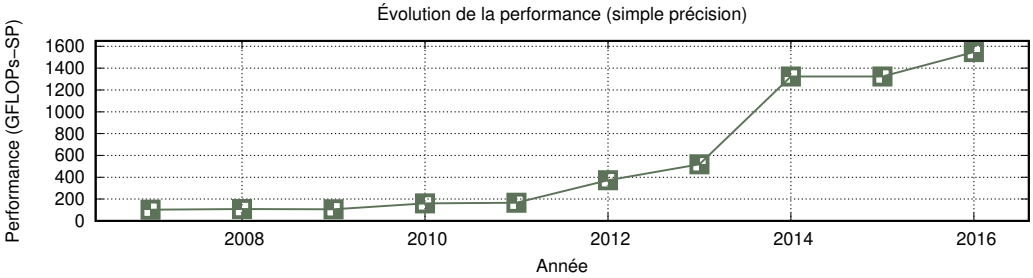
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

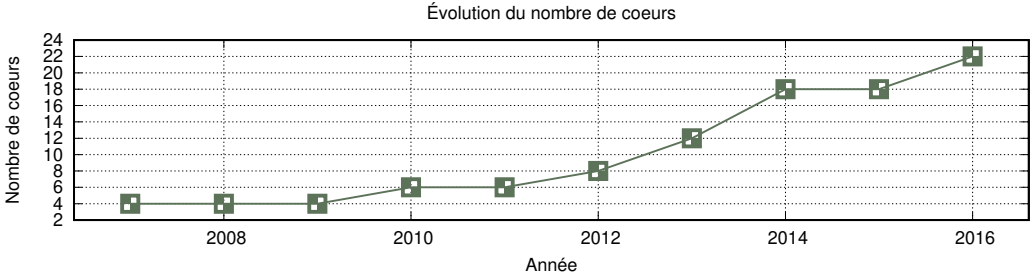
La performance n'est plus dans la puissance d'un cœur





Mais dans le nombre de cœurs





En savoir plus / Références

[1] Daniel Etiemble. *Jeux d'instructions*. 2014. URL : https://www.lri.fr/%7Ede/M2R-Jeux_instructions-1314.pdf.

[2] Daniel Etiemble. *Pipeline scalaire*. 2014. URL : https://www.lri.fr/%7Ede/M2R-Pipeline_scalaire-1314.pdf.

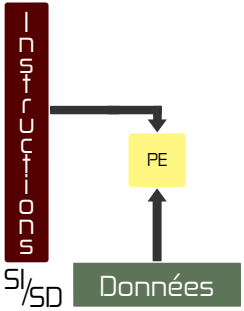
[3] Nathanaël Prémillieu. "Increase Sequential Performance in the Manycore Era". Theses. Université Rennes 1, déc. 2013. URL : https://tel.archives-ouvertes.fr/tel-00916589.

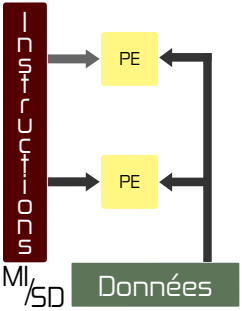
[4] Wired. *A Critical Intel Flaw Breaks Basic Security for Most Computers*. 2018. URL : https://www.wired.com/story/critical-intel-flaw-breaks-basic-security-for-most-computers/ (visité le 05/01/2018).

Paradigme parallèle

Taxonomie de Flynn (1/2)

- **SISD** : Cas d'un programme séquentiel sur un cœur sans instructions vectorielles
- **MISD** : Cas très rare de processeur de traitement du signal

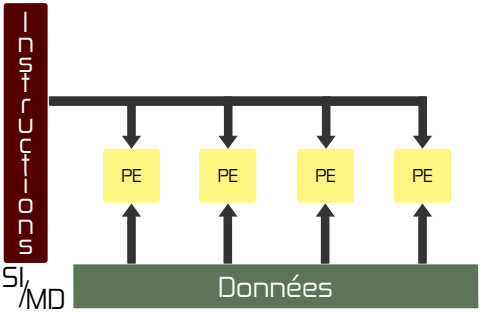


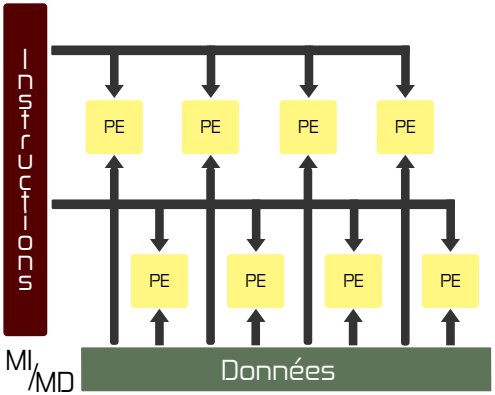


plus de détails dans l'article[3]

Taxonomie de Flynn (2/2)

- **SIMD** : Processeur vectoriel ou instructions vectorielles dans un processeur
- **MIMD** : Cas omniprésent aujourd'hui d'une ou plusieurs machines avec un ou plusieurs processeurs multi-cœur, chaque cœur pouvant exécuter des instructions vectorielles





Evaluation de la performance

Pour un programme séquentiel exécuté sur un seul cœur de processeur, la performance de programme sera exprimée soit :

- en fonction du **temps d'exécution** si la taille des données à traiter est fixe
- en fonction du **nombre de données traitées par unité de temps**.

Dans la suite nous nous placerons dans le premier cas et donc :

$$T_S = \text{Temps d'exécution séquentiel}$$

Pas extension on défini :

$$T_P = \text{Temps d'exécution du programme sur une architecture parallèle}$$

L'accélération obtenue par l'exécution parallèle est donc :

$$Acc = \frac{T_S}{T_P}$$

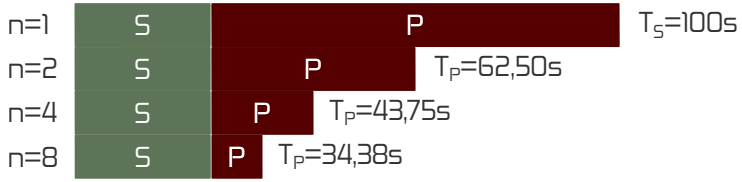
Quelle accélération peut-on espérer ?

Chaque programme est constitué d'une portion parallélisable et d'une portion non parallélisable (séquentielle).
Soit S la portion séquentielle et P la portion parallélisable. Sur un cœur on a :

$$T_S = S + P$$

L'exécution d'un tel programme sur une architecture à n cœurs prendra alors :

$$T_P = S + \frac{P}{n}$$



Point de vue de Amdahl

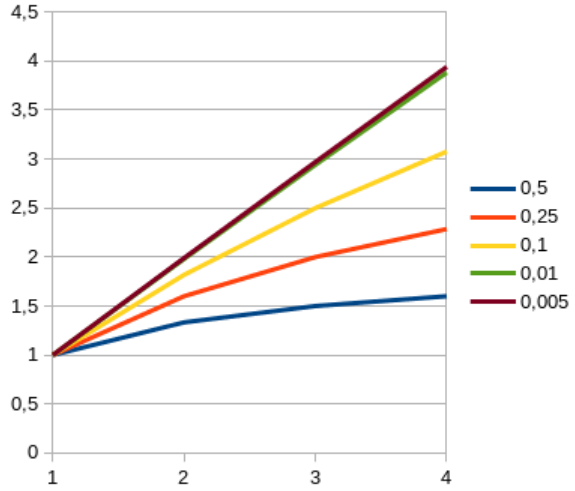
Lorsque la quantité de données est constante l'accélération maximale en fonction du nombre n de processeurs est donnée par la loi d'Amdahl [1].

$$Acc = \frac{S+P}{S+\frac{P}{n}}$$

en posant s tel que $S = s \times T_S$ (et donc $P = (1-s) \times T_S$).

$$\boxed{Acc(n) = \frac{1}{s+\frac{1-s}{n}}}$$

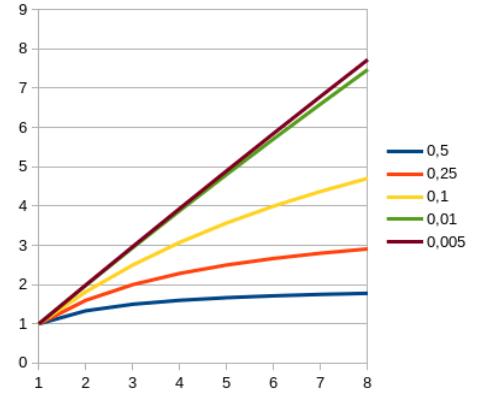
n=4



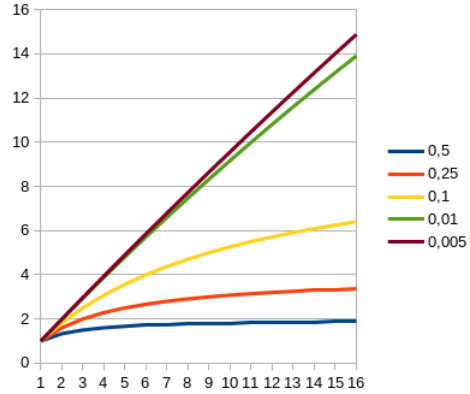
Point de vue de Amdahl

$$\lim_{n \to \infty} Acc(n) = \frac{1}{s}$$

n=8



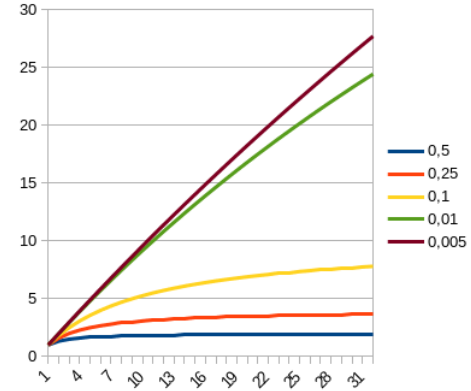
n=16



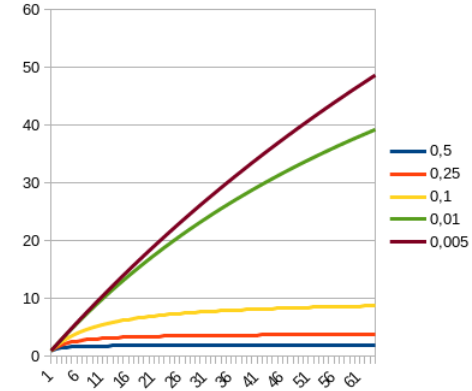
Point de vue de Amdahl

$$\lim_{n \to \infty} Acc(n) = \frac{1}{s}$$

n=32



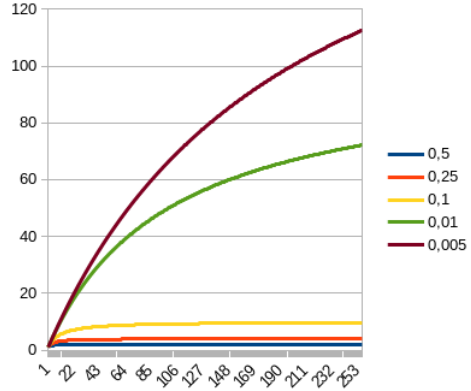
n=64



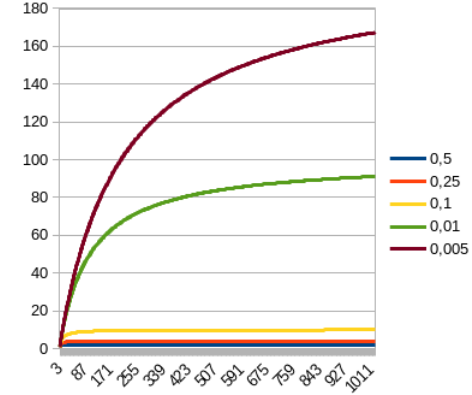
Point de vue de Amdahl

$$\lim_{n\to\infty} Acc(n) = \frac{1}{s}$$

n=256



n=1024



$N_{1/2}$

$N_{1/2}$ est la valeur de n qui permet d'obtenir 50% de la performance maximale. La performance maximale ne l'étant qu'avec un nombre infini de processeurs, la valeur de $N_{1/2}$ est un bon indice de la capacité de l'algorithme de passer à l'échelle.

$$\frac{1}{2 \times s} = \frac{1}{s + \frac{1-s}{N_{1/2}}}$$

$$N_{1/2} = \frac{1-s}{s}$$

Portion séquentielle s	Accélération maximale ($n = \inf$)	$N_{1/2}$
0,5	2	1
0,25	4	3
0,1	10	9
0,01	100	99
0,005	200	199

Remarques sur Amdahl

Comment obtenir s ?

Il n'est pas possible de calculer s directement à partir de la description de l'algorithme même si il est possible de le qualifier entre «faible» et «fort».
s est obtenu expérimentalement par comparaison du temps dans un cas séquentiel et du temps dans plusieurs configurations de processeurs.

Que nous dit la loi de Amdahl ?

La loi d'Amdahl démontre qu'il n'est pas possible de passer correctement à l'échelle (*scaling*) en parallélisant une application donnée tout en conservant la même quantité de données.
L'augmentation constante de la quantité de données disponible permet d'aborder le parallélisme sous un autre angle : que devient l'accélération si la quantité de données augmente avec le nombre de processeurs.

Point de vue de Gustafson

Gustafson a proposé de reconsidérer Amdahl en se libérant de la contrainte d'une quantité de données limitée [4]. La quantité de données augmentant exactement comme le nombre n de processeurs, on a $T_S = S + n \times P$ et $T_P = S + P$ et donc :

$$Acc(n) = \frac{S + n \times P}{S + P}$$

en posant s tel que $S = s \times T_P$ (et donc $P = (1 - s) \times T_P$).

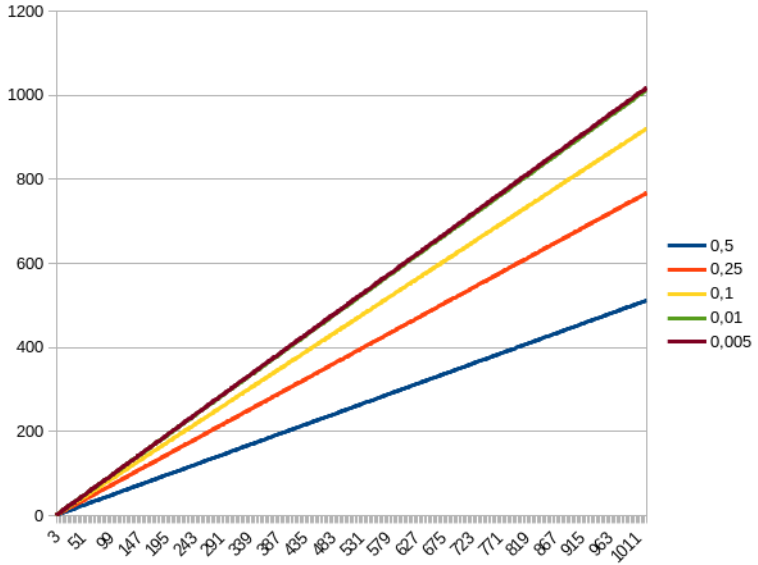
$$Acc(n) = \frac{s \times T_P + n \times (s \times T_P))}{T_P} = s + n \times (1 - s) = n + s \times (1 - n)$$

$$\lim_{s \to 0} Acc(n) = n$$

$$\lim_{s \to 1} Acc(n) = 1$$

$Acc(n)$ tend asymptotiquement vers $n \times (1 - s)$

Point de vue de Gustafson



Coût du parallélisme

Dans la présentation précédente des lois d'Amdahl et de Gustafson, nous avons considéré que le coût du parallélisme (*overhead*) était nul et que la charge était équirépartie sur l'ensemble des cœurs. Nous introduisons ici $\omega(n)$ qui représente les altérations qui proviennent des mécanismes de parallélisation.

$$T_P = S + \frac{P}{n} + \omega(n)$$

Paramètres influençant les performances de la parallélisation

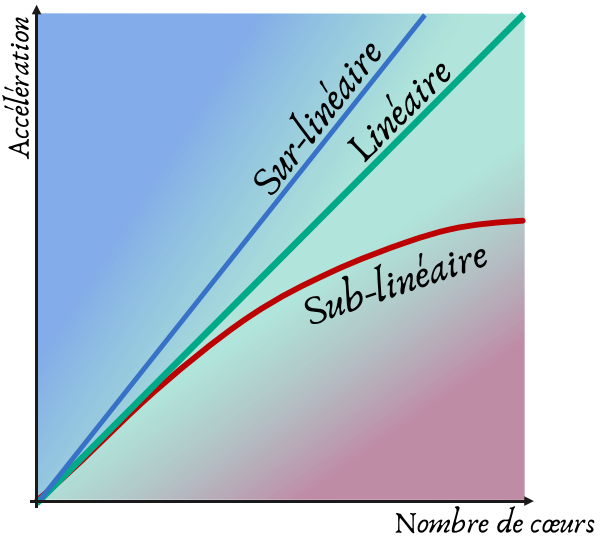
- Pour la création de chaque nouveau processus, le programme doit mettre en place des ressources supplémentaires, si la quantité de donnée est faible par rapport au nombre de thread utilisé, ce temps de mise en place devient non négligeable.
- Selon les applications, le découpage des données à traiter ne peut pas être équilibré *a priori*. Certains cœurs auront plus de travail à effectuer que d'autres.
- Le découpage des données peut renforcer la localité des traitements et permettre une meilleure utilisation des caches.
- La dispersion des données entre cœurs peut invalider les différents caches et entraîner des chargements depuis la mémoire principale. Le parallélisme peut entraîner une concurrence d'accès au données entraînants des erreurs, des ralentissements ou des blocages.

Types d'accélérations (speedup)

Linéaire

Sub-linéaire

Sur-linéaire

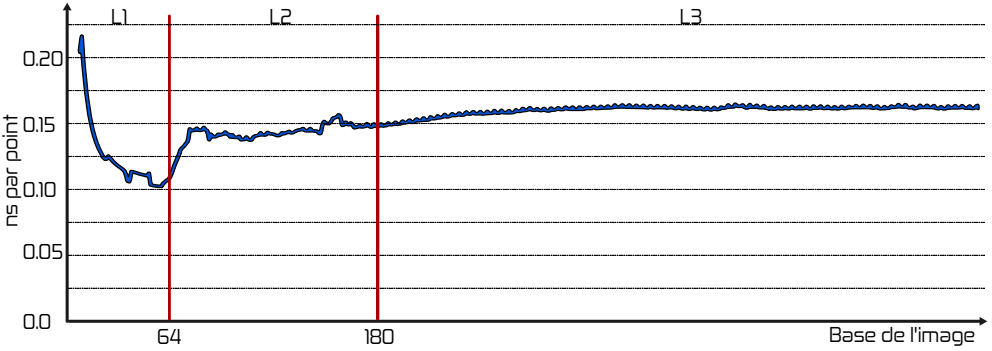


Sur-linéaire ? « À cause » des caches ?

Cache	Capacité	Ligne	Associativité
L1	32 KB	64 B	8-WAY
L2	256 KB	64 B	4-WAY
L3	8 MB	64 B	16-WAY

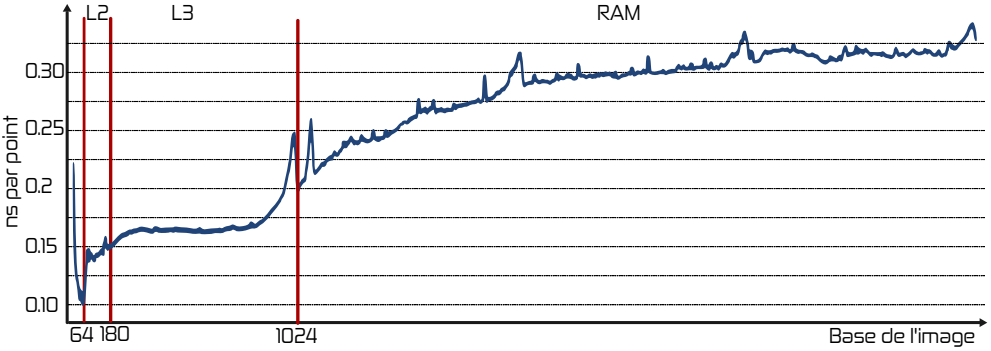
```
A = (float *) malloc(W*H * sizeof(float));
S = (float *) malloc(W*H * sizeof(float));
for (auto i = 0; i < (W*(H-1)); i++) {
    S[i] = A[i] + A[i+W];
}
```

cf. illustration 5 (slide 5)



Questions : $64^2 * 4 * 2 =$?, $180^2 * 4 * 2 =$?

Sub-linéaire ? « À cause » des défauts de caches ?

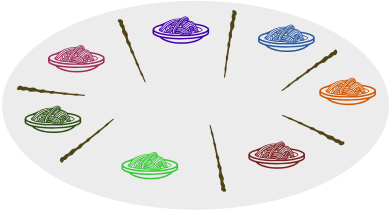


Prochain arrêt le fichier d'échange (*swap disc*)

Concurrence dans l'accès aux ressources

Un dîner des philosophes[2] presque parfait (revisité pour l'occasion)

Prems !



- Un philosophe a besoin de 2 baguettes pour manger des spaghetti
- Ils sont polis et il attendront d'avoir tous mangé avant de philosopher tous ensemble

Selon les stratégies il peux y avoir

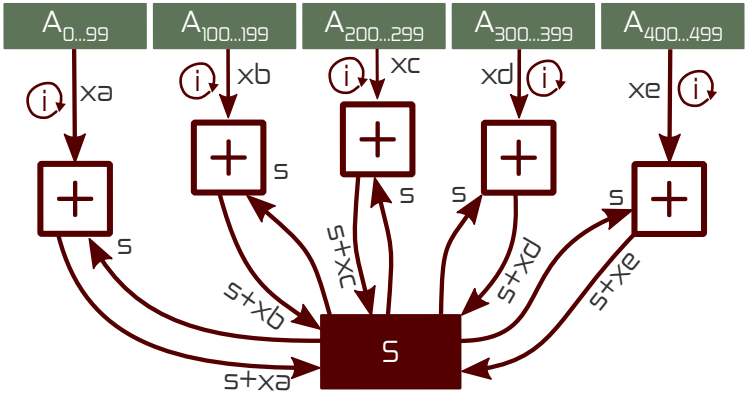
- Interblocage (dead lock) et donc famine (et donc aucune pensée de sortira de ce dîner !)
- Sous-utilisation des baguettes et des philosophes plus ou moins élevée
- Un sérieux manque d'hygiène

Quelque soit la stratégie, le nombre de philosophes en train de manger simultanément ne dépasse pas 3.

Concurrence dans l'accès aux données

$$s = \sum_{n=0}^{499} A[i]$$

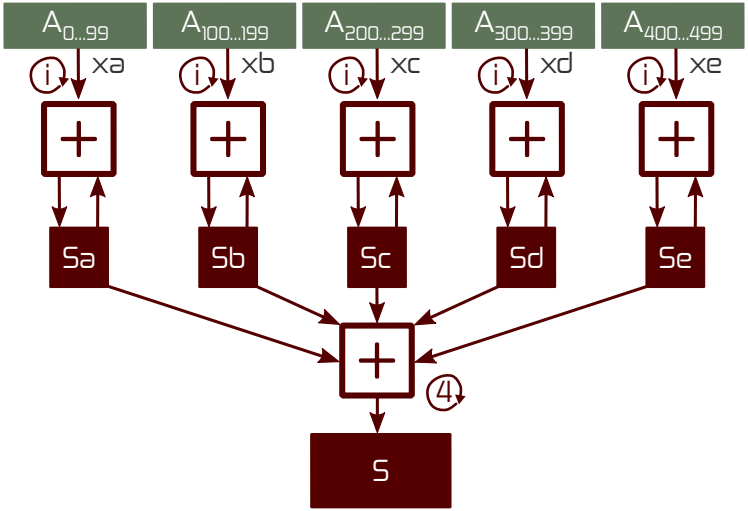
Cf. illustration 5 (slide 5)



Cas de « Race condition »

Concurrence dans l'accès aux données

Il faut gérer la concurrence au niveau algorithmique



Points d'attention pour un parallélisme efficace (voir correct !)

- Rendre le programme séquentiel le plus performant possible (sinon c'est triché).
- Monitorer les performances pour lutter contre les fausses bonnes idées et avoir un retour objectif sur la qualité de la parallélisation.
- Évaluer le ratio quantité de calculs/besoins mémoire. *Memory bound* vs *Computation bound*.
- Une bonne pratique est de penser le découpage des données pour minimiser la concurrence, les communications et le déséquilibre des charges. Mais selon les architectures et les applications ce n'est pas nécessairement optimal ou possible, il faut donc évaluer les performances de la parallélisation.
- S'assurer du déterminisme des résultats (le non déterminisme cache des *race condition*).

En savoir plus / Références

[1] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In : *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. ACM, 1967.

[2] Edsger Wybe Dijkstra. *Cooperating Sequential Processes, Technical Report EWD-123*. Rapp. tech. 1965. URL : https://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF.

[3] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In : *IEEE Trans. Comput.* 21.9 (sept. 1972), p. 948-960. ISSN : 0018-9340. DOI : 10.1109/TC.1972.5009071. URL : http://dx.doi.org/10.1109/TC.1972.5009071.

[4] John L. Gustafson. "Reevaluating Amdahl's Law". In : *Communications of the ACM* (1988).