Multiprocesseurs, multicœurs et parallélisme

Parallélisme avec les processeurs d'usage général

Fabrice HARROUET

U*niversité* **E***uropéenne de* **B***retagne*
É*cole* **N***ationale d'***I***ngénieurs de* **B***rest*
L*aboratoire d'***I***nformatique des* **SY***stèmes* **C***omplexes*

Contexte

Tendance du marché des ordinateurs grand public

- Processeurs récents *multicœurs* à un tarif très raisonnable
- Les machines *multiprocesseurs* deviennent plus abordables

Déjà bien supporté par les systèmes d'exploitation

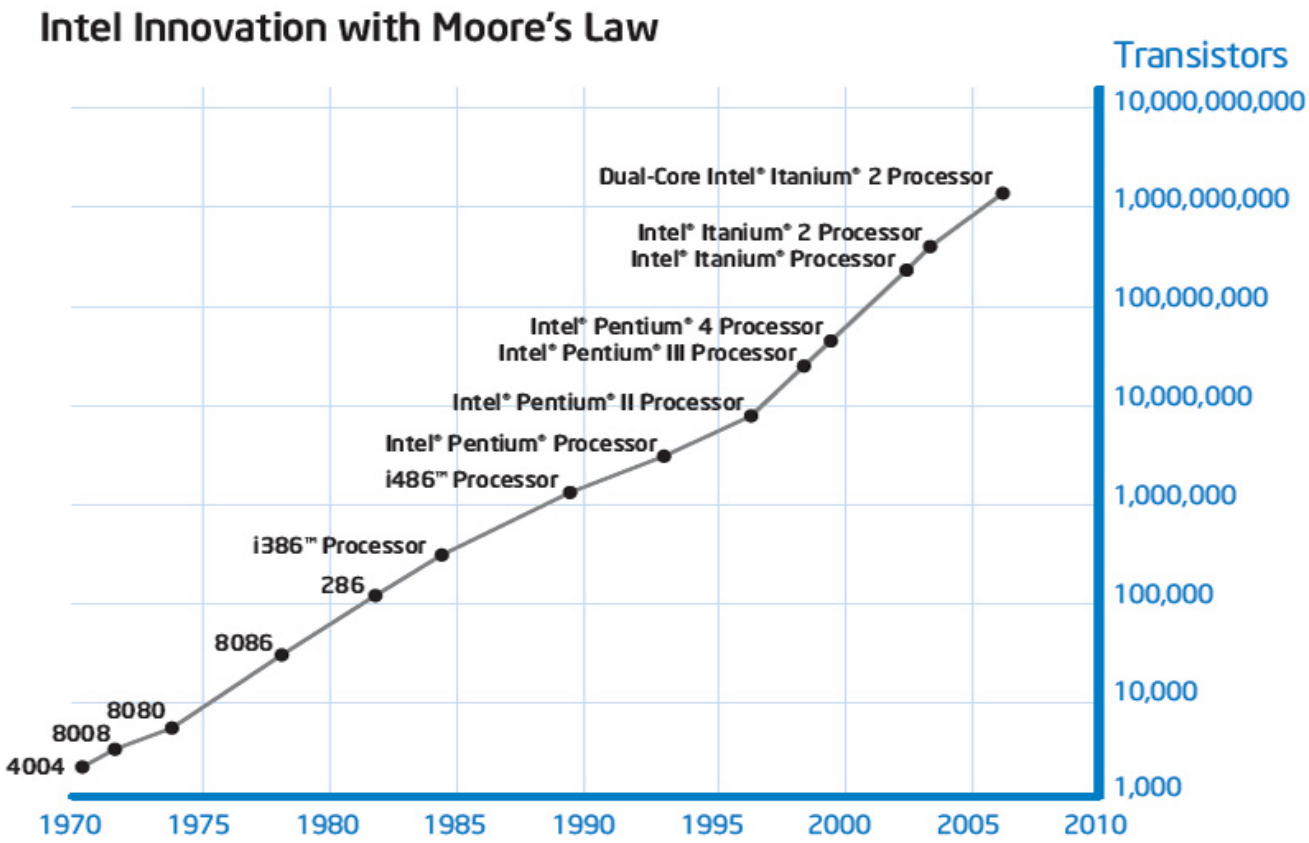
- Équilibrage des traitements sur les *processeurs*/*cœurs*
- Primitives du système exécutables en parallèle

Les logiciels n'en tirent pas encore pleinement bénéfice

- Des précautions/complications deviennent nécessaires
- Très pénalisantes en *monoprocesseur* !
  - On espère les compenser/dépasser en *multiprocesseurs* ...

Table des matières

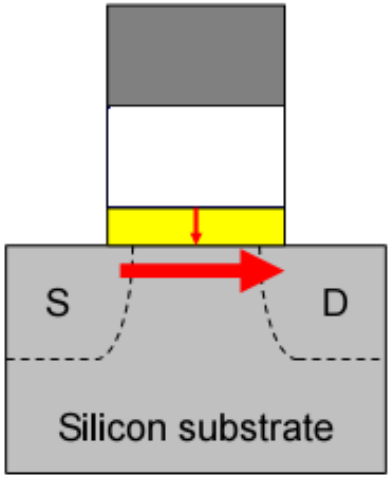
“Loi” de Gordon Moore



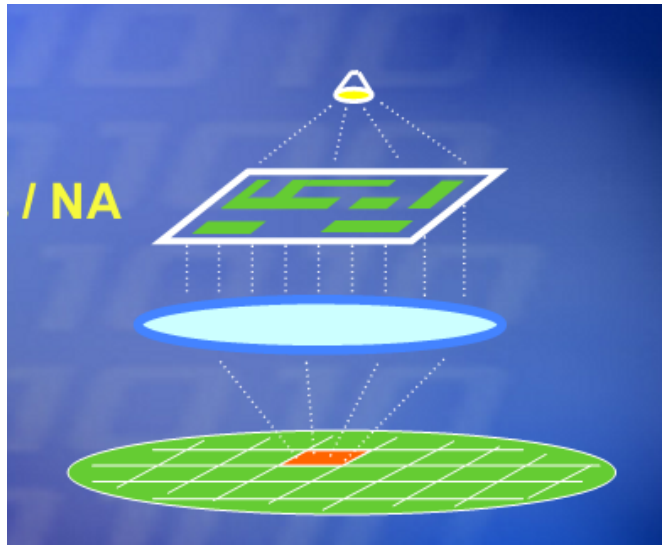
En pratique, nombre de transistors $\times$ 2 tous les 1,96 ans

- En 1979, *Intel 8086* : 29 000 transistors
- En 2006, *Intel Core2 Duo* : 291 000 000 transistors

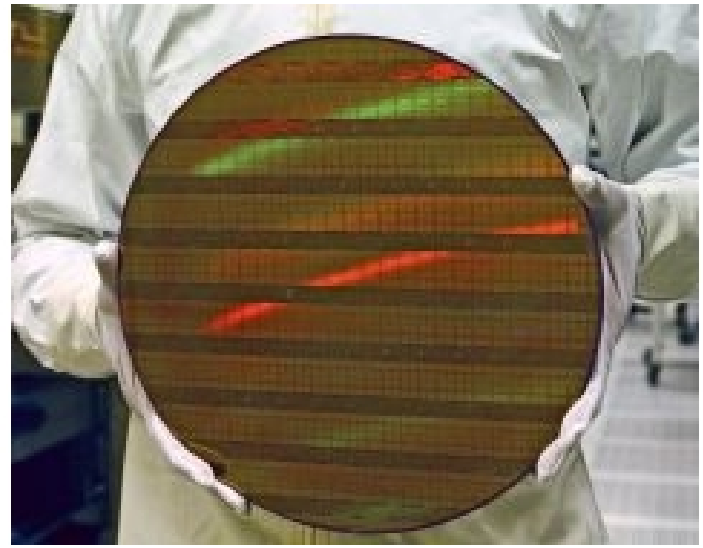
D'où viennent tous ces transistors ?



Un transistor



Photolithographie d'un "die"



Un wafer

Fabrication par *photolithographie*

- Semi-conducteur en *silicium*, isolant en *dioxyde de silicium*
- Une lampe à UV, un masque, une lentille convergente
  - Plusieurs itérations car plusieurs couches
- Un *wafer* ($\phi \simeq$ 30cm) contient de nombreux *dies* (qq centaines)

Plus de transistors ou de plus petits transistors ?

Fabriquer des transistors plus petits ?

- Temps de commutation plus courts → monter en fréquence
- Consommation électrique moindre → limiter l'échauffement

On peut en mettre plus sur la même surface de semi-conducteur !

- Plus de mémoire cache, de fonctionnalités évoluées ...
- C'est une conséquence *"heureuse"* de la miniaturisation

Des transistors toujours plus petits ?

Limite de la finesse de *photolithographie* régulièrement annoncée . . .

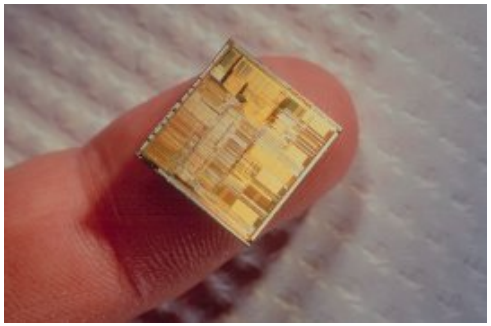
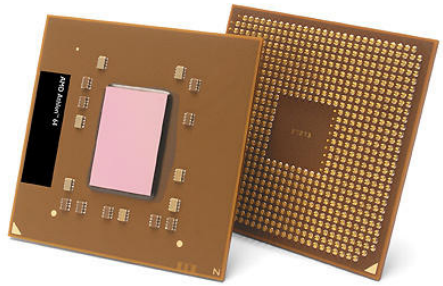
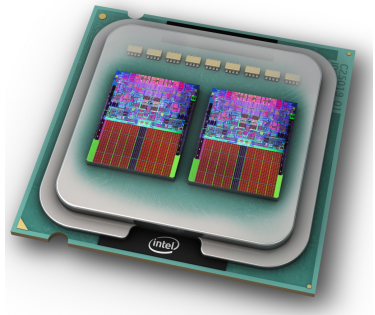
- Mais régulièrement dépassée . . .
  - Procédé industriel (2007) : 45nm avec un isolant à base d'*hafnium*
    - 32nm déjà en préparation . . .
- Précision de la *photolithographie*
  - Convergence de la lentille, longueur d'onde . . .
- Maîtriser l'isolation à si petite échelle !
  - Fuites de courant → interférences, échauffement

Limite physique ?

- Assemblage en laboratoire, à base de *silicium* → 15nm
  - *"Un transistor ou juste des atomes ?"*
- À base de *graphite* → moins de 10nm (avril 2008)

Des transistors toujours plus nombreux ?

Le *package* est bien plus gros que le *die* (connectique)

Le problème : le *yield* (proportion de *dies* corrects par *wafer*)

- Dépend de la fréquence de fonctionnement souhaitée
- ex :

Fréquence (GHz)	1,4	1,6	1,8	2,0	2,2	2,4	2,6
Yield (%)	100	97	90	80	65	45	10

Plus de transistors par *die* → défauts plus probables → moins bon *yield* !

- Il n'y a toutefois pas de barrière théorique à franchir !
- *"Juste"* améliorer la maîtrise du procédé de fabrication . . .

Processeurs d'usage général à court/moyen terme ?

Montée en fréquence peu probable

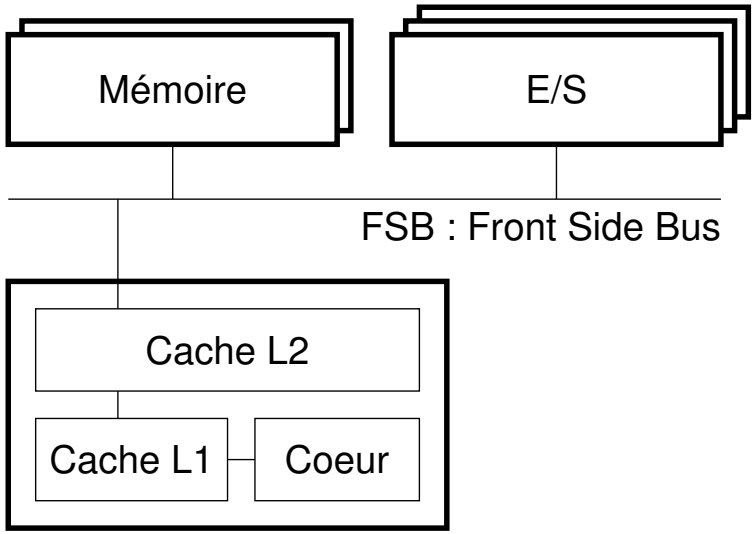
- La moindre amélioration dans cette direction est très difficile/coûteuse
- Au mieux, on peut espérer approcher des 4GHz
- À moins d'un procédé physique différent (long terme)

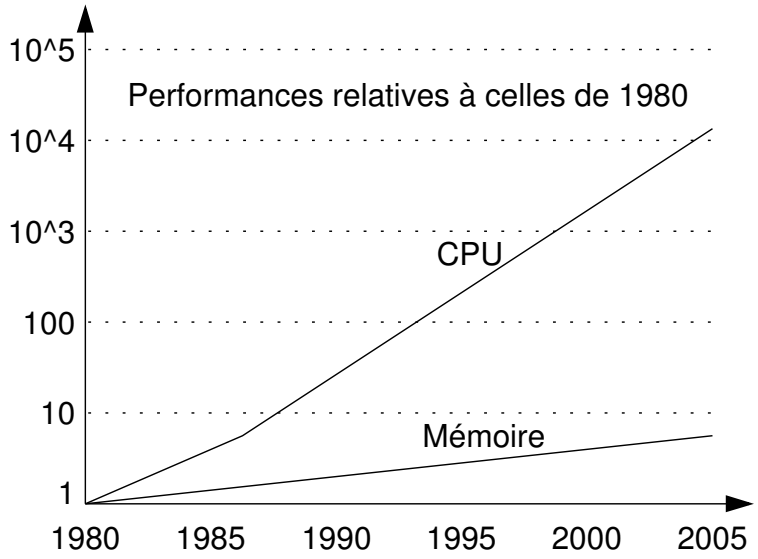
Augmentation du nombre de transistors très probable

- Maîtrise toujours meilleure du procédé de fabrication actuel
- Mémoire cache plus grande $\rightarrow$ forte amélioration des performances
- Plus de *cœurs* dans le même *die*

Table des matières

Machine monoprocesseur





Organisation hiérarchique de l'accès à la mémoire

- Mémoire cache de niveau 1 : très rapide (fréquence $= cœur$)
  - Faible capacité, données et instructions généralement séparées
- Mémoire cache de niveau 2 : rapide (fréquence $\propto cœur$)
  - Capacité moyenne
- Mémoire principale : relativement lente (fréquence $< FSB$)
  - Grande capacité, peu chère à produire
- ex : $Intel\ Pentium4$ à 3.8GHz, FSB 800MHz, $cache\ L1$ 16KB, $cache\ L2$ 2MB

Machine monoprocesseur

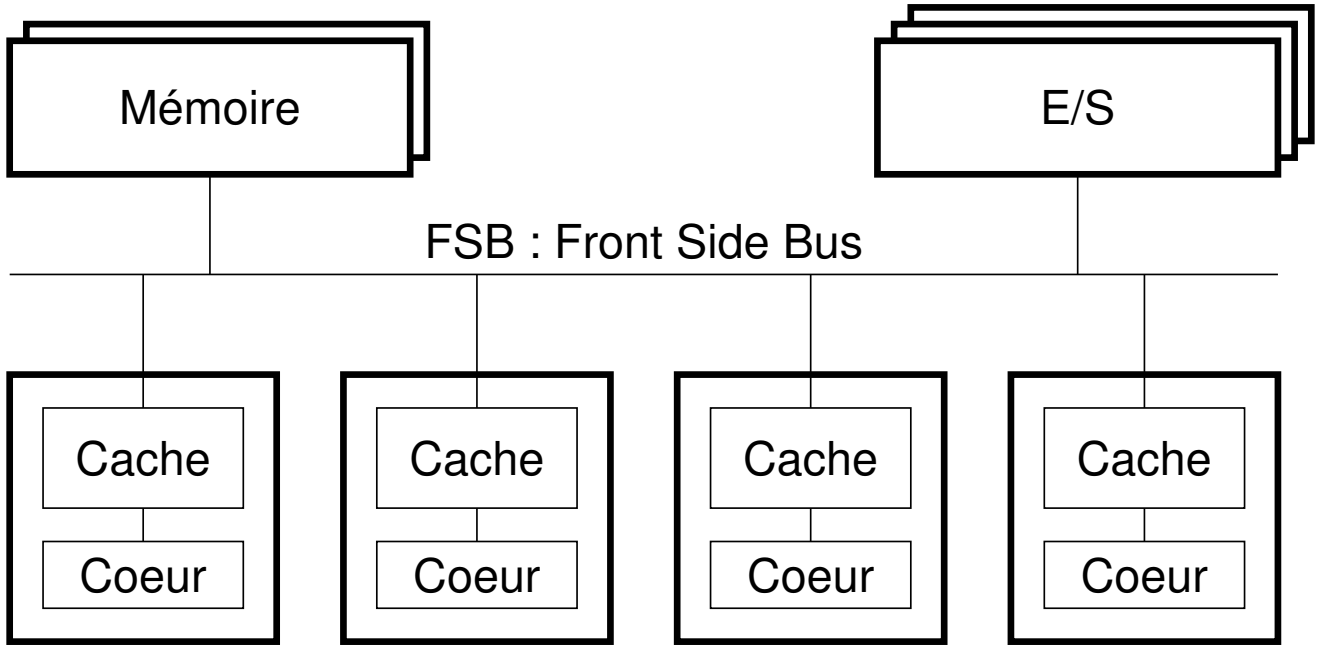
Principe des mémoires caches

- Localité temporelle : les mêmes données reserviront prochainement
  - Garder en cache les données dernièrement utilisées
- Localité spatiale : des données proches reserviront prochainement
  - Manipulation de *lignes de cache* (64, 128 ... octets)

Défaut de cache (*cache-miss*)

- La donnée souhaitée n'est pas présente dans le cache
  - Il faut charger la ligne depuis la mémoire
  - Très pénalisant : plusieurs dizaines (centaines) de cycles d'attente !
- Obligatoire lorsque la ligne n'a jamais été chargée
- En cas de dépassement de capacité (ou de collision)
  - Libérer des lignes pour en charger d'autres
  - Écriture éventuellement nécessaire dans la mémoire

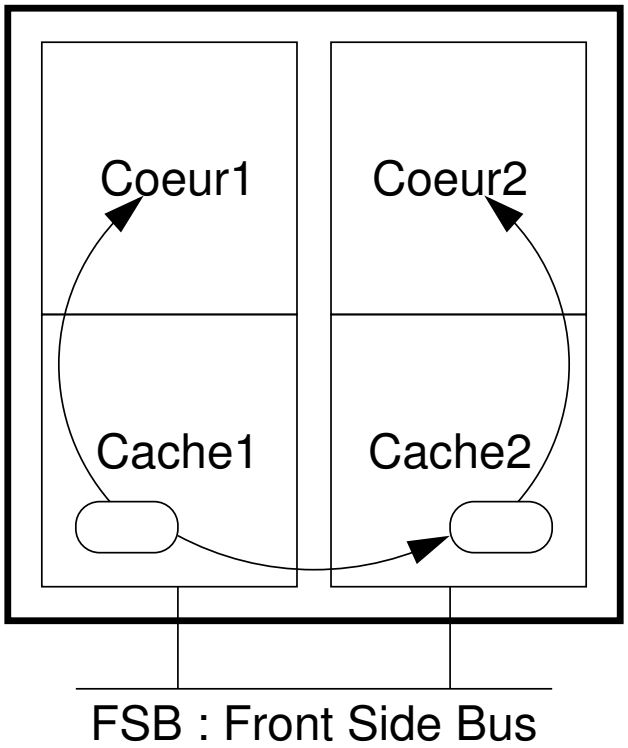
Machine multiprocesseurs

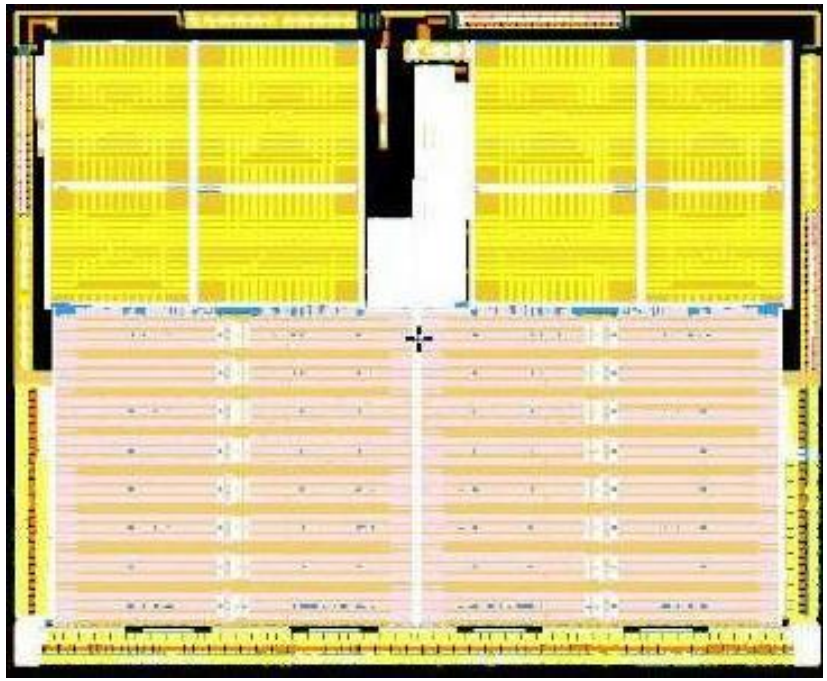


SMP-UMA : Symetric MultiProcessing — Uniform Memory Access

- Assurer la cohérence entre les mémoires caches !
  - *Modified* : L'entrée dans le cache est valide, mais la mémoire n'est pas à jour
  - *Owned* : D'autres caches possèdent cette entrée, mais la mémoire n'est pas à jour
  - *Exclusive* : Aucun autre cache ne possède cette entrée, et la mémoire est à jour
  - *Shared* : D'autres caches possèdent cette entrée, et la mémoire est peut-être à jour
  - *Invalid* : L'entrée dans le cache est invalide
- Synchronisation par le *FSB* !

Premiers processeurs multicœurs

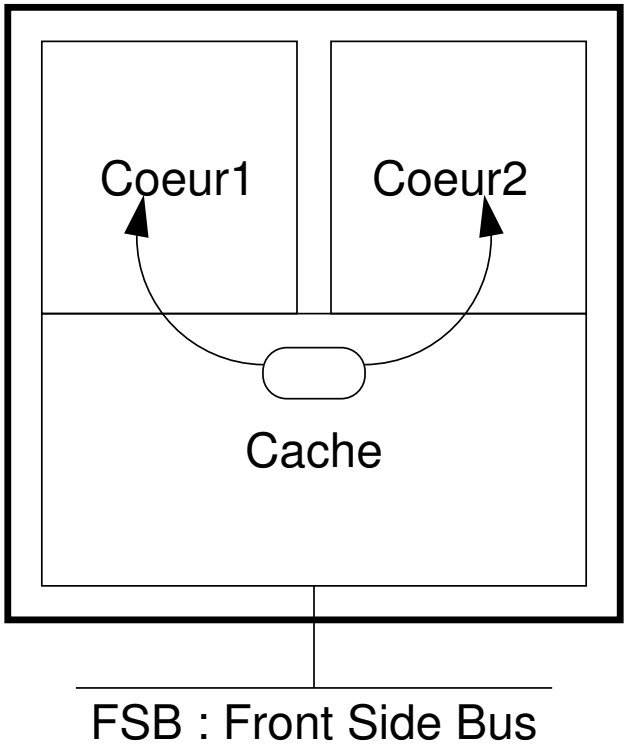


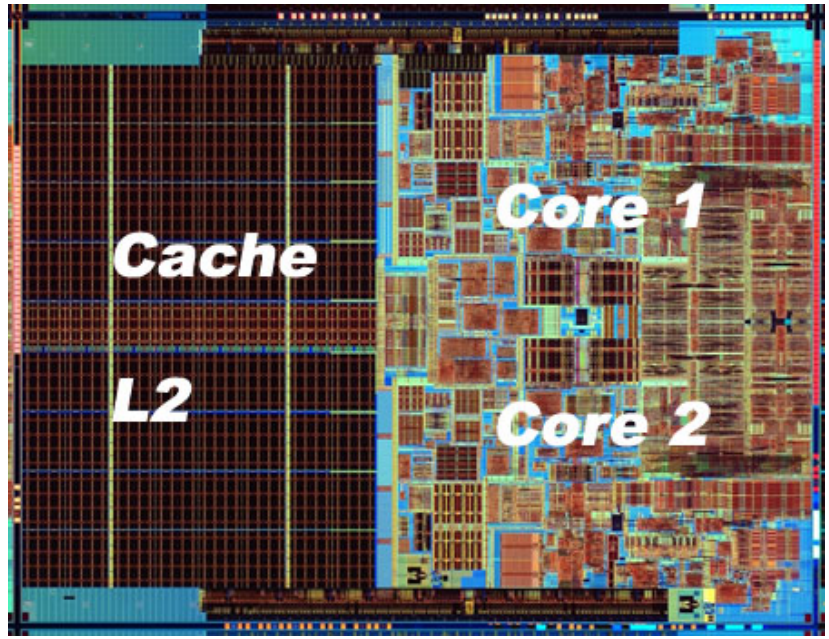


Les mémoires caches des *cœurs* sont distinctes

- Très similaire à la solution *multiprocesseurs*
- Communication privilégiée pour la cohérence des caches

Processeurs multicœurs modernes

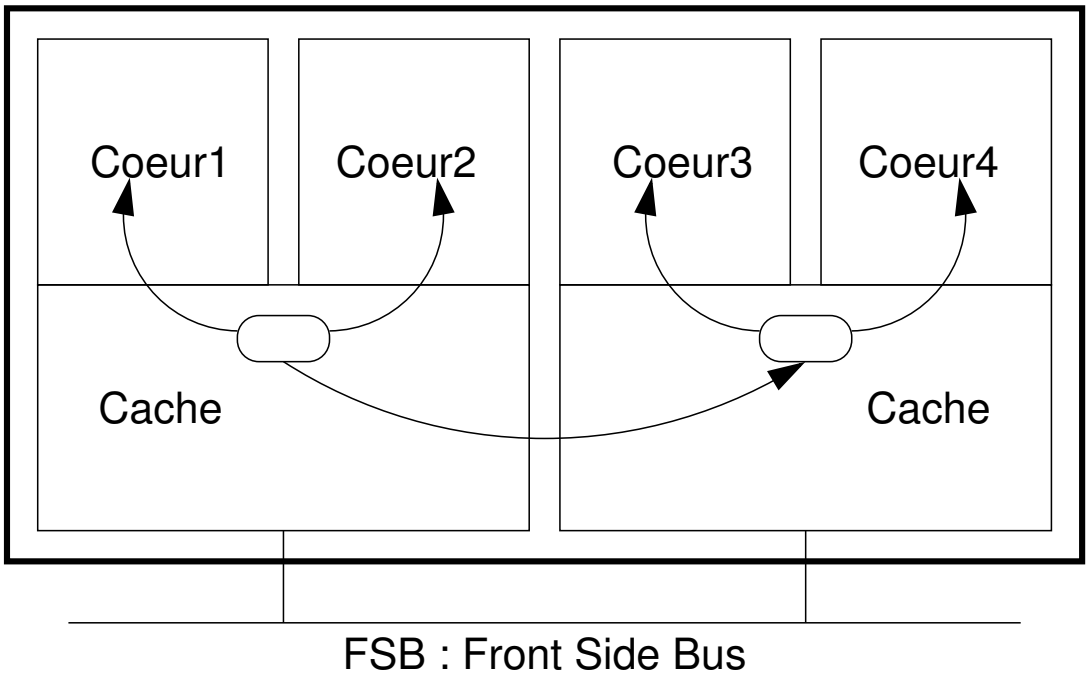


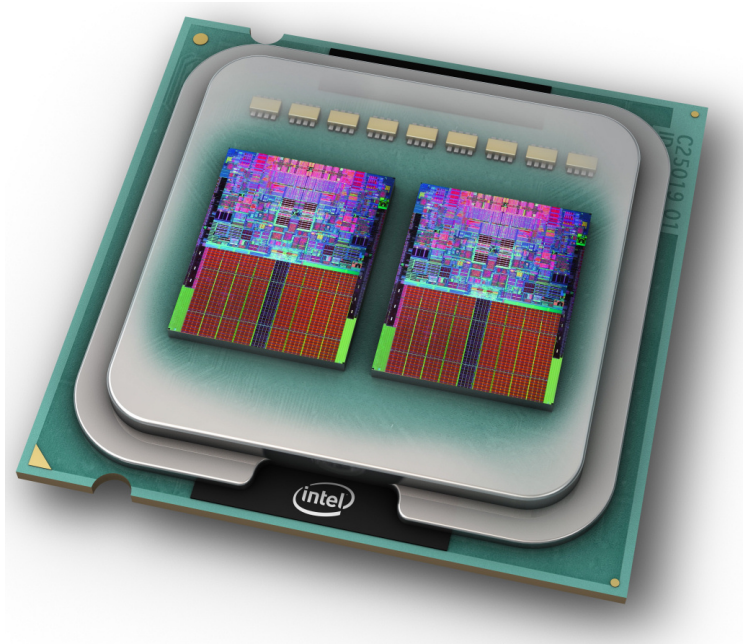


La mémoire cache est commune (mais synchronisée)

- Quantité totale librement partageable entre les $cœurs$
- Évite la recopie/duplication de données entre caches

Processeurs multicœurs modernes

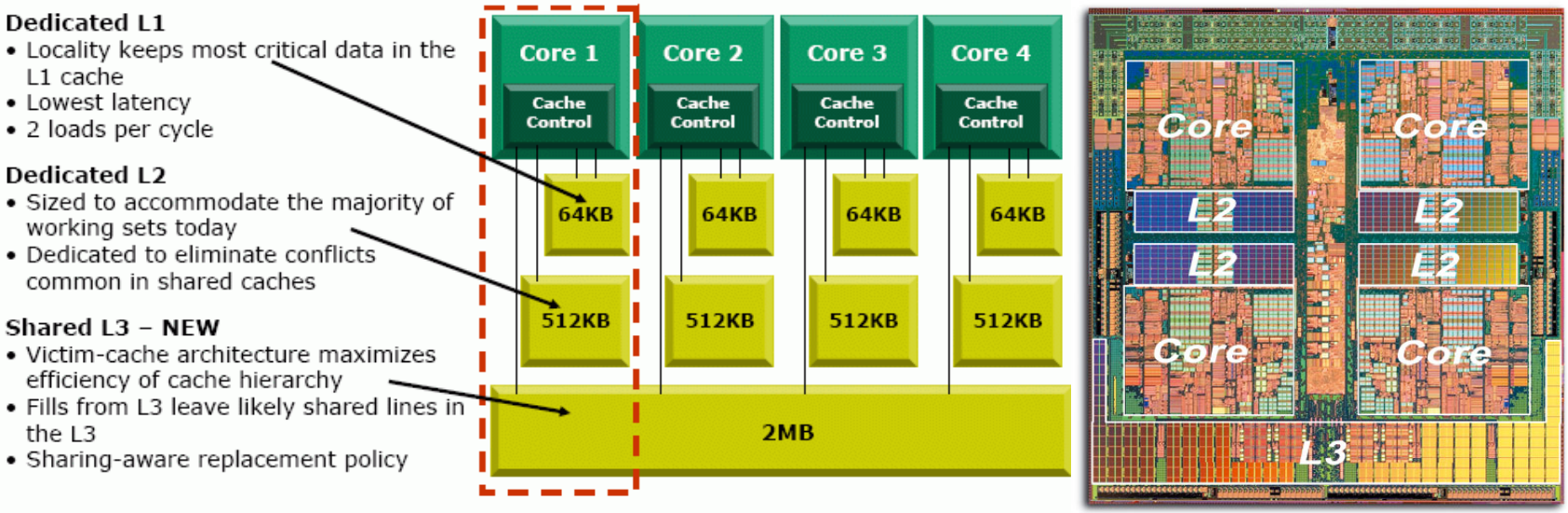




Solution *"prétendument" Quad-Core* d'*Intel*

- 2 × *Dual-Core* dans un même *package* (début 2008)
  - ex : 2 × *Intel Xeon Quad-Core* ≡ 4 × *Dual-Core*
- ex : *Intel QX9775* à 3.2GHz, *cache L2* 2 × 6MB, $2 \times 410 \times 10^6$ transistors

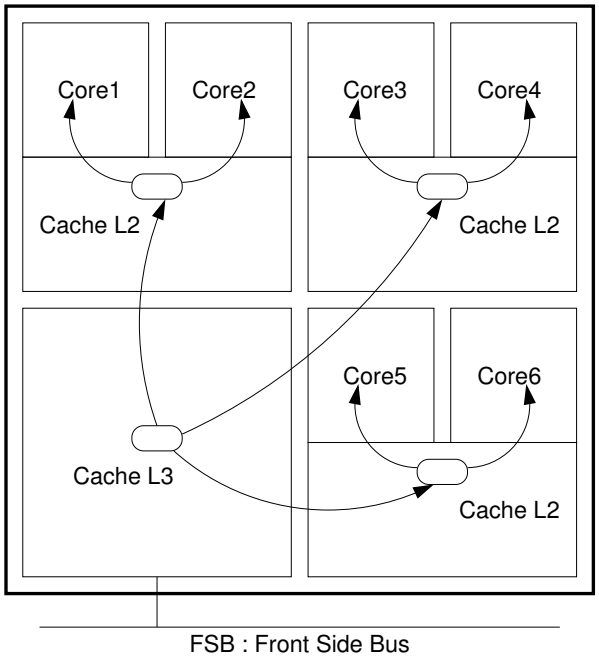
Processeurs multicœurs modernes

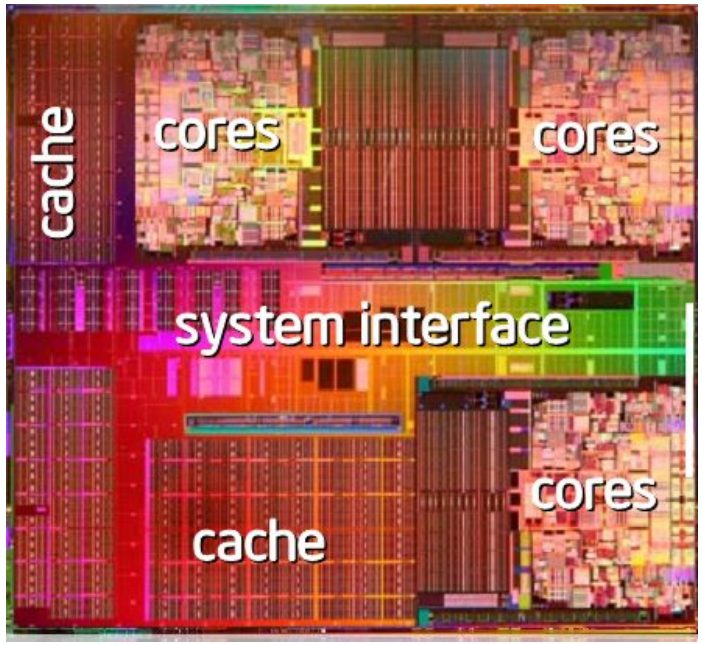


Solution *"réellement" Quad-Core* d'*AMD*

- Architecture complètement nouvelle (fin 2007)
  - Nouvelles gestions des branchements, des caches, de l'énergie ...
- Très séduisante en théorie ... décevante en pratique !
  - Beaucoup plus de mémoire cache et de GHz chez *Intel* (meilleure maîtrise du procédé de fabrication)
- ex : *AMD Phenom 9600* à 2.3GHz, *cache L3* 2MB, 463×10^6 transistors

Processeurs multicœurs modernes





Solution *Hex-Core* d'*Intel*

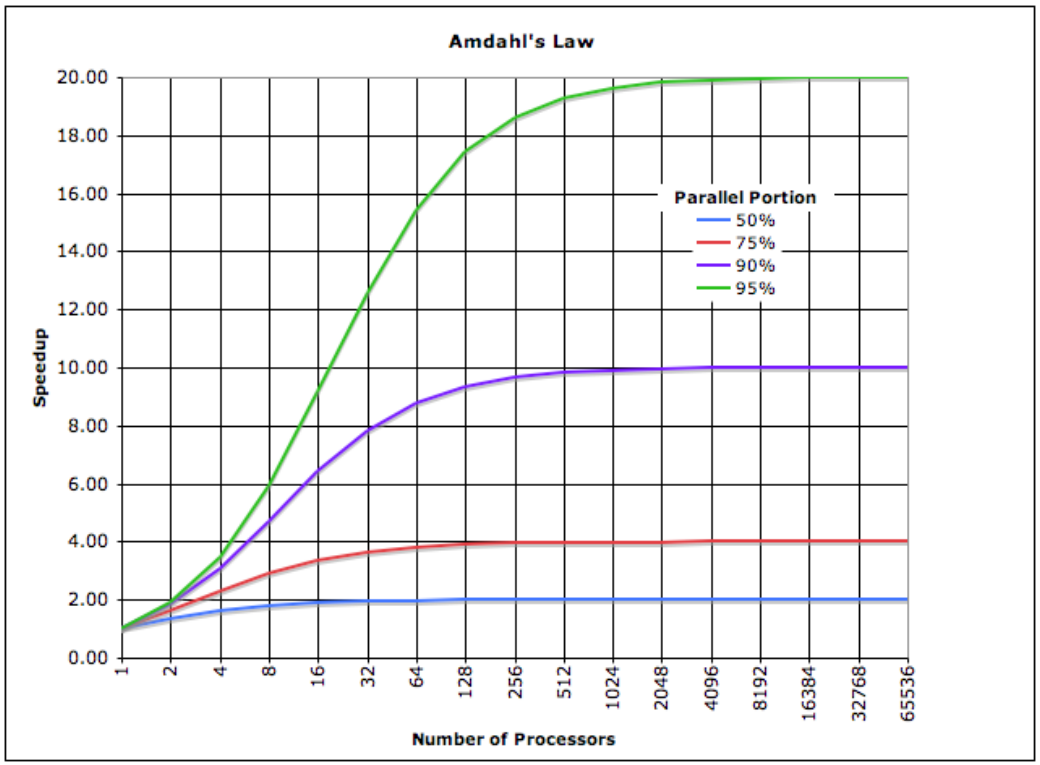
- Assembler des *Dual-Core* sous un cache commun (septembre 2008)
  - Le cache commun de niveau 3 n'est pas très rapide
  - Il limite l'impact des fautes de cache du niveau 2
- ex : *Intel Xeon X7460* à 2.66GHz, *L3* 16MB, *L2* 3 × 3MB, 1.9×10^9 transistors

Table des matières

Limites théoriques

"Loi" de Amdahl : *"la bouteille à moitié vide"*

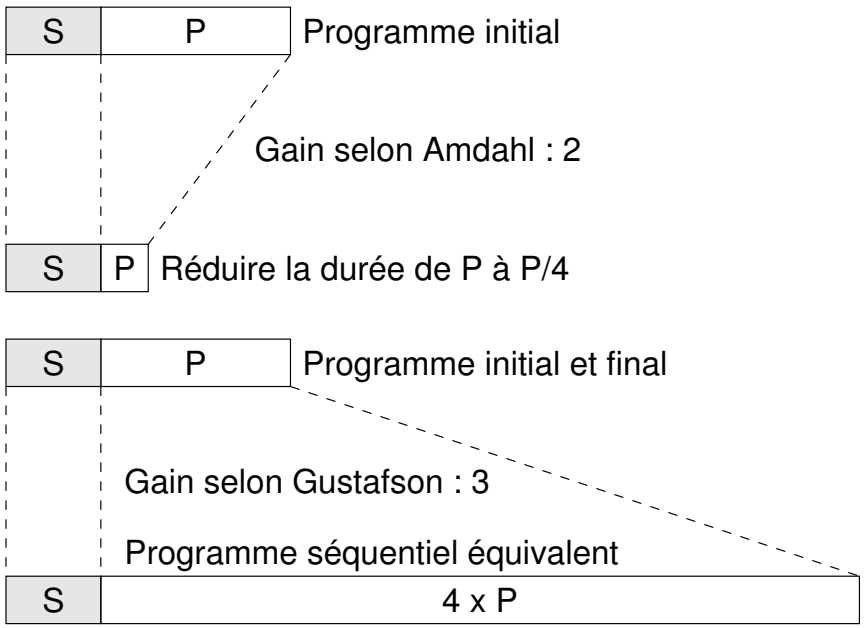
- Un programme contient des portions fondamentalement séquentielles
- Seules les portions parallélisables (p) bénéficient du parallélisme
- Accélération maximale pour N processeurs : $\lim_{N\to\infty} \frac{1}{(1-p)+\frac{p}{N}} = \frac{1}{(1-p)}$

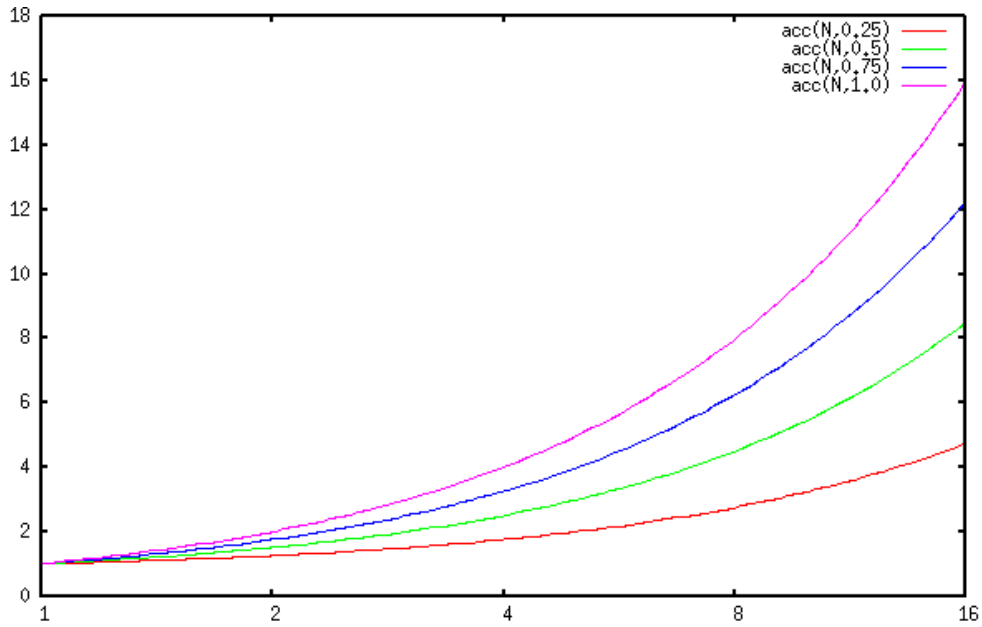


Limites théoriques

"Loi" de Gustafson : *"la bouteille à moitié pleine"*

- On peut choisir de conserver la même durée globale
- Le portions parallèles effectuent alors plus de traitements
- Accélération maximale pour N processeurs : $(1-p)+p\times N$





Synchronisation/exclusion mutuelle

Exemple : retrait d'un montant sur un solde

```
consulter le solde
si le montant est inférieur au solde
    le solde devient (solde-montant)
    donner le montant en billets
```

Thread A : montant=200

1. lire *solde* → 300
2. 200<=300 ? → oui
3. écrire *solde* → 100
4. donner 200

Thread B : montant=250

1. lire *solde* → 300
2. 250<=300 ? → oui
3. écrire *solde* → 50
4. donner 250

On a retiré 450 d'un solde de 300 et il reste 100 ou 50 !

→ Ce traitement devrait être une *section critique*

Synchronisation/exclusion mutuelle

Primitives *système* pour protéger une ressource (verrous, conditions . . .)

- Traitement suspendu tant que la ressource est utilisée par un autre
  - Relancé dès que la ressource redevient disponible
- Géré par le système d'exploitation
- Attente passive : n'utilise pas le processeur pendant ce temps

Primitives *utilisateur* pour protéger une ressource (*spin-locks* . . .)

- Tentatives répétées d'acquérir le droit d'accès à la ressource
- Le système d'exploitation n'est pas au courant
- Attente active : utilise le processeur *"inutilement"* pendant ce temps

Synchronisation/exclusion mutuelle

Un seul processeur : il y a nécessairement des commutations

- Un seul traitement est actif à la fois
- L'attente passive est très largement préférable

Nombre de traitements $\leq$ nombre de processeurs : pas de commutation

- Tous les traitements sont actifs à la fois
- L'attente active est préférable (sauf en cas de longue section critique)

Nombre de traitements $>$ nombre de processeurs : quelques commutations

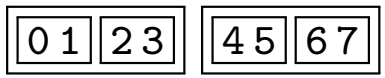
- On ne sait pas quels sont les traitements actifs
- Tenter une courte attente active puis une attente passive

Influence de la mémoire cache

Élément déterminant pour la performance

- Pour les systèmes monoprocesseur
  - Au moins aussi important que la fréquence
- Pour les systèmes multiprocesseurs/multicœurs
  - Très dépendant de la cohérence des caches ($MOESI$)
  - Très lié à la manière d'écrire les programmes
    - Mérite d'être étudié et compris

Machine d'expérimentation

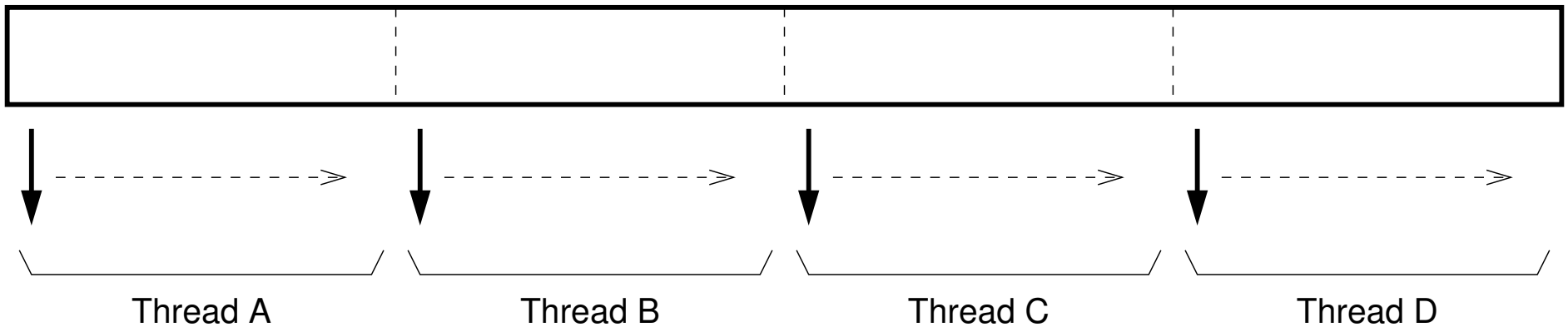
- Double *Intel Xeon Quad-Core E5405* à 2GHz
  - Chacun est constitué de deux *dies Dual-Core*
  - Chaque paire de cœurs dispose de 6MB de cache L2 commun
- Représentation utilisée pour nos expériences : 

  - Les deux processeurs physiques : 0123 et 4567
  - Les quatre *dies Dual-Core* : 01, 23, 45 et 67

Influence de la mémoire cache

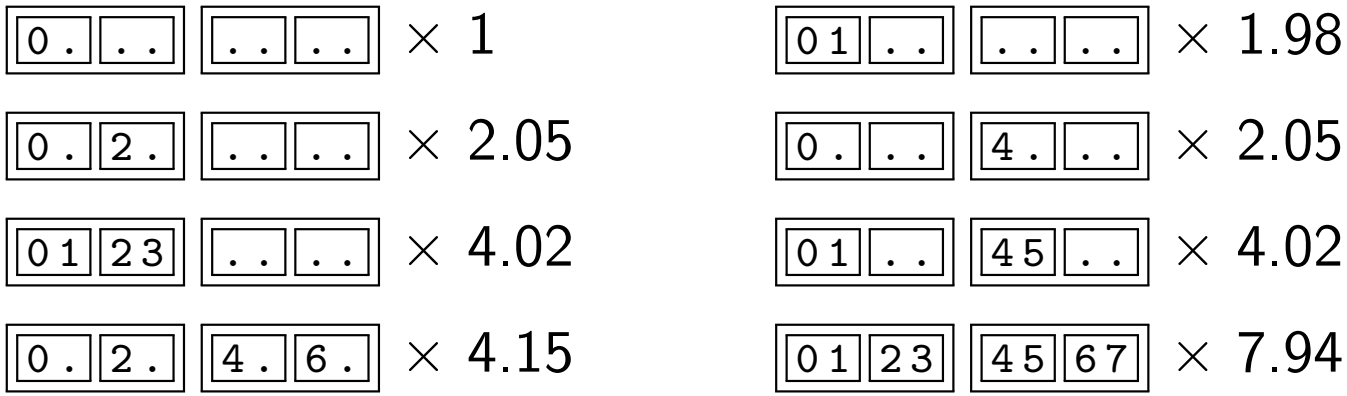
Expérience : lectures par blocs

- Lire tout le contenu d'un grand tableau pour en calculer la somme
  - Répéter de nombreuses de fois
- Chaque *thread* traite un bloc de données contiguës



Influence de la mémoire cache

Mesures : lectures par blocs tenant dans le cache L2

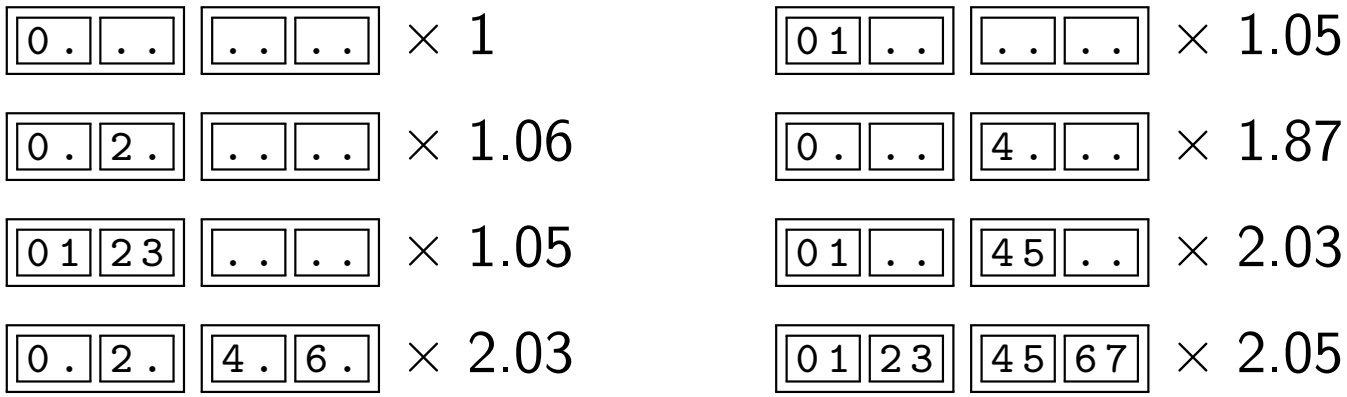


Interprétation

- Aucune faute de cache pendant le traitement
  - Données de chaque *thread* chargées une seule fois dans les caches
- Parallélisme toujours bénéfique
  - Quel que soit le placement des *threads* sur les cœurs

Influence de la mémoire cache

Mesures : lectures par blocs dépassant le cache L2



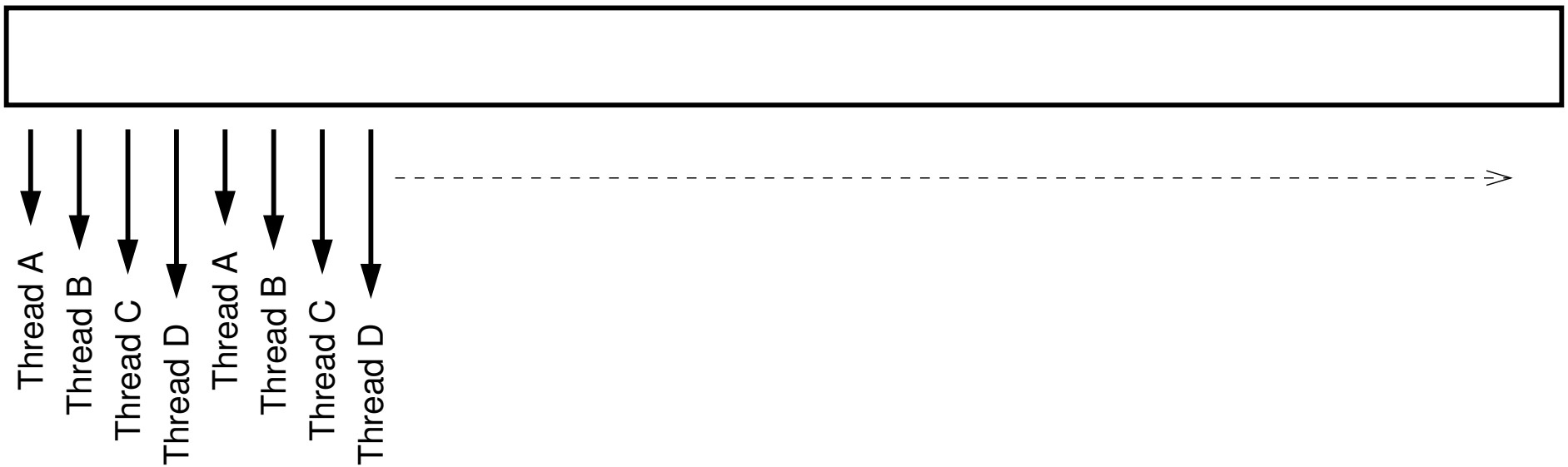
Interprétation

- Fautes de cache permanentes
  - Évacuer les données précédentes pour charger les nouvelles
- Parallélisme peu intéressant sauf pour deux processeurs distincts
  - Chaque cœur utilise des données distinctes
  - Accès mémoire simultanés sur les deux processeurs physiques ?

Influence de la mémoire cache

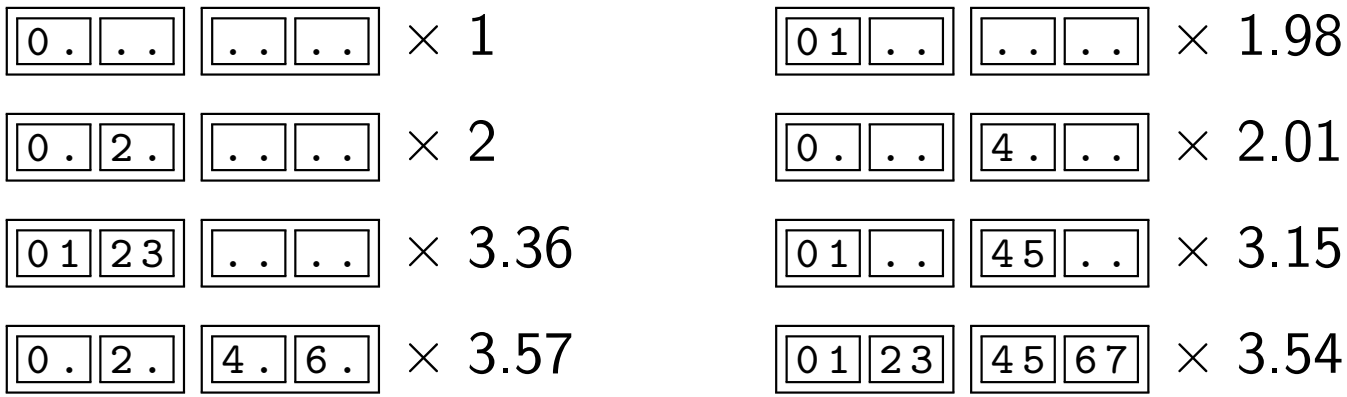
Expérience : lectures entremêlées

- Lire tout le contenu d'un grand tableau pour en calculer la somme
  - Répéter de nombreuses de fois
- Chaque *thread* parcourt tout le tableau (en sautant des éléments)



Influence de la mémoire cache

Mesures : lectures entremêlées tenant dans le cache L2

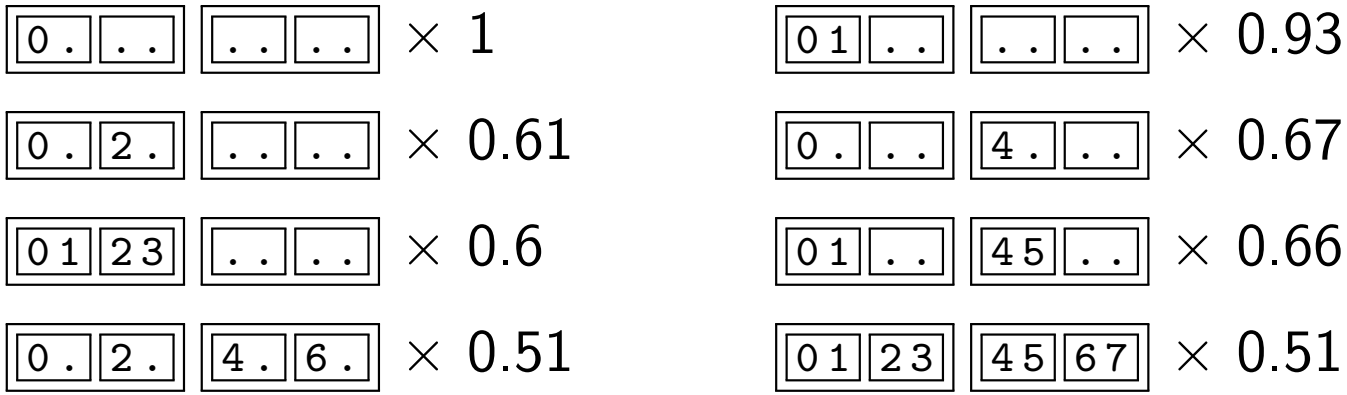


Interprétation

- Aucune faute de cache pendant le traitement
  - Données de chaque *thread* chargées une seule fois dans les caches
- Parallélisme peu bénéfique au delà de deux *threads*
  - Les lignes de cache sont communes (phénomène de *faux partage*)
  - Synchronisation des lignes de cache même en lecture ?

Influence de la mémoire cache

Mesures : lectures entremêlées dépassant le cache L2



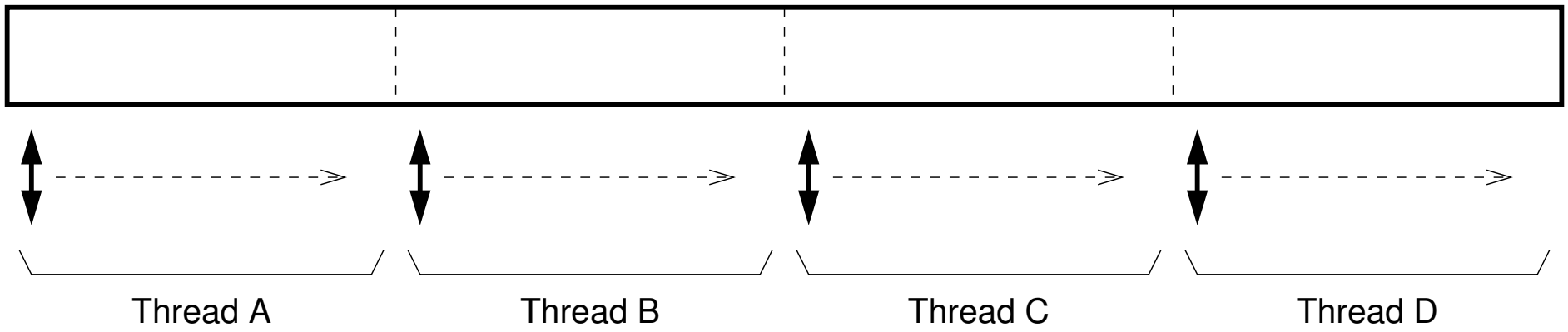
Interprétation

- Fautes de cache permanentes
  - Évacuer les données précédentes pour charger les nouvelles
- Parallélisme généralement néfaste (au mieux inutile)
  - Cumule les fautes de cache et les synchronisations en lecture ?

Influence de la mémoire cache

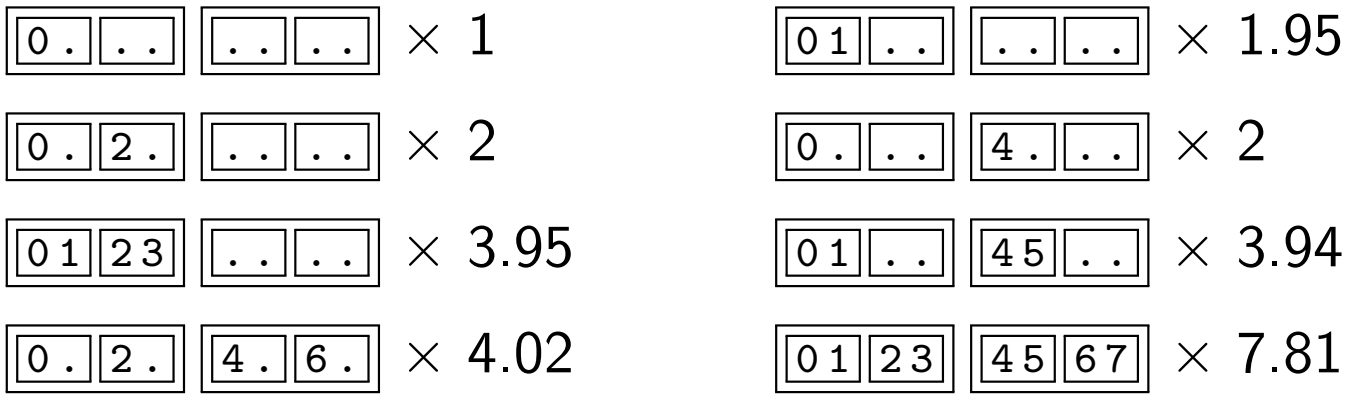
Expérience : lectures/écritures par blocs

- Remplacer chaque élément d'un grand tableau par son opposé
  - Répéter de nombreuses de fois
- Chaque *thread* traite un bloc de données contiguës



Influence de la mémoire cache

Mesures : lectures/écritures par blocs tenant dans le cache L2

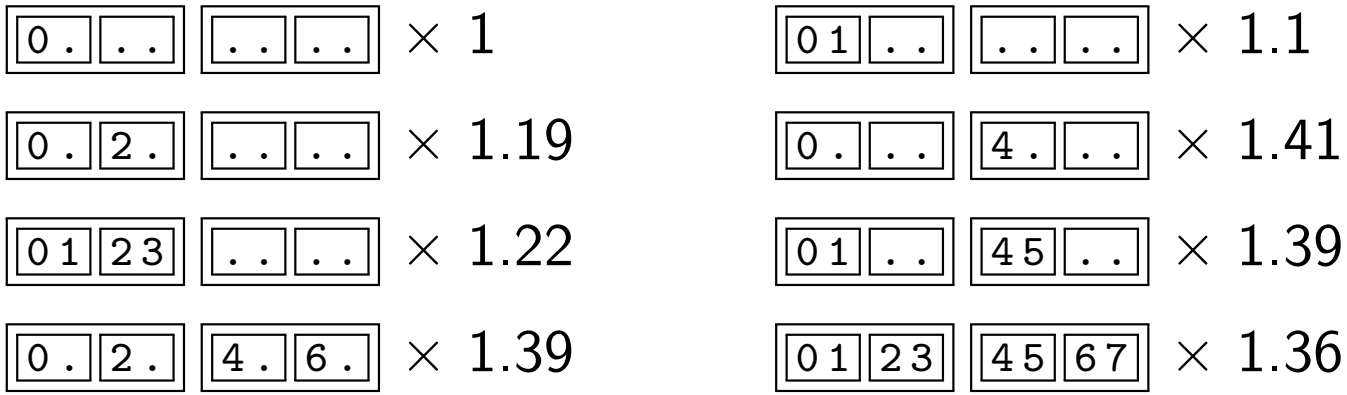


Interprétation

- Aucune faute de cache pendant le traitement
  - Données de chaque *thread* chargées une seule fois dans les caches
  - Les écritures ne concernent pas les autres cœurs → pas d'invalidation
- Parallélisme toujours bénéfique
  - Quel que soit le placement des *threads* sur les cœurs

Influence de la mémoire cache

Mesures : lectures/écritures par blocs dépassant le cache L2



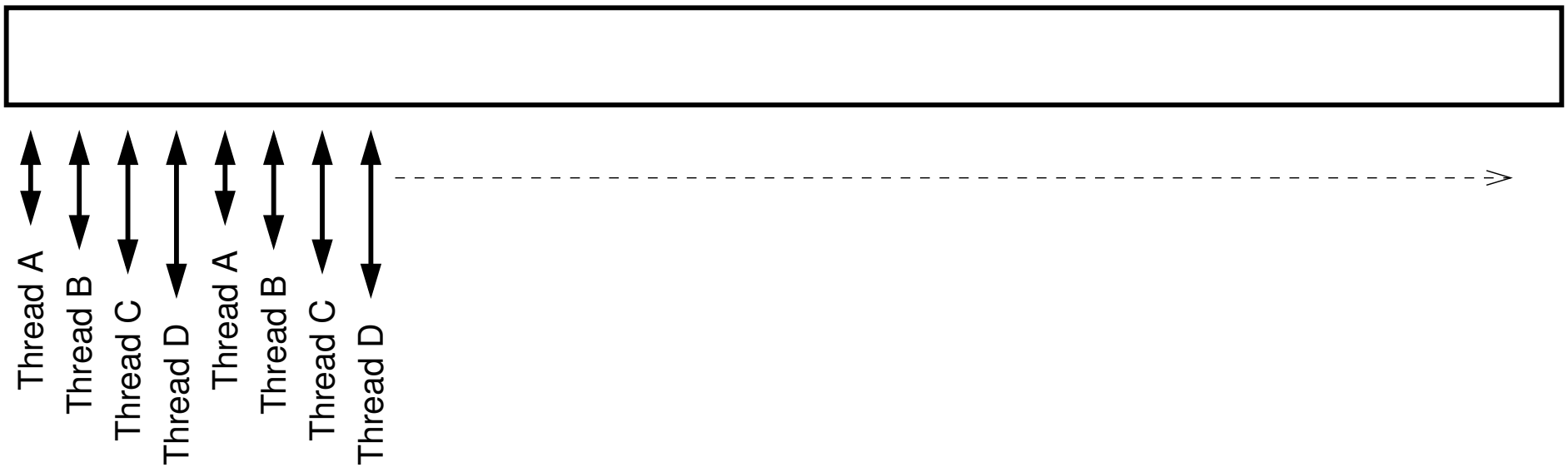
Interprétation

- Fautes de cache permanentes
  - Évacuer les données précédentes pour charger les nouvelles
    - Nécessite la mise à jour de la mémoire !
  - Les écritures ne concernent pas les autres cœurs → pas d'invalidation
- Accès mémoire simultanés pour deux processeurs physiques distincts
- Parallélisme peu intéressant pour ce type de problème

Influence de la mémoire cache

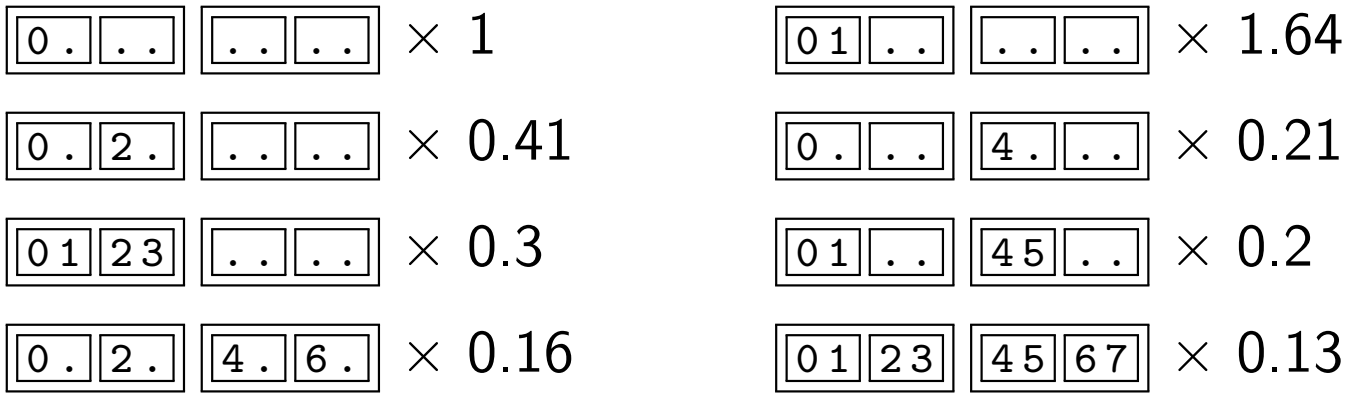
Expérience : lectures/écritures entremêlées

- Remplacer chaque élément d'un grand tableau par son opposé
  - Répéter de nombreuses de fois
- Chaque *thread* parcourt tout le tableau (en sautant des éléments)



Influence de la mémoire cache

Mesures : lectures/écritures entremêlées tenant dans le cache L2

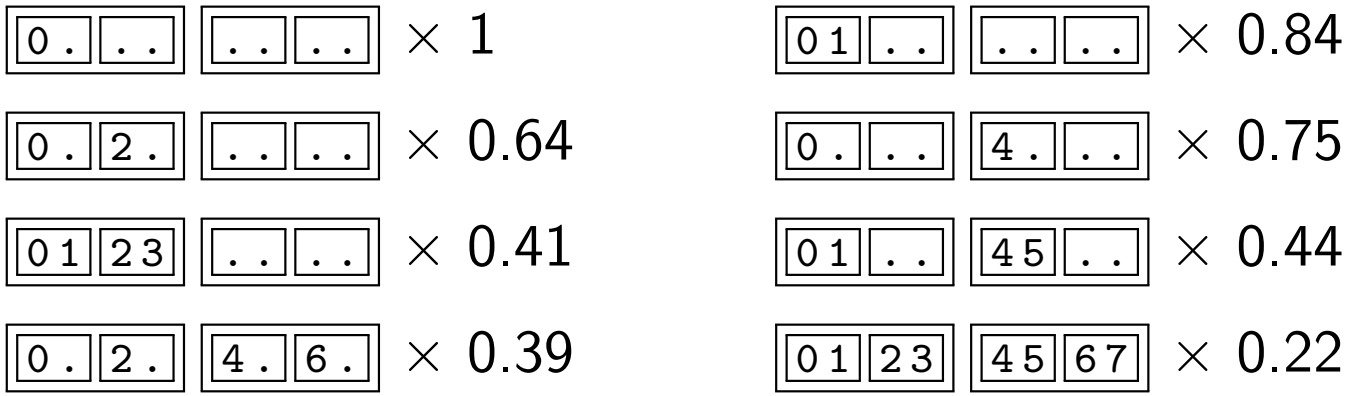


Interprétation

- Fautes de cache permanentes
  - Chaque *thread* invalide le cache des autres (*faux partage*)
- Seule la solution à cache $L2$ commun est assez intéressante
  - Les caches $L1$ s'invalident tout de même mutuellement
- Parallélisme très néfaste si les caches sont distincts
  - L'*"éloignement"* des caches accentue le problème

Influence de la mémoire cache

Mesures : lectures/écritures entremêlées dépassant le cache L2



Interprétation

- Fautes de cache permanentes
  - Chaque *thread* invalide le cache des autres (*faux partage*)
  - Évacuer les données précédentes pour charger les nouvelles
- Parallélisme toujours néfaste
  - L' *"éloignement"* des caches accentue le problème

Influence de la mémoire cache

Comportement très sensible à la limite du cache

- Expérience : lectures par blocs d'un tableau de 16MB
  - L'ensemble (16MB) ne tient pas dans un seul cache (6MB)
- Parallélisation sur les 8 cœurs (4 $Dual\text{-}Core$)
  - Chaque cache (6MB) peut contenir ses deux blocs (2×2MB)
- Le gain de performance est plus que linéaire ! $\rightarrow \times$ 30 !
  - Ce seuil n'existe pas s'il y a un unique cache commun ...

Le cache vis-à-vis de l'exclusion mutuelle

- Verrou $\equiv$ variable globale ($\forall$ implémentation)
  - À chaque manipulation du verrou $\rightarrow$ une faute de cache
- Expérience : 8 cœurs utilisent une variable locale/globale $\rightarrow \div$ 11 !
  - La différence ne peut que s'amenuiser avec un cache commun

Quelques recommandations

De manière générales

- Éviter l'entremêlement des données (*faux partages*)
  - Attention toutefois à la capacité des mémoires caches !
- Travailler sur une copie locale des données
  - De rares synchronisations globales
  - Synchronisation *"relachée"*
- Préférer les lectures aux écritures !
  - ex : lecture avant de tenter *test-and-set* sur un *spin-lock*

Quelques recommandations

Pas moins de *threads* actifs que de cœurs

- Ressources inemployées sinon !
- Pas de traitements bloquants
  - Les placer dans des *threads* supplémentaires si besoin
    - Toujours bloqués, se débloquent de temps en temps
- Sections critiques peu nombreuses et de très courte durée
  - Sinon les autres *threads* attendent !

Pas plus de *threads* actifs que de cœurs

- Ne pas risquer une suspension pendant une section critique
  - Les autres *threads* devraient attendre sa relance !
- Ne pas évincer du cache les données d'un *thread* par celles d'un autre
  - Le surcoût de changement de contexte est négligeable

Quelques recommandations

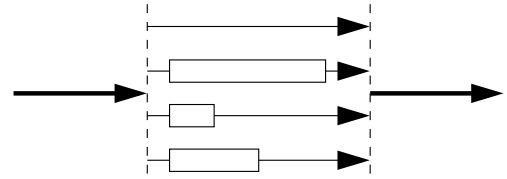
Placer précisément les *threads* sur les processeurs ?

- Le système équilibre automatiquement entre les cœurs
  - Ne provoque pas de migration intempestive
  - Ne prend pas en compte la *"distance"* entre les caches
- Saurions-nous faire mieux explicitement ?
  - Cas d'études caricaturaux ici
  - Difficile de catégoriser toutes les phases d'un problème pratique

Quelques recommandations

Parallélisme de tâche

Chaque traitement distinct est calculé dans un *thread*

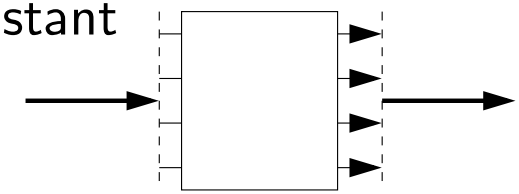


- ☺ Solution la plus intuitive au premier abord
- ☹ Cadence de la tâche la plus lente
- ☹ Pas de passage à l'échelle : que faire avec plus de cœurs ?

Parallélisme de données

Subdiviser chaque traitement en portions parallélisables

- ☹ Solution peu intuitive au premier abord
  - *"Aller chercher"* le parallélisme dans le code existant
  - *"Penser parallèle"* pour le code à produire



- ☺ Pas de temps mort, *"plein-emploi"*
- ☺ Bon passage à l'échelle : subdivision plus fine si plus de cœurs

Table des matières

Threads natifs

Le seul moyen mis à disposition par le système

- `CreateThread()` sous *Window$*, API *pthread* partout ailleurs
  - Pas de difficulté de mise en œuvre, quelques types et fonctions
- Primitives de synchronisation variées
  - Mises à jours de de la norme *POSIX 1003.1*
  - *Window$ Vi$ta* comble des lacunes

Un *thread* ≡ *"un appel de fonction dont on n'attend pas le retour"*

- Création d'une pile d'exécution pour chaque *thread*
- *"Assembleur"* du parallélisme
  - Gérer tous les détails d'organisation dans l'application
  - Niveau de contrôle assez fin
- Tendance à un découpage *"grossier"* du code à paralléliser

Threads natifs

```
void * myTask(MyObject *obj)
{
... travailler sur obj ...
return NULL;
}

int main(void)
{
...
pthread_t th1,th2;
MyObject *o1= ... ;
MyObject *o2= ... ;
pthread_create(&th1,NULL,&myTask,o1); /* executer myTask(o1) en // */
pthread_create(&th2,NULL,&myTask,o2); /* executer myTask(o2) en // */
... continuer un autre traitement ...
}
```

OpenMP

Directives de parallélisation en C (ou $Fortran$)

- Objectif : paralléliser la moindre portion de code
- `#pragma` influançant le compilateur
  - Nécessite un compilateur spécialisé (GCC depuis 4.2, $Intel\text{-}ICC$ ...)
- Le code doit fonctionner si ces directives ne sont pas supportées
  - Exécution séquentielle avec un compilateur classique

Gestion transparente des $threads$

- Un $pool$ de $threads$ est utilisé automatiquement
  - Pas de créations explicites
  - $\text{“}Découpage\text{”}$ transparent du code en fonctions anonymes
- Directives de synchronisation
- Quelques moyens d'exercer une influence
  - Variables d'environnement, API de contrôle (`#ifdef _OPENMP`)

OpenMP

```
...
int var1,var2,var3;
...
#pragma omp parallel private(var1,var2) shared(var3)
  {
#pragma omp sections
    {
#pragma omp section
      {
      ... une première section de code en // ...
      }
#pragma omp section
      {
      ... une seconde section de code en // ...
      }
    }
  }
...
```

OpenMP

```
...
#pragma omp parallel for private(i) shared(x,y)
for(i=0;i<nb;++i)
  {
  y[i]=f(x[i]);
  }
...


...
sum=0;
#pragma omp parallel for private(i,tmp) shared(x) reduction(+:sum)
for(i=0;i<nb;++i)
  {
  tmp=f(x[i]);
  sum+=tmp*tmp;
  }
...
```

Threading Building Blocks

Bibliothèque *C++* pour le parallélisme (*Intel*, license *GPL*)

- Utilisation des *templates* dans l'esprit *STL*
- Dissimule les détails de la gestion des *threads*
  - Objectif semblable à *OpenMP* : paralléliser tout ce qui peut l'être
  - Mais se contente d'un compilateur *"standard"*

Raisonnement en terme de tâches divisibles

- Division récursive des tâches pour occuper tous les *threads*
  - Préférence à la localité dans l'attribution des tâches (cache !)
  - *"Vol"* de tâches par les *threads* innoccupés
- Tâches $\equiv$ *foncteurs C++* $\rightarrow$ lourdeur d'écriture !
- La division repose sur des objets *intervalles* divisibles
- Algorithmes d'application des tâches aux *intervalles*
  - `parallel_for<T>()`, `parallel_reduce<T>()`, `parallel_sort<T>()`, ...

Threading Building Blocks

Très grande variété de primitives de synchronisation

- En mode *utilisateur* et encapsulation des primitives *système*
- Facilités de mise en œuvre (déverrouillage automatique)
- L'usage en est découragé !
  - Ne pas bloquer les *threads*
  - Les *intervalles* divisibles ne requierent pas de synchronisation
- Opérations atomiques non bloquantes
  - Affecter, incrémenter, permuter des types de base

Threading Building Blocks

Autres particularités

- Conteneurs réentrants
  - Pas une simple synchronisation des conteneurs standards
  - Accès simultané à des données distinctes d'un même conteneur
- Allocateurs de mémoire
  - Efficace en parallèle et minimisant les risques de *faux-partage*
  - Peuvent remplacer les `malloc/free/new/delete` de *C/C++*
- Démarche globale très cohérente
  - Très bien expliqué et justifié dans le livre de référence
    "Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism"
    James Reinders — O'Reilly, 2007

Threading Building Blocks

```
float f(float x) { ... }
void applyF(float *x,float *y,size_t nb) { while(nb--) y[nb]=f(x[nb]); }

struct f_Functor
{
float *_x,*_y;
f_Functor(float *x,float *y) : _x(x), _y(y) {}
void operator()(const blocked_range<size_t> &r) const
  {
  for(size_t i=r.begin();i!=r.end();++i)
    _y[i]=f(_x[i]);
  }
};

void parallelApplyF(float *x,float *y,size_t nb)
{
parallel_for(blocked_range<size_t>(0,nb), // intervalle complet
             f_Functor(x,y),              // traitement à effectuer
             auto_partitioner());         // politique de division
}
```

Threading Building Blocks

Exemple du livre : compter les occurences des mots d'un texte

- Nécessite une table associative : $mot \rightarrow occurences$

!!☹ `map` STL + `Mutex` $Win32 \rightarrow \times$ 0.1 (1→4 $threads$)

☹ `map` STL + `CRITICAL_SECTION` $Win32 \rightarrow \times$ 0.75 (1→4 $threads$)

≃☺ `concurrent_hash_map` $TBB \rightarrow \times$ 1.5 (1→4 $threads$)

Exemple du livre : le moteur de physique ODE

- Adaptation du code existant $\neq$ réécriture complète
  - Un jour de développement (concepteurs de TBB)

≃☺ Simulation de 400 objets simples $\rightarrow \times$ 1.29 (1→4 $threads$)

Comparaison des solutions

Threads natifs

- Contrôle fin
- Compilateur standard
- Moyennement invasif
- Primitives de synchronisation
- Granularité assez grossière
- Équilibrage statique

OpenMP

- Peu de contrôle
- Compilateur spécifique
- Peu invasif
- Primitives de synchronisation
- Granularité fine
- Équilibrage dynamique

Threading Building Blocks

- Peu de contrôle
- Compilateur *C++* standard
- Très invasif
- Synchro., conteneurs, alloc.
- Granularité fine
- Équilibrage très dynamique

Table des matières

Bilan

Aspect matériel

- Orientation franchement prise mais encore jeune (début 2008)
  - Quelques cœurs, architecture hétérogène
- Gain comparable aux instructions vectorielles
  - Très bien gérées par le compilateur *Intel*

Aspect logiciel

- Fonctionne déjà bien en multiprocessus
  - Indépendance des données, pas de synchronisation
- Difficile de paralléliser une application (habitudes)
  - Documentation des outils → *patterns* évidents *"et en vrai ?!?!?!"*
  - Ça n'apportait pas grand chose jusqu'à maintenant
- On ne gagne pas à tous les coups !
  - On peut même perdre beaucoup ! (variables communes, *faux partage*)
  - Il reste des traitements fondamentalement séquentiels (Amdahl)

Perspectives

Aspect matériel

- Tendance annoncée : plus de cœurs, plus de mémoire cache
  - Cache commun $\rightarrow$ homogénéisation du comportement
  - Pour 2009, *Intel Nehalem*, 2–8 cœurs, cache *L3* 4–24MB commun
- Évolution d'autres tendances
  - Toujours plus d'instructions vectorielles
    - Exploitation automatique par les compilateurs
  - Programmation des *Graphics Processing Units*
    - Massivement parallèle mais encore trop spécialisé
    - *NVIDIA* propose le *toolkit CUDA* pour leurs *GPUs* récents
    - Le futur *Intel Larrabee* se veut également plus flexible

Perspectives

Aspect logiciel

- *Intel* met le paquet !
  - Outils : compilateur, bibliothèques, mise au point ...
  - Documents : livres, articles techniques
- Pas de solution générique !
  - Encourager la synchronisation relachée
  - Distinguer les traitements bloquants/non-bloquants
  - S'imposer un *framework* spécifique ?
    - À quel prix ? surcoût en calcul, perte de confort ...