

Algorithmique et Optimisation Discrète

Séance I – Analyse des défauts de cache

- Programmation récursive et itérative, mémoïsation
-

Jean-Louis Roch & Équipe pédagogique AOD

Ensimag 2^{ème} année



Présentation du cours

- ▶ **Objectif du cours :** Etant donné un "algorithme", assimiler pourquoi la façon d'écrire le programme impacte le coût (temps, énergie, ...) et comment écrire des programmes portables et efficaces
→ maîtrise de la programmation récursive et itérative.
- ▶ **Cadre applicatif :** résoudre des problèmes d'optimisation discrète
- ▶ **Plan :**
 - ▶ Défauts de cache : analyse (Modèle CO) et design d'algorithmes :
 - ▶ Construction d'algorithmes : techniques *cache-aware* et *cache oblivious*
 - ▶ Modélisation de problèmes d'optimisation discrète sous forme récursive et programmation efficace (cascade)
 - ▶ Méthodes de résolution : *Programmation dynamique* et *Branch and Bound*
- ▶ **Pré-requis :** programmation (C, python, ...), algo., complexité, RO, maths. discrètes ;
- ▶ **Organisation :** 5 séances de CM, 6 de TD, 1h30 de permanence.
- ▶ **Évaluation /MCC :** 1/3 TP (téléchargement en binômes), 2/3 examen sur papier
 - ▶ Contrôle continu rendu 1 : produit matrices (cf chamilio)
 - ▶ Contrôle continu rendu 2 : programmation dynamique avec contrôle localité

Rappel : notation de Landau

- ▶ $f = O(g) \iff \exists C > 0 \ \forall n > n_0 : f(n) < C \times g(n)$
- ▶ $f = \Omega(g) \iff \exists C > 0 \ \forall n > n_0 : f(n) > C \times g(n)$
- ▶ $f = \Theta(g) \iff f = O(g) \text{ et } f = \Omega(g).$

Introduction : hiérarchie mémoire et performance

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Hiérarchie mémoire

Gestion du cache : LRU et approximations

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Mesurer les défauts

Modélisation et analyse théorique des défauts de cache

Le modèle CO : cache de taille Z , ligne de taille L , police LRU

Exemples sur parcours de tableau

Dichotomie et tri QuickSort

Améliorer la localité : algorithmique et programmation

Regrouper données et instructions

Transposition de matrice

Conclusion cours + TP

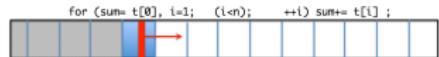
Coût d'exécution d'un programme : temps, énergie, ...

- ▶ *Travail* : nombre d'opérations effectuées ($+, \times, \dots$)
- ▶ *Localité* : Les accès mémoire "non locaux" ont un coût
 - ▶ temps (*elapsed*, pas *cpu*) et énergie
- ▶ Programmation (C, C++, ...) :
 - ▶ parcours récursifs ou itératifs (données et calculs)
 - ▶ Comment programmer efficacement ces parcours ?
- ▶ Objectif du cours : comprendre les "défauts de localité" d'un programme
→ maîtrise/ couplage récursivité et itération
- ▶ Cadre applicatif : *optimisation discrète* : trouver une solution optimale à un problème dans un ensemble fini et grand

Exemple : somme des éléments d'un tableau

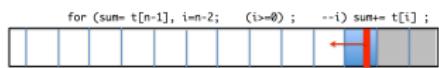
++ parcours INCR : par indice croissant

```
for (sum= t[0], i=1; (i<n); ++i) sum+= t[i];
```



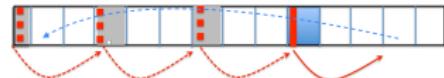
-- parcours DECR : par indice décroissant

```
for (sum= t[n-1], i=n-2; (i>=0); --i) sum+= t[i];
```



+s parcours STRIDE : cyclique par "pas" de s

```
for (sum = t[first=0], i=n-1, first=0, cour=0; i>0; --i)
{cour+=s; if (cour>=n) cour=++first; sum+=t[cour];}
```



Exemple : somme des éléments d'un tableau

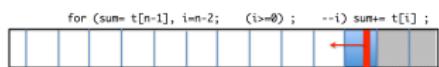
++ parcours INCR : par indice croissant

```
for (sum= t[0], i=1; (i<n); ++i) sum+= t[i];
```



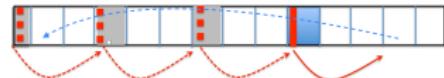
-- parcours DECR : par indice décroissant

```
for (sum= t[n-1], i=n-2; (i>=0); --i) sum+= t[i];
```



+s parcours STRIDE : cyclique par "pas" de s

```
for (sum = t[first=0], i=n-1, first=0, cour=0; i>0; --i)
{cour+=s; if (cour>=n) cour=++first; sum+=t[cour];}
```



- ▶ Temps en millisecondes pour $n = 10^6$ [MacOSX 10.9.4, gcc 4.7.2]

	++	--	+1	+2	+3	+4	+5	+10	+20	+30	+40	+50	+100	+1000	+1001
	2.90	2.82	2.81	2.86	2.94	3.00	3.29	4.38	6.49	5.94	5.67	5.97	4,92	7,61	10,4

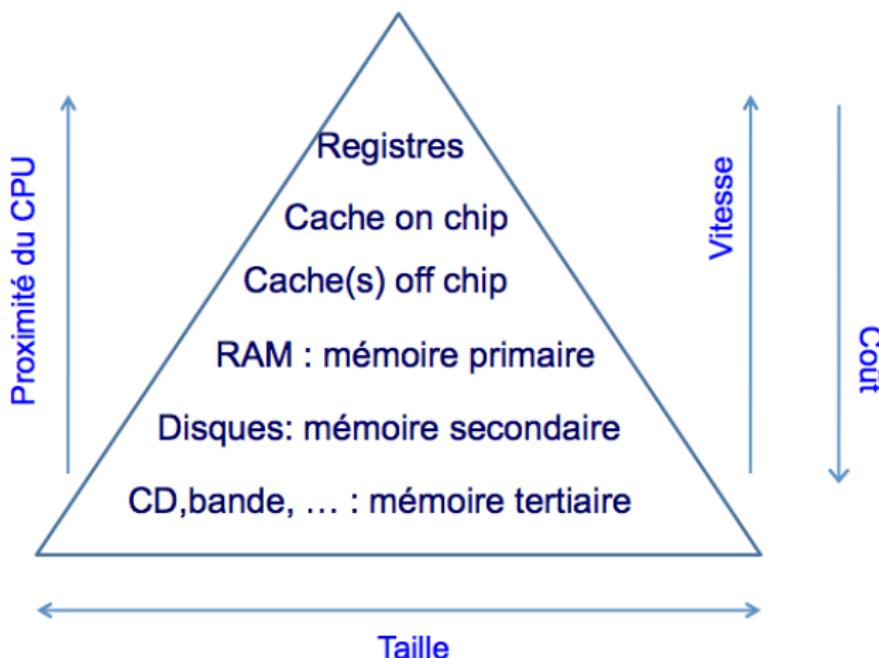
Temps mesurés avec gettimeofday;
Moyenne de 5 exécutions; variance de 10^{-5} à 10^{-4} s²

x 3,7 !!!

Défauts de localité (ou *de cache*)

- ▶ Tout accès en mémoire est fait via une copie de la donnée dans le cache :
 - ▶ Soit la donnée est déjà dans le cache : **cache hit** (coût nul)
 - ▶ Soit elle est absente \implies **défaut de cache** (*cache miss*)
 - \hookrightarrow chargement dans le cache du bloc qui contient la donnée
- ▶ Les défauts de cache sont coûteux mais peuvent être amortis
 - ▶ en accédant les données dans le même bloc (déjà dans le cache)
 - \hookrightarrow **localité spatiale**
 - ▶ en regroupant les accès à une même donnée
 - \hookrightarrow **localité temporelle**
- ▶ Chaque algorithme/programme a un impact sur la localité : une analyse expérimentale ou algorithmique permet d'analyser cet impact

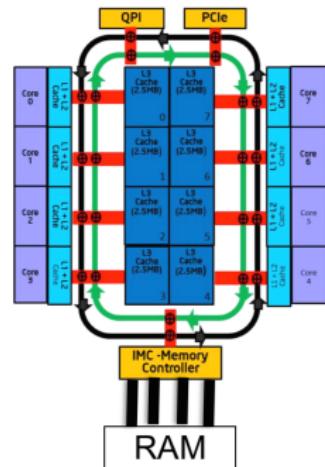
Hiérarchie mémoire



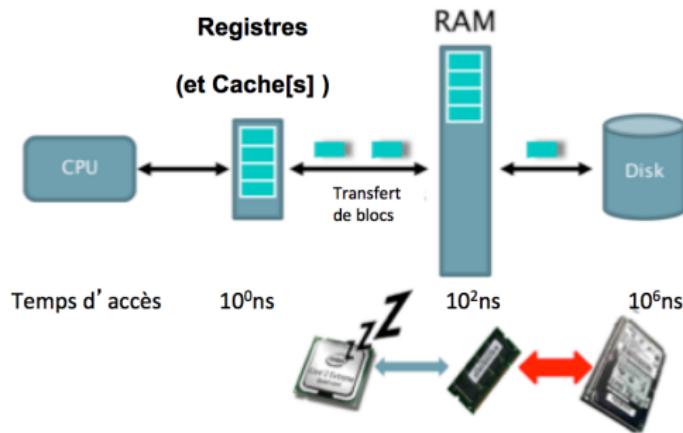
Exemple : Intel SandyBridge-EP

Intel SandyBridge-EP

- ▶ Disque : bloc="page"
- ▶ RAM : 4 contrôleurs mémoire (DDR3)
- ▶ Cache L3 (20 Mo partagé par les 8 coeurs)
- ▶ Cache L2 (256 ko, 8 voies associatif)
- ▶ Cache L1 (32ko données et 32 ko instructions)
- ▶ Registres



Hiérarchie mémoire et accès par blocs



Pour amortir le coût des accès, l'unité de chargement est le "bloc" :

- ▶ la taille de bloc varie : plus on est proche du processeur, plus la mémoire est petite et rapide, plus la taille du bloc est petite ;
- ▶ le nom du "bloc" varie : "ligne de cache" ("page" pour la RAM)

Illustration : parcours simple (INCR ++)

```
for (sum= t[0], i=1; (i<n); ++i) sum += t[i];
```



Element en cours d'accès

Sens du parcours



Bloc qui sera chargé en cas d'accès à un élément



Bloc chargé en cours d'accès



Blocs déjà chargés

Les différents types de défauts de cache

- ▶ **Dans ce cours :** nous nous intéressons qu'aux défauts de localité spatiale ou temporelle :
 - ▶ défauts de cache **obligatoires** : la donnée n'est pas dans le cache, il faut charger son bloc ;
 - ▶ défauts de cache **capacitifs** : le cache est plein, les données les plus anciennes sont remplacées (LRU ou variantes).
- ▶ Il y a d'autres défauts de cache :
 - ▶ défauts de cache **conflictuels** : deux adresses de niveau supérieur sont en collision (chargées au même endroit dans le cache) ;
 - ▶ défauts de **cohérence** : sur multi-processeurs, l'écriture par un processeur d'une donnée invalide la ligne de cache associée à cette donnée sur chacun des autres processeurs.

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
- ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
- ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.

Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

1. accès $T[4] \implies L_1$ est chargée en cache ;	cache=[?, ?, ?]
2. accès $T[1] \implies L_0$ est chargée en cache ;	cache=[?, ?, L_1]
3. accès $T[17] \implies L_4$ est chargée en cache ;	cache=[?, L_1 , L_0]

Le cache est plein ; du plus ancien au plus récent : cache= [L_1 , L_0 , L_4].

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
- ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
- ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.
Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

1. accès $T[4] \Rightarrow L_1$ est chargée en cache ;	cache=[?, ?, ?]
2. accès $T[1] \Rightarrow L_0$ est chargée en cache ;	cache=[?, ?, L_1]
3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;	cache=[?, L_1 , L_0]
- Le cache est plein ; du plus ancien au plus récent : cache= [L_1 , L_0 , L_4].
- 4. accès $T[8] \Rightarrow$

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
 - ▶ **LRU = Least Recently Used** : on écrase celui dont le dernier accès est le plus ancien
 - ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + 3]$ sont sur une même ligne, notée L .

- ▶ Initialement, le cache est vide ; on fait les accès suivants : $\text{cache}=[?, ?, ?]$
 1. accès $T[4] \Rightarrow L_1$ est chargée en cache ; $\text{cache}=[?, ?, L_1]$
 2. accès $T[1] \Rightarrow L_0$ est chargée en cache ; $\text{cache}=[?, L_1, L_0]$
 3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;

Le cache est plein ; du plus ancien au plus récent : $\text{cache}=[L_1, L_0, L_4]$.

 4. accès $T[8] \Rightarrow$ cache **miss** : L_2 est chargée en cache ; $\text{cache}=[L_0, L_4, L_2]$

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
- ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
- ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.

Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

cache=[?, ?, ?]	
-----------------	--
- 1. accès $T[4] \Rightarrow L_1$ est chargée en cache ;

cache=[?, ?, L_1]	
----------------------	--
- 2. accès $T[1] \Rightarrow L_0$ est chargée en cache ;

cache=[?, L_1 , L_0]	
---------------------------	--
- 3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;

Le cache est plein ; du plus ancien au plus récent : cache= [L_1 , L_0 , L_4].
- 4. accès $T[8] \Rightarrow$ cache **miss** : L_2 est chargée en cache ;

cache = [L_0 , L_4 , L_2]	
-----------------------------------	--
- 5. accès $T[2]$

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
- ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
- ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.

Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

1. accès $T[4] \Rightarrow L_1$ est chargée en cache ;	cache=[?, ?, ?]
2. accès $T[1] \Rightarrow L_0$ est chargée en cache ;	cache=[?, ?, L_1]
3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;	cache=[?, L_1 , L_0]

Le cache est plein ; du plus ancien au plus récent : cache= [L_1 , L_0 , L_4].	
4. accès $T[8] \Rightarrow$ cache miss : L_2 est chargée en cache ;	cache = [L_0 , L_4 , L_2]
5. accès $T[2] \Rightarrow$ cache hit	cache = [L_4 , L_2 , L_0]

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
 - ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
 - ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.
- Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

1. accès $T[4] \Rightarrow L_1$ est chargée en cache ;	cache=[?, ?, ?]
2. accès $T[1] \Rightarrow L_0$ est chargée en cache ;	cache=[?, ?, L_1]
3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;	cache=[?, L_1 , L_0]

Le cache est plein ; du plus ancien au plus récent : cache= [L_1 , L_0 , L_4].

4. accès $T[8] \Rightarrow$ cache miss : L_2 est chargée en cache ;	cache = [L_0 , L_4 , L_2]
5. accès $T[2] \Rightarrow$ cache hit	cache = [L_4 , L_2 , L_0]
6. accès $T[13]$	

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
 - ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
 - ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.
- Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

1. accès $T[4] \Rightarrow L_1$ est chargée en cache ;	cache=[?, ?, ?]
2. accès $T[1] \Rightarrow L_0$ est chargée en cache ;	cache=[?, ?, L_1]
3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;	cache=[?, L_1 , L_0]

Le cache est plein ; du plus ancien au plus récent : cache= [L_1 , L_0 , L_4].

4. accès $T[8] \Rightarrow$ cache miss : L_2 est chargée en cache ;	cache = [L_0 , L_4 , L_2]
5. accès $T[2] \Rightarrow$ cache hit	cache = [L_4 , L_2 , L_0]
6. accès $T[13] \Rightarrow$ cache miss	cache = [L_2 , L_0 , L_3]

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
 - ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
 - ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.
- Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

1. accès $T[4] \Rightarrow L_1$ est chargée en cache ;	cache=[?, ?, ?]
2. accès $T[1] \Rightarrow L_0$ est chargée en cache ;	cache=[?, ?, L_1]
3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;	cache=[?, L_1 , L_0]

Le cache est plein ; du plus ancien au plus récent : cache=[L_1 , L_0 , L_4].

4. accès $T[8] \Rightarrow$ cache miss : L_2 est chargée en cache ;	cache=[L_0 , L_4 , L_2]
5. accès $T[2] \Rightarrow$ cache hit	cache=[L_4 , L_2 , L_0]
6. accès $T[13] \Rightarrow$ cache miss	cache=[L_2 , L_0 , L_3]
7. accès $T[2]$	

Remplacement d'une ligne dans le cache : LRU

- ▶ Si il n'y a plus de place dans le cache, le nouveau bloc écrase un existant
 - ▶ **LRU** = *Least Recently Used* : on écrase celui dont le dernier accès est le plus ancien
 - ▶ *Exemple* : cache avec 3 lignes de cache de 4 mots chacune géré par LRU.
- Soit T : un tableau aligné de mots :

$\forall i : T[4 \times i], T[4 \times i + 1], T[4 \times i + 2], T[4 \times i + ..3]$ sont sur une même ligne, notée L_i .

- ▶ Initialement, le cache est vide ; on fait les accès suivants :

1. accès $T[4] \Rightarrow L_1$ est chargée en cache ;	cache=[?, ?, ?]
2. accès $T[1] \Rightarrow L_0$ est chargée en cache ;	cache=[?, ?, L_1]
3. accès $T[17] \Rightarrow L_4$ est chargée en cache ;	cache=[?, L_1 , L_0]

Le cache est plein ; du plus ancien au plus récent : cache=[L_1 , L_0 , L_4].

4. accès $T[8] \Rightarrow$ cache miss : L_2 est chargée en cache ;	cache=[L_0 , L_4 , L_2]
5. accès $T[2] \Rightarrow$ cache hit	cache=[L_4 , L_2 , L_0]
6. accès $T[13] \Rightarrow$ cache miss	cache=[L_2 , L_0 , L_3]
7. accès $T[2] \Rightarrow$ cache hit	cache=[L_2 , L_3 , L_0]
8. ...	

Approximation de LRU : cache associatif

LRU est à un facteur au plus 2 de l'optimal, mais est coûteux à implanter

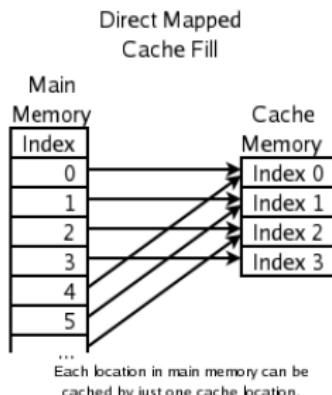
cache de taille 4 : $4! = 24$ états possibles

cache de taille 8 : $8! = 40320$ états possibles

→ approximation matérielle de LRU : **cache associatif**

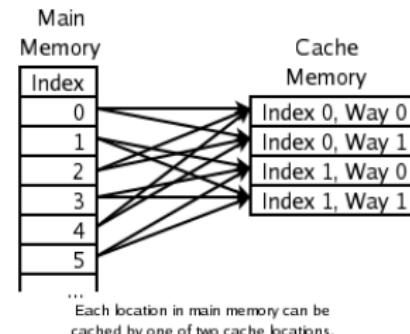
Exemple : cache de taille 4 (i.e. LRU pur = 4 voies)

Simplifications : *Direct mapped* = 1 voie



2 voies (2! = 2 états par index)

2-Way Associative
Cache Fill



[https://en.wikipedia.org/wiki/File:Cache_associative-fill-both.png]

Défauts de localité (ou *de cache*)

- ▶ Tout accès en mémoire est fait via une copie de la donnée dans le cache :
 - ▶ Soit la donnée est déjà dans le cache : **cache hit** (coût nul)
 - ▶ Soit elle est absente \implies **défaut de cache** (*cache miss*)
 - ↪ chargement dans le cache du bloc qui contient la donnée
- ▶ Les défauts de cache sont coûteux mais peuvent être amortis
 - ▶ en accédant les données dans le même bloc (déjà dans le cache)
 - ↪ **localité spatiale**
 - ▶ en regroupant les accès à une même donnée
 - ↪ **localité temporelle**
- ▶ Chaque algorithme/programme a un impact sur la localité : une analyse expérimentale ou algorithmique permet d'analyser cet impact

Mesurer les défauts de cache

- Par compteurs matériels (dépend du processeur, peut requérir accès sudo)

Exemple : PAPI <http://icl.cs.utk.edu/papi/>

Performance Application Programming Interfaces : exemples de compteurs

- ▶ PAPI_L1_TCM Level 1 cache misses
- ▶ PAPI_L1_TCH Level 1 total cache hits
- ▶ PAPI_L2_TCM Level 2 cache misses
- ▶ PAPI_L3_TCM Level 3 cache misses
- ▶ PAPI_L3_LDM Level 3 load misses
- ▶ PAPI_L3_STM Level 3 store misses

- Par simulation (à partir d'un modèle)

Exemple : VALGRIND <http://valgrind.org>

Modèle de cache : peut différer de la machine

- ▶ valgrind --tool=cachegrind <programme>
génère un fichier de statistiques "cachegrind.out.<pid>"

configuration du cache
(en entête de
cachegrind.out.7323)

desc: I1 cache:	32768 B, 64 B, 8-way associative
desc: D1 cache:	32768 B, 64 B, 8-way associative
desc: L2 cache:	2097152 B, 64 B, 8-way associative

```

==7323==
==7323== I refs: 28,972,480 #instructions exécutées
==7323== I1 misses: 1,185 #défauts dans cache I1
==7323== L2i misses: 1,176 #défauts dans cache L2
==7323== I1 miss rate: 0.00% taux = #I1 misses / #I refs
==7323== L2i miss rate: 0.00% taux = #L2i misses / #I refs
==7323==

==7323== D refs: 19,579,638 (17,059,542 rd + 2,520,096 wr)
==7323== D1 misses: 377,596 ( 127,171 rd + 250,425 wr)
==7323== L2d misses: 377,267 ( 126,865 rd + 250,402 wr)
==7323== D1 miss rate: 1.9% ( 0.7% + 9.9% )
==7323== L2d miss rate: 1.9% ( 0.7% + 9.9% )
==7323==

==7323== L2 refs: 378,781 ( 128,356 rd + 250,425 wr)
==7323== L2 misses: 378,443 ( 128,041 rd + 250,402 wr)
==7323== L2 miss rate: 0.7% ( 0.2% + 9.9% )

```

Instructions
Cache I1 = 32 ko
Cache L2= 2Mo



Données
Cache D1 = 32 ko
Cache L2 = 2 Mo
rd = #accès read
wd = #accès write



Total instr+data
sur dernier niveau
LL = L2 ici, 2 Mo



Exemple sur tableau avec accès par stride

- ▶ valgrind --tool=cachegrind ./parcoursTableau 1000000 STRIDE 1
- ▶ quasi même nombre d'instructions pour tous les strides : $29 \cdot 10^6$

Stride	#L2 miss	Taux de défauts de cache
1	378,448	0.7%
2	503,445	1.0%
3	628,449	1.2%
4	753,450	1.5%
8	1,253,447	2.5%
99	1,002,564	2.0%
100	384,168	0.7%
101	382,563	0.7%

- ▶ Attention : c'est une simulation, les résultats mesurés sur machine sont plus complexes.
- ▶ La localité a un impact sur les performances et doit être prise en compte dans la programmation.

Introduction : hiérarchie mémoire et performance

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Hiérarchie mémoire

Gestion du cache : LRU et approximations

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Mesurer les défauts

Modélisation et analyse théorique des défauts de cache

Le modèle CO : cache de taille Z , ligne de taille L , police LRU

Exemples sur parcours de tableau

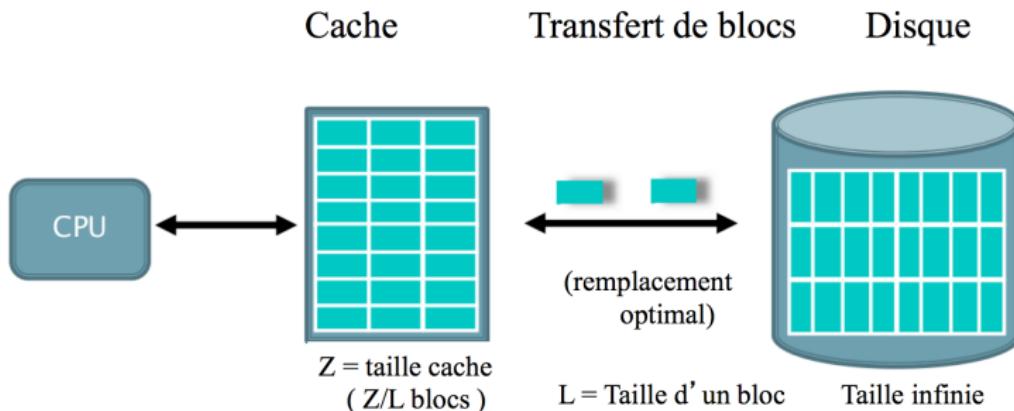
Dichotomie et tri QuickSort

Améliorer la localité : algorithmique et programmation

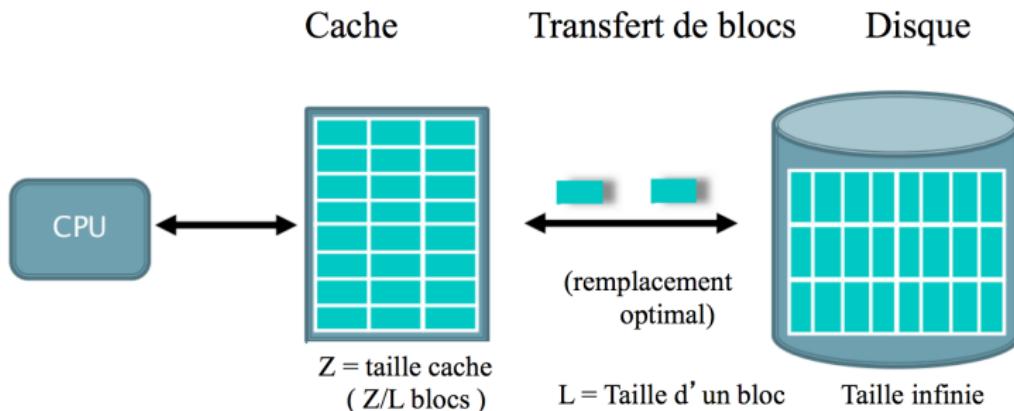
Regrouper données et instructions

Transposition de matrice

Conclusion cours + TP



- ▶ Simplification : hiérarchie à 1 niveau seulement et politique LRU
- ▶ Paramètres : $Z = \text{taille cache}$ $L = \text{taille ligne de cache}$
 Hypothèse : $Z \gg L$ en général : $Z = \Omega(L^2)$
- ▶ $Q(n, L, Z) = \text{nombre de transferts de blocs}$ ($n = \text{taille instance}$)



- ▶ Simplification : hiérarchie à 1 niveau seulement et politique LRU
 - ▶ Paramètres : $Z = \text{taille cache}$ $L = \text{taille ligne de cache}$
Hypothèse : $Z \gg L$ en général : $Z = \Omega(L^2)$
 - ▶ $Q(n, L, Z) = \text{nombre de transferts de blocs}$ ($n = \text{taille instance}$)
 - ▶ Objectif : programme qui minimise $Q(n, L, Z)$ [pour tout L et Z]
 - ▶ **Cache aware** : le programme utilise les valeurs de L et Z
 - ▶ **Cache oblivious** : il est indépendant de L et Z (donc de la hiérarchie!!!!)

- ▶ Modèle CO : cache de taille Z octets, chargé par lignes de L octets
 - ▶ politique de remplacement des lignes dans le cache : LRU
- ▶ Parcours $T[0..n - 1]$ de n octets stockés consécutivement en mémoire :
 - ▶ initialement, aucun élément de T n'est en cache

```
for (sum= t[0], i=1;    (i<n);      ++i) sum+= t[i] ;
```



- ▶ Modèle CO : cache de taille Z octets, chargé par lignes de L octets
 - ▶ politique de remplacement des lignes dans le cache : LRU
- ▶ Parcours $T[0..n - 1]$ de n octets stockés consécutivement en mémoire :
 - ▶ initialement, aucun élément de T n'est en cache

```
for (sum= t[0], i=1; (i<n); ++i) sum+= t[i] ;
```



- ▶ Meilleur cas : $T[0]$ est au début d'une ligne de cache (tableau aligné)
 - ▶ Exemple : $L = 3$ et $n = 5$: T s'étend sur 2 lignes.
 - ▶ $\Rightarrow Q(n, L, Z) = \lceil \frac{n}{L} \rceil$

- ▶ Modèle CO : cache de taille Z octets, chargé par lignes de L octets
 - ▶ politique de remplacement des lignes dans le cache : LRU
- ▶ Parcours $T[0..n - 1]$ de n octets stockés consécutivement en mémoire :
 - ▶ initialement, aucun élément de T n'est en cache

```
for (sum= t[0], i=1; (i<n); ++i) sum+= t[i] ;
```


- ▶ Meilleur cas : $T[0]$ est au début d'une ligne de cache (tableau aligné)
 - ▶ Exemple : $L = 3$ et $n = 5$: T s'étend sur 2 lignes.
 - ▶ $\Rightarrow Q(n, L, Z) = \lceil \frac{n}{L} \rceil$
- ▶ Pire Cas : $T[0]$ est en fin d'une ligne de cache (tableau non aligné)
 - ▶ Exemple : $L = 3$ et $n = 5$: T s'étend sur 3 lignes.
 - ▶ il peut y avoir un bloc à charger en plus $\Rightarrow Q(n, L, Z) = \lceil \frac{n}{L} \rceil + (0 \text{ ou } 1)$

- ▶ Modèle CO : cache de taille Z octets, chargé par lignes de L octets
 - ▶ politique de remplacement des lignes dans le cache : LRU
- ▶ Parcours $T[0..n - 1]$ de n octets stockés consécutivement en mémoire :
 - ▶ initialement, aucun élément de T n'est en cache

```
for (sum= t[0], i=1; (i<n); ++i) sum+= t[i] ;
```


- ▶ Meilleur cas : $T[0]$ est au début d'une ligne de cache (tableau aligné)
 - ▶ Exemple : $L = 3$ et $n = 5$: T s'étend sur 2 lignes.
 - ▶ $\Rightarrow Q(n, L, Z) = \lceil \frac{n}{L} \rceil$
- ▶ Pire Cas : $T[0]$ est en fin d'une ligne de cache (tableau non aligné)
 - ▶ Exemple : $L = 3$ et $n = 5$: T s'étend sur 3 lignes.
 - ▶ il peut y avoir un bloc à charger en plus $\Rightarrow Q(n, L, Z) = \lceil \frac{n}{L} \rceil + (0 \text{ ou } 1)$
- ▶ Dans tous les cas : $\lceil \frac{n}{L} \rceil \leq Q(n, L, Z) \leq \lceil \frac{n}{L} \rceil + 1$

Localité spatiale et temporelle

- ▶ Localité spatiale : accès à des données différentes, mais stockées consécutivement (même bloc).
 - ▶ Si on accède n nouvelles données : $Q(n, L, Z) \geq \frac{n}{L}$
 - ▶ Exemple : parcours contigu d'un tableau

Localité spatiale et temporelle

- ▶ Localité spatiale : accès à des données différentes, mais stockées consécutivement (même bloc).
 - ▶ Si on accède n nouvelles données : $Q(n, L, Z) \geq \frac{n}{L}$
 - ▶ Exemple : parcours contigu d'un tableau
- ▶ Localité temporelle : accès consécutifs (ou presque) à la même donnée
 - ▶ Exemple sans localité temporelle :

```
for (i=0, sum=0; i < n; ++i) { sum += T[i]; }
for (i=0, sumSquare=0; i < n; ++i)) {
    sumSquare += T[i]*T[i];}
```

Localité spatiale et temporelle

- ▶ Localité spatiale : accès à des données différentes, mais stockées consécutivement (même bloc).
 - ▶ Si on accède n nouvelles données : $Q(n, L, Z) \geq \frac{n}{L}$
 - ▶ Exemple : parcours contigu d'un tableau
- ▶ Localité temporelle : accès consécutifs (ou presque) à la même donnée
 - ▶ Exemple sans localité temporelle :

```
for (i=0, sum=0; i < n; ++i) { sum += T[i]; }
for (i=0, sumSquare=0; i < n; ++i)) {
    sumSquare += T[i]*T[i];}
```

- ▶ si $n < Z$: $Q(n, L, Z) = \frac{n}{L}$ si $n > Z$: $Q(n, L, Z) = 2\frac{n}{L}$

- ▶ Localité spatiale : accès à des données différentes, mais stockées consécutivement (même bloc).
 - ▶ Si on accède n nouvelles données : $Q(n, L, Z) \geq \frac{n}{L}$
 - ▶ Exemple : parcours contigu d'un tableau
- ▶ Localité temporelle : accès consécutifs (ou presque) à la même donnée
 - ▶ Exemple sans localité temporelle :

```
for (i=0, sum=0; i < n; ++i) { sum += T[i]; }
for (i=0, sumSquare=0; i < n; ++i)) {
    sumSquare += T[i]*T[i];}
```

▶ si $n < Z$: $Q(n, L, Z) = \frac{n}{L}$ si $n > Z$: $Q(n, L, Z) = 2\frac{n}{L}$

- ▶ Même calcul avec localité temporelle :

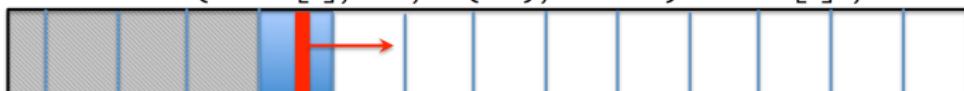
```
for(i=0, sum=0, sumSquare=0; i < n; ++i) {
    sum += T[i]; sumSquare += T[i]*T[i];}
```

$$\implies Q(n, L, Z) = \frac{n}{L}$$

Calculer $Q(n, L, Z)$ pour les 3 parcours du tableau

- ▶ **++INCR** somme par parcours croissant

```
for (sum= t[0], i=1;    (i<n);      ++i) sum+= t[i] ;
```

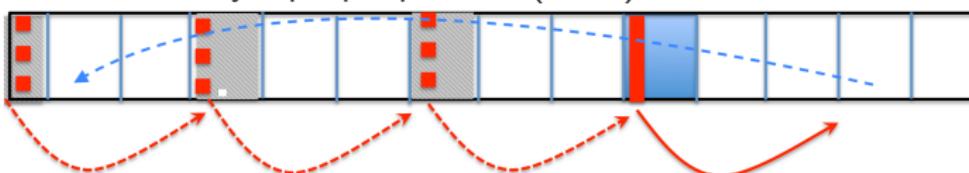


- ▶ **-- DECR** somme par parcours décroissant

```
for (sum= t[n-1], i=n-2;    (i>=0) ;      --i) sum+= t[i] ;
```



- ▶ **+s STRIDE** somme cyclique par pas de s (stride)



Quizz : dichotomie (Binary search)

- ▶ Recherche dichotomique dans un tableau trié de n éléments

```
T* binarySearch( T* begin, T* end, T* val)
{   T* mid = begin + (end - begin)/2 ;
    while (begin < end )
    {   switch (cmp(mid, val))
        {   case LESSER   : begin = mid+1 ; break;
            case GREATER : end = mid-1 ; break;
            case EQUAL    : return mid ;
        }   }
    return NULL ;
}
```

- ▶ Modèle CO : cache de taille Z chargé par blocs de taille L , politique LRU
- ▶ Combien de défauts de cache $Q(n, L, Z) =$
 - A) $n - L$
 - B) n/L
 - C) $\log n - \log L$
 - D) $\log n / \log L$
- ▶ Sondage : <https://doodle.com/poll/7bq5g2ynyxuepkn4>



Qsort : algorithme de partition

```
1 pivotval = *first ;           // pivotval is the value of the pivot
2 pivot = first+1, right = last-1 ;
3 while ( left < right ) {
4     while( *left <= pivotval) left++; // Move left while item<pivot
5     while(*right > pivotval) right--; // Move right while item>pivot
6     if ( left < right ) SWAP(left, right);
7 }
8 first = *right; *right = pivotval; // right is pivot final position
```

- ▶ Calcul *en place* avec toujours 2 blocs locaux



- ▶ $Q(n, L, Z) = \frac{n}{L}$: optimal pour une seule partition

```

1 void Sort (T, first, last) {
2     if ( last -first > 1 ) {
3         int m = partition( first, last);
4         Sort( T, first, m);
5         Sort( T, m, last);
6     }
7 }
```

- ▶ Les défauts faits lors du 1er appel ne profitent pas au 2ème sauf si tout le tableau tient dans le cache
- ▶ si $n < Z$: $Q(n, L, Z) = \frac{n}{L}$
- ▶ si $n > Z$: dès qu'on a un sous tableau de taille Z , on a Z/L défauts.
 - ▶ Meilleur cas : arbre des partitions équilibré : $Q(n, L, Z) = \frac{n}{L} \log \frac{n}{Z}$
 - ▶ Pire cas : arbre déséquilibré $Q(n, L, Z) = \sum_{k=n}^Z \frac{k}{L} = \frac{n^2 - Z^2}{2L}$
- ▶ Non optimal : on peut programmer en $Q(n, L, Z) = O\left(\frac{n}{L} \log_Z n\right)$
 - ▶ cf exercice à la maison (après TD1) : analyser le mergesort (similaire)

Introduction : hiérarchie mémoire et performance

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Hiérarchie mémoire

Gestion du cache : LRU et approximations

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Mesurer les défauts

Modélisation et analyse théorique des défauts de cache

Le modèle CO : cache de taille Z , ligne de taille L , police LRU

Exemples sur parcours de tableau

Dichotomie et tri QuickSort

Améliorer la localité : algorithmique et programmation

Regrouper données et instructions

Transposition de matrice

Conclusion cours + TP

Comment améliorer la localité ? Grouper!

- ▶ Regrouper les données (*localité spatiale*) ou/et les instructions *localité temporelle*
- ▶ Analogie : si on doit acheter :
 1. 3 baguettes
 2. 1 citron
 3. 1 reblochon
 4. 3 litres d'eau
 5. 6 oranges
 6. 2 plaques de beurre
 7. 1 kg de quetsches
- ▶ Méthode : schémas algorithmiques **par blocs**
 - ▶ Données : tableaux multidimensionnels
 - ▶ Instructions : itératif par bloc ou récursif jusqu'à seuil

Défauts lors du parcours d'une matrice

- ▶ En C/C++, python, java, rust, ... : stockage par lignes (**row major**)
 $A[...][k - 1], A[...][k], A[...][k + 1]$ sont stockés consécutivement.
- ▶ Soit A une matrice $n \times m$ avec n lignes et m colonnes
- ▶ Analyse des défauts de cache
 - ▶ parcours par **lignes** : `for (i=0; i<n; ++i) for (j=0; j<m; ++j) { ... A[i][j] ...}`

$$\forall n, m : \quad \left\lceil \frac{nm}{L} \right\rceil \leq Q(n, m, Z, L) \leq \left\lceil \frac{nm}{L} \right\rceil + 1$$

cache oblivious !

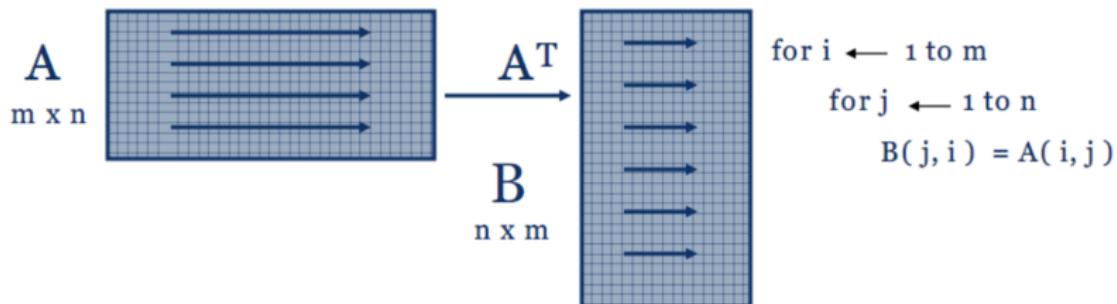
Défauts lors du parcours d'une matrice

- ▶ En C/C++, python, java, rust, ... : stockage par lignes (**row major**)
 $A[...][k - 1], A[...][k], A[...][k + 1]$ sont stockés consécutivement.
 - ▶ Soit A une matrice $n \times m$ avec n lignes et m colonnes
 - ▶ Analyse des défauts de cache
 - ▶ parcours par **lignes** : `for (i=0; i<n; ++i) for (j=0; j<m; ++j) { ... A[i][j] ...}`
- $$\forall n, m : \quad \left\lceil \frac{nm}{L} \right\rceil \leq Q(n, m, Z, L) \leq \left\lceil \frac{nm}{L} \right\rceil + 1$$
- cache oblivious !
- ▶ parcours par **colonnes** : `for (j=0; j<m; ++j) for (i=0; i<n; ++i) { ... A[i][j] ...}`

Défauts lors du parcours d'une matrice

- ▶ En C/C++, python, java, rust, ... : stockage par lignes (**row major**)
 $A[...][k - 1], A[...][k], A[...][k + 1]$ sont stockés consécutivement.
 - ▶ Soit A une matrice $n \times m$ avec n lignes et m colonnes
 - ▶ Analyse des défauts de cache
 - ▶ parcours par **lignes** : `for (i=0; i<n; ++i) for (j=0; j<m; ++j) { ... A[i][j] ...}`
- $$\forall n, m : \quad \left\lceil \frac{nm}{L} \right\rceil \leq Q(n, m, Z, L) \leq \left\lceil \frac{nm}{L} \right\rceil + 1$$
- cache oblivious !
- ▶ parcours par **colonnes** : `for (j=0; j<m; ++j) for (i=0; i<n; ++i) { ... A[i][j] ...}`
 - ▶ si $nL \ll Z$, même nombre de défauts que par lignes
 - ▶ mais si $n > Z$ alors $Q(n, m, Z, L) = nm$: beaucoup moins performant !

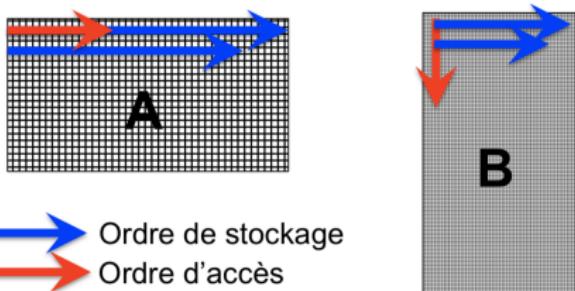
- Le disque = séquence de mots
 - Matrices stockées par lignes (*row major*)



- Matrices stockées par lignes

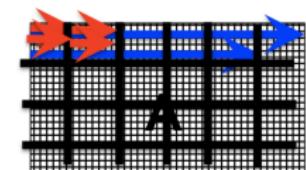
```

For i = 0.. m-1
  For j=1 .. n
    B[ j, i ] = A[ i, j ]
  
```

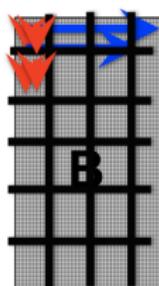


- Sur A : nm/L défauts de cache
- Sur B: si $n > Z$, l'accès de B par colonne cause un défaut de cache à chaque pas => $n.m$ défauts sur B
- Au total: $Q(n,m,L,Z) = n.m (1 + 1/L)$ défauts
- Remarque: idem en inversant les boucles: `for j { for i { ... }}`

- Partitionner A et B en « blocs » de taille $K \times K = K^2$ tel que la transposition d'un bloc dans l'autre tienne en cache ($2 \cdot K^2 \ll Z$)
 - Si bloc $K \times K$: $\leq 2 \cdot K + K^2 / L$ cache miss par bloc de K^2 éléments
 - Or $n \cdot m / K^2$ blocs par matrice => $2 \cdot n \cdot m (1/K + 1/L)$ défauts de cache



→ Ordre de stockage
 → Ordre d'accès



```

For ( I = 0; I < m ; I += K )
  For ( J = 0; J < n ; J += K )
    // transpose bloc
    For ( i=I ; i < I+K; ++i)
      For ( j=J ; j < J+ K ; ++j )
        B[ j, i ] = A[ i, j ]
```

- $Q(m,n) = O(m \cdot n / Z^{1/2} + m \cdot n / L)$: optimal.
- Remarque: le programme doit connaître Z pour déterminer la valeur de K proche de $(Z/2)^{1/2}$ (mais pas besoin de connaître L).

- Partitionner A selon la plus grande dimension, et récursivement transposer chaque bloc jusqu'à un seuil:

A_{11}	A_{21}
A_{12}	A_{22}

A_{11}^T	A_{12}^T
A_{21}^T	A_{22}^T

- Analyse:
 - $Q(m,n) = O(n + m + n.m/L)$ si $n, m < K = \text{Seuil}$ (et $Z > 2 K^2$)
 - Sinon si $m > n$: $Q(m,n) \leq 2Q(m/2,n) + O(1)$
ou si $m < n$: $Q(m,n) \leq 2Q(m,n/2) + O(1)$
- D'où $Q(m,n) = O(m+n + m.n/L)$: optimal.
- «Cache-oblivious»: le programme ne référence pas Z ou L .

Introduction : hiérarchie mémoire et performance

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Hiérarchie mémoire

Gestion du cache : LRU et approximations

Types de défauts de localité et analyse

Défauts de localité (ou défauts de cache)

Mesurer les défauts

Modélisation et analyse théorique des défauts de cache

Le modèle CO : cache de taille Z , ligne de taille L , police LRU

Exemples sur parcours de tableau

Dichotomie et tri QuickSort

Améliorer la localité : algorithmique et programmation

Regrouper données et instructions

Transposition de matrice

Conclusion cours + TP

- ▶ La programmation d'un algorithme a un impact important sur son coût (temps, énergie, ...)
- ▶ Analyse : un seul cache de taille Z accédé par blocs de taille L (+LRU)
- ▶ Regrouper par bloc : de taille Z (cache aware) ou récursif (cache oblivious)
- ▶ Sur une machine réelle, il y a plusieurs niveaux de caches : un algorithme "cache-oblivious" qui fonctionne bien à chaque niveau, devrait être efficace sur toute la hiérarchie.
- ▶ Exercice : TP0 à faire à la maison :
 - ▶ analyse expérimentale du coût d'un programme simple de produit de matrices
 - ▶ à rendre sur teide AVANT le 26 septembre