TD 1 d'Algorithmique et Optimisation Discrète — Ensimag

Analyse des défauts de localité

Équipe pédagogique AOD
12 septembre 2022

Préambule : Synthèse du cours

Une unité de calcul ne pouvant manipuler que les données stockées dans l'unité, tout accès à une donnée distante (en lecture ou en écriture) se fait via une copie locale au sein de l'unité. Le coût d'un accès mémoire varie fortement selon la localisation de la donnée accédée. La mémoire est donc hiérarchique, des volumes mémoire les plus éloignés (les plus grands et les plus coûteux à accéder) à ceux les plus proches (les registres).

Pour amortir le coût des accès mémoire, à chaque niveau de la hiérarchie on utilise un mécanisme de *cache* : lorsqu'on accède une donnée qui n'est pas dans le cache, on va lire sur la mémoire distante le bloc de mémoire (appelé *ligne de cache*) qui contient la donnée et on le charge dans le cache à la place d'un bloc qui n'a pas été accédé depuis longtemps (qui est écrasé). Par contre, si le bloc contenant la donnée est déjà dans le cache, on y accède directement avec un coût moindre. Parmi les caractéristiques importantes d'un cache : 1. sa taille ; 2. la longueur de la ligne de cache ; 3. le choix du bloc écrasé qui est souvent une approximation de LRU (*Least Recently Used*) comme par exemple dans un cache associatif à 8 voies, etc. Il existe des compteurs pour mesurer les défauts de cache en cours d'exécution ; ainsi que des simulateurs (comme `valgrind` avec l'option `-cachegrind` permettant d'émuler un cache théorique.

Lors de la programmation, pour tirer partie du cache, il est important de prendre en compte la localité :

- *localité spatiale* : en accédant des données (resp. en exécutant des instructions) qui sont situées dans la zone mémoire proche des données (resp. instructions) accédées récemment ;
- *localité temporelle* : en ré- accédant des données (resp. en ré- exécutant des instructions) accédées dans le passé pas trop lointain.

Pour concevoir et analyser des algorithmes en prenant en compte la localité, le modèle de cache simplifié CO consiste en un cache de taille Z avec une taille L de ligne de cache ; on suppose toujours $Z > L^2$ (i.e. en théroie $Z = \Omega(L^2)$ suffit).

- Un algorithme *cache-aware* utilise les paramètres du cache, ici L et Z.
- Un algorithme *cache-olivious* ne les utilise pas dans le code, mais effectue un nombre de défauts de cache analysé pour des valeurs arbitraires de L et Z (et si possible proche des meilleurs algorithmes cache-aware).

Pour simplifier l'analyse des algorithmes avec des tableaux, on suppose que l'unité est la place mémoire occupée par un élément de tableau. Ainsi, L éléments consécutifs d'un tableau et alignés tiennent sur une ligne de cache. Le parcours de n cases consécutives d'un tableau non déjà en cache génère donc $Q(n, L, Z) = \lceil \frac{n}{L} \rceil + \delta$ défauts de cache avec $\delta \in \{0, 1\}$ [1] ; donc pour n grand, $Q(n, L, Z) \simeq \frac{n}{L}$.

Préambule - Vocabulaire

- **LRU** (Least Recently Used) : politique de remplacement des lignes de cache
- **CO**, modèle de cache simplifié : cache de taille Z accèdé par lignes de taille L avec défauts gérés par LRU.
- **alignement** : une zone mémoire (ex. tableau) est dite *alignée* ssi son adresse de début est multiple de la taille L de ligne de cache.
  Ainsi, les L premiers octets sont chargés en cache dans une même ligne, les L suivants dans une autre ligne etc.
- **row major** : le stockage *row major* d'une matrice A de taille $n \times m$ consiste à la stocker dans un tableau unidimensionnel `TA` ligne après ligne, donc contigüment en mémoire.
  Pour A de taille $N \times M$, $A_{i,j}$ est stocké dans la case `TA[ i*M + j ]`.
  Exemple : les 6 éléments de la matrice A de taille 2×3 (i.e. $A[2](3]$) sont stockés consécutivement dans le tableau `TA`, dans l'ordre $A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2]$ aux adresses respectives `TA+0, TA+1, TA+2, TA+3, TA+4, TA+5`.
- **tableaux multidimensionnels**. dans la plupart des langages (C,C++, Java, Python, Ada, Rust, ...) les tableaux multidimensionnels sont stockés contigüment en mémoire lignes par lignes (*row-major*) /, : les tableaux multidimensionnels sont stockés en mémoire par lignes, comme en C, Java ou Python (à la différence de Fortran où ils sont stockés par colonne). Exemple en C : double A[2][2][2] stocke les 8 éléments contigüment dans l'ordre : $A[0][0][0], A[0][0][1], A[0][1][0], A[0][1][1], A[1][0][0], A[1][0][1], A[1][1][0], A[1][1][1]$ aux adresses respectives `A+0, A+1, A+2, A+3, A+4, A+5, A+6, A+7`.
  Remarque : Parfois on peut les représenter aussi comme un tableau de pointeurs vers des tableaux.

1. $\delta = 1$ nécessite que premiere et dernière cases accédées ne soient pas alignées respectivement sur le début et la fin d'un bloc de cache).

1 Cache, alignement et politique LRU (10')

Un cache sur le modèle CO avec $Z = 8$ mots et $L = 2$ mots contient 4 lignes de cache de 2 mots chacune.
Soit T un tableau aligné de mots : pour tout i, les 2 mots $T[2 \times i], T[2 \times i + 1]$ ocupent une zone mémoire L_i qui est chargée par ligne de cache.
On accède $T[4]$: la ligne mémoire L_2 est chargée en cache ; puis $T[1]$, la ligne L_0 est chargée ; puis $T[7]$ et L_3 est chargée ; puis $T[8]$ et L_4 est chargée.
Le cache est plein et contient donc les 4 lignes L_2, L_0, L_3, L_4 du tableau T.

Question 1 On effectue alors dans l'ordre les 4 accès suivants : en suivant la politique LRU de remplacement des lignes dans le cache, indiquer pour chaque accès si il engendre ou non un défaut et quelles lignes de T sont en cache après l'accès :
1. accès $T[6]$; *2.* accès $T[2]$; *3.* accès $T[4]$; *4.* accès $T[1]$. *5.* accès $T[2]$.

Initialement, le cache contient dans l'ordre du plus ancien au plus récent : L_2, L_0, L_3, L_4

1. accès $T[6]$: pas de défaut/hit et le cache contient dans l'ordre L_2, L_0, L_4, L_3
2. accès $T[2]$: défaut/miss et le cache contient dans l'ordre L_0, L_4, L_3, L_1
3. accès $T[4]$: défaut/miss et le cache contient dans l'ordre L_4, L_3, L_1, L_2
4. accès $T[1]$: défaut/miss et le cache contient dans l'ordre L_3, L_1, L_2, L_0
5. accès $T[2]$: pas de défaut/hit et le cache contient dans l'ordre L_3, L_2, L_0, L_1

Question 2 Quels éléments de T sont dans le cache ?

Les éléments d'indices 0..7 *soit, dans l'odre du dernier accédé au plus ancien :* $(2,3); (0,1); (4,5); (6,7)$.

2 Parcours de tableau aligné (10')

On considère un cache de taille $Z = 1024$ octets avec une taille de ligne de cache $L = 64$ octets géré par politique LRU.
Soit t un tableau de 1000 `double`, aligné en mémoire. La boucle suivante, qui parcourt t par pas constant de K, lit et modifie un élément sur K :

```
1     for (size_t i=0; i < 1000 ; i += K ) t[i] += 1   ;
```

On suppose qu'avant cette boucle, le cache ne contient aucune donnée de t.

Question 3 Un `double` tenant 8 octets, combien de défauts de cache fait la boucle pour : a) $K = 1$; b) $K = 4$; c) $K = 10$.

Une ligne contient une zone alignée de 64 octets donc 8 double consécutifs (ou si non alignée, 7 double et 8 octets). Donc $t[8 * i..8 * i + 7]$ sont chargés simultanément dans une même ligne de cache.

a) #défauts = 1000 / 8 = 125 (le tableau commence sur une ligne de cache avec t[0]

b) #défauts = 125 (comme t[8*i] et t[8*i+4] sont modifiés, tous les éléments de t sont en fait chargés en mémoire)

c) #défauts = 100 : car 1000/10 = 100 double sont accédés, chacun figurant dans une ligne de cache distincte

Remarque : si la politique est une approximation de LRU (exemple cache associatif 2 voies ou 4 voies etc), le nombre de défauts reste le même.

Question 4 On considère une hiérarchie mémoire à q niveaux suivant le modèle CO $(Z_1, L_1), \ldots, (Z_q, L_q)$ avec $Z_1 < Z_2 < \cdots < Z_q$ et 8 octets $\leq L_1 \leq L_2 \leq \cdots \leq L_q$. Pour $K = 1$, la boucle ci-dessus est-elle optimale en nombre de défauts de cache à tout niveau de la hiérarchie (on dit *cache oblivious*) ?

Le tableaue de 1000 double est de taille $n = 8000$ octets ; Le nombre défauts est n/L_i au niveau i ce qui est minimal. Le programme est cache oblivious.

3 Parcours par ligne ou par colonne de tableau bidimensionnel (20')

On considère un cache de taille Z mots, accèdé par lignes de cache de taille L mots (modèle CO avec LRU).

Soit $A[n][n]$ une matrice carrée de taille $n \times n$ mots stockée *row major* (i.e. ligne après ligne) dans un tableau contigu en mémoire (comme en C, Java ou Python) : pour $0 \leq i < n$ et $0 \leq j < n$, $A[i][j]$ est stocké à l'adresse $A + i \times n + j$.

Ces deux programmes calculent la somme des éléments de la matrice : (a) par ligne, (b) par colonne.

(a) Calcul de $r = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_{i,j}$

```
1 double r = 0 ;
2 for (int i = n-1; i >= 0; --i) {
3    for (int j = n-1; j >= 0; --j)
4       r += A[i][j];
5 }
```

(b) Calcul de $s = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} A_{i,j}$

```
1 double s = 0 ;
2 for (int j = n-1; j >= 0; --j) {
3    for (int i = n-1; i >= 0; --i)
4       s += A[i][j] ;
5 }
```

Question 5 Ici le cache est très grand, suffisant pour contenir la matrice A ($Z \gg n^2$). Initialement, le cache ne contient aucun des coefficients de A. Analyser le nombre de défauts de cache de chacun des 2 programmes.

Les 2 programmes effectuent $\frac{n^2}{L} + O(1)$ défauts de cache.

Question 6 Ici le cache est très petit. (i.e. $Z \ll n$). Analyser le nombre de défauts de cache de chacun des 2 programmes.

Le programme de gauche (r) parcourt A par ligne, dans le sens de son stockage, donc avec $n^2/L + O(1)$ défauts.

le programme de droite (s) parcourt A par colonne avec $n^2 + O(n)$ défauts :

- la boucle interne `for i` parcourt la colonne j et fait 1 défaut à chaque élément de A, donc $n + O(1)$ défauts ;
- la boucle externe `for j` fait donc $n \times (n + O(1)) = n^2 + O(n)$ défauts en tout ;

Question 7 En déduire un programme *cache-oblivious* optimal en cache qui calcule la moyenne $M = \frac{1}{n^2} \sum_{i,j=0}^{n-1} A_{i,j}$ et la variance $V = \frac{1}{n^2} \sum_{i,j=0}^{n-1} (A_{i,j} - M)^2 = \left(\frac{1}{n^2} \sum_{i,j=0}^{n-1} A_{i,j}^2\right) - M^2$ des éléments de A.

Le programme ci-dessous fait $n^2/L + O(1)$ défauts, ce qui est minimal puisque les n^2 éléments de A doivent être accédés.

Calcul de $r = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_{i,j}$ *et* $r2 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_{i,j}^2$

```
1 double r = 0 ;
2 double r2 = 0 ;
3 for (int i = n-1; i >= 0; --i)
4 {  for (int j = n-1; j >= 0; --j)
5    {  r += A[i][j];
6       r2 += A[i][j] * A[i][j];
7    }
8 }
9 M = r/(n*n) ;
10 V = r2/(n*n) - M*M ;
```

4 Parcours de tableau multidimensionnel selon le stockage (20')

On considère un cache de taille Z mots, accèdé par lignes de cache de taille L mots (modèle CO avec LRU).

On stocke une matrice $n \times m$ dans un tableau `A [ n * m ] ]` : $A_{i,j}$ est stocké dans la case `A[ i * m + j]`.

Pour limiter les défauts de cache, on prrivilégie les parcours de A dans le sens de stockage, par exemple de `beginA = A` à `endA = A + n*m` avec la boucle `for ( itA= beginA; (itA != endA); ++itA ).`

Question 8 Le programme C ci-dessous à droite remplit le tableau S (préalloué de taille n) avec le max de chaque ligne de A : pour $0 \leq i < n$, $S[i] = \max_{j=0}^{m-1} \{A[i][j]\}$.

Combien ce programme engendre-t-il de défauts de cache sur A ? sur S ? Est-il bon en cache ? cache aware ? cache oblivious ?

```
// Principe du calcul par ligne (row major)
for (int  i=0; i<n ; ++i)
{ S[i] = A[i][0] ;
  for (int j=1; j<m ; ++j)
  {
    if (S[i] < A[i][j]) S[i] = A[i][j];
  }
}
```

```
// Programmation en C par itérateur
Elt *itA=A ;
for (int  i=0; i<n ; ++i)
{ S[i] = *(itA++) ;
  for (int j=1; j<m ; ++j)
  { Elt v = *(itA++);
    if (S[i] < v) S[i] = v ;
  }
}
```

Le programme effectue nm comparaisons et $\frac{nm}{L} + \frac{n}{L} + O(1)$ défauts de cache.
- A est parcourue de manière contigue donc nm/L défauts sur A.
- à i fixé, La ligne de cache qui contient $S[i]$ reste en cache (à cause de LRU) ; donc n/L défauts sur S (en lecture/écriture).

Donc ce programme est optimal en cache et *cache oblivious* (car indépendant de la taille du cache).

Question 9 Écrire un programme qui remplit le tableau T (préallloué de taille m) avec le min de chaque colonne, i.e. pour $0 \leq j < m$, $T[j] = \min_{i=0}^{n-1} \{A[i][j]\}$; votre programme doit effectuer $\left(\frac{nm}{L}\right)$ défauts de cache sur la matrice A. Préciser le nombre de comparaisons effectuées et le nombre total de défauts de cache.

```
      for (int j=0; j<m ; ++j) T[j] = A[0][j] ;
      for (int  i=1; i<n ; ++i)
        for (int j=0; j<m ; ++j)
          if (T[j] >  A[i][j] )  T[j] = A[i][j] ;
```

Le programme effectue $m(n-1)$ comparaisons et $\frac{2nm}{L}$ défauts de cache ($\frac{nm}{L}$ sur A et $\frac{nm}{L}$ sur T).
Explication :
- A est lu dans le sens de stockage : donc $\frac{nm}{L}$ défauts de cache (en lecture) sur A.
- T : l'initialisation de T fait $\frac{m}{L}$ défauts de cache (en écriture) sur T. En pire cas, si $Z < m$ (ou même $Z < m.L$), la boucle interne sur j (à i fixé) parcourt T dans le sens de stockage donc fait $\frac{m}{L}$ défauts de cache sur T en lecture (et au pire cas en écriture si les éléments de A sont décroissants dans chaque colonne) ; donc, pour tous les i, en tout $\frac{nm}{L}$ défauts sur T en lecture et en écriture.

Soit au total dans tous les cas $2\frac{nm}{L} + \frac{m}{L} \simeq 2\frac{nnm}{L}$ défauts de cache en lecture ; et au pire cas $\frac{nm}{L}$ défauts de cache en écriture.

Question 10 On considère une hiérarchie mémoire à q niveaux $(Z_1, L_1), (Z_2, L_2), \ldots, (Z_q, L_q)$ avec $Z_1 < Z_2 < \cdots < Z_q$. Pogrammez le calcul de S et T simultanément pour effectuer $O\left(\frac{nm}{L_i}\right)$ défauts à chaque niveau (cache oblivious) ; analyser le nombre de défauts (donner un équivalent).

On construit un algorithme cache oblivious. Deux solutions :
- **Solution 1 :** simpliste : les deux algorithmes précédents étant cache oblivious, on les enchaîne séquentiellements en faisant $3nm/L$ défauts, donct $2nm/L$ sur A.
- **Solution 2 :** Pour ne faire qu'un seul parcours sur A et donc seulement nm/L défauts sur A, on entrelace les deux algorithmes précédents :

```
  { // Initialisation  S et T
    for (int i=0; i<n ; ++i) S[i] = -MAXFLOAT ;
    for (int j=0; j<m ; ++j) T[j] =  MAXFLOAT ;
  }
  { // Calcul S et T
    for (int i=0; i<n ; ++i)
      for (int j=0; j<m ; ++j) {
        if ( S[i] < A[i][j] ) S[i] = A[i][j] ;
        if ( T[j] > A[i][j] ) T[j] = A[i][j] ;
      }
  }
```

ou mieux, en évitant l'initialisation avec MAXFLOAT pour faire seulement $n(m-1)+(n-1)m = 2(n-1)(m-1)+n+m$ comparaisons (au lieu de nm dans la version précédente) :

```
1   S[0] = T[0] = A[0][0] ;
2   for (int j=1; j<m ; ++j) { // cas i=0
3     T[j]= A[0][j] ;
4     if (S[0] < A[0][j]) S[0] = A[0][j];
5   }
6   for (int  i=1; i<n ; ++i)  { // cas i>=1
7     S[i] = A[i][0] ;
8     if  (T[0] > A[i][0] )  T[0] = A[i][0] ;
9     for (int j=1; j<m ; ++j) { // cas j>=1
10      if (S[i] < A[i][j]) S[i] = A[i][j];
11      if (T[j] > A[i][j] )  T[j] = A[i][j] ;
12    }
13  }
```

Cet algorithme itératif est cache oblivious et effectue :
- $(n-1)(m-1)$ comparaisons ;
- $\frac{nm}{L_i}$ défauts en lecture sur A au niveau i, $\forall 1 \leq i \leq q$;
- $\frac{nm}{L_i}$ défauts en lecture-écriture sur T au niveau i, $\forall 1 \leq i \leq q$;
- $\frac{n}{L_i}$ défauts en lecture-écriture sur S au niveau i, $\forall 1 \leq i \leq q$;

soit en tout $O\left(\frac{nm}{L_i}\right)$ défauts au niveau i de la hiérarchie.
Ce programme cache oblivious itératif est très simple et au plus à un facteur 2 du nombre de défauts optimal.

⋆ **Remarque :** la suite du cours présente une méthode *cache oblibious* qui permet de calculer théoriquement les deux tableaux S et T en faisant, à chaque niveau i de la hiérarchie, un nombre total de défauts de cache inférieur à $(1+\varepsilon)\frac{nm}{L_i} + O\left(\frac{n+m}{L_i}\right)$ où ε est petit quand $Z_i \gg L_i^2$ est grand, même lorsque $n \gg Z_i$ et $m \gg Z_i$.

La méthode est d'utiliser en paramètre le taille Z du cache pour construire un programme ad-hoc (cache-aware) en regroupant les données par blocs qui tiennent en cache, puis de rendre ce paramètre Z implicite grâce à une découpe récursive des données en blocs de plus en plus petits.

- **Cache aware** : le programme précédent génère nm/L défauts sur A qui est optimal, mais aussi nm/L sur T (ce qui est beaucup) et n/L sur S (ce qui est inutilement optimal). Donc on va essayer de diminuer le nombre de défauts sur T quitte à augmenter le nombre de défauts sur S avec une technique de blocking : on calcule S et T en parcourant A par blocs de taille $K_l \times K_c$: on choisit alors K_l et K_c pour minimiser les défauts de cache, avec la contrainte que les 3 blocs $(A[I][J], S[I], T[J])$ tiennent en cache simultanément, ie $K_l \times K_c + K_l + K_c < Z$:

```
1   for (int i=0; i<n ; ++i) S[i]= -MAXFLOAT ;
2   for (int j=0; j<m ; ++j) T[j]= MAXFLOAT ;
3   for (int I=0; I<n ; I+=K_l )
4     for (int J=0; J<m ; J+=K_c ) {
5       int maxi = (n < I+K_l) ? n : I+K_l ;
6       int maxj = (m < J+K_c) ? m : J+K_c ;
7       for (int i=I; i<maxi ; ++j)
8         for (int j=0; j<maxj ; ++j) {
9           if (S[i] < A[i][j]) S[i] = A[i][j] ;
10          if (T[j] >  A[i][j] )  T[j] = A[i][j] ;
11        }
12    }
```

Le nombre de défauts $Q(n,m,L,Z)$ est inférieur à $\left\lceil \frac{n}{K_l} \right\rceil \times \left\lceil \frac{m}{K_c} \right\rceil \times \left(K_l \times \left\lceil \frac{K_c}{L} + 1 \right\rceil + \left\lceil \frac{K_l}{L} + 1 \right\rceil + \left\lceil \frac{K_c}{L} + 1 \right\rceil + O(1)\right)$
Comme a $K_l.K_c + K_l + K_c \simeq Z^-$, les 2 boucles internes (i,j) tiennent en cache d'où les défauts :
$Q(n,m,L,Z) \leq \frac{nm}{L}\left(1 + \frac{L}{\sqrt{K_l}} + \frac{1}{\sqrt{K_l}} + \frac{1}{\sqrt{K_c}} + O\left(\frac{L}{K_l.K_c}\right)\right) \simeq \frac{nm}{L}\left(1 + \frac{L+2}{\sqrt{Z}} + O\left(\frac{L}{Z}\right)\right) = (1+\epsilon)\frac{nm}{L} = (1+\epsilon)\frac{nm}{L}$. Donc
$Q(n,m,L,Z) \simeq \frac{nm}{L}\left(1 + \frac{s+L}{\sqrt{Z}} + O\left(\frac{L}{Z}\right)\right) = (1+\epsilon)\frac{nm}{L}$ avec ϵ qui tend vers 0 quand $Z \to \infty$.

- **Cache oblivious** : du programme cache-aware, on dérive un programme cache oblivious en écrivant un programme récursif qui met à jour S et T en découpant récursivement en deux la sous-matrice A en entrée sur sa plus grande dimension, jusqu'à atteindre une dimension minimale $s \times s$ en dessous de laquelle on parcourt la sous-matrice en faisant la mise à jour des blocs correspondants de S et T.
  Le seuil s est choisi suffisament petit pour tenir dans le cache (i.e. $s^2 + 2s < Z_1$) et suffisament grand pour amortir le surcoût des appels de découpe récursive ; typiquement $s = 10$ ou 20.

En supposant chacun des Z_i de taille plus grande que s^2+2s, le nombre de défauts de cache au niveau i est similaire à l'algorithme cache aware, soit proche de $\frac{nm}{L_i}$ si $L_i \ll \sqrt{Z_i}$.

— **Conclusion** : par rapport à l'algorithme cache oblivious itératif de la question précédente qui fait $2nm/L$ défauts (nm/L sur A et nm/L sur T), ce programme récursif avec seuil est plus compliqué à règler ; de plus, si L_i est proche de $\sqrt{Z_i}$, il est moins bon en nombre de défauts que l'algorithme cache oblivious itératif de la question 3.
En pratique, sur une hiérarchie mémoire, même si il peut fair un peu moins de défaut de cacche (en prie cas jusqu'à presque 2 fois moins), on s'attend à ce qu'il soit moins performant que l'algorithme cache oblivious itératif de la question 3 qui est plus simple.

5 Mergesort - Tri par fusion (15')

On considère un cache de taille Z mots accèdé par lignes de cache de taille L mots et géré par LRU (modèle CO).

Le programme C ci-dessous implémente un tri par fusion du tableau `a` de n mots ; il utilise un tableau auxiliaire b de n mots préalloué.

```
1 void merging(int low, int mid, int high) { /* Fusion de a[low .. mid] et a[mid+1 .. high] */
2    int l1, l2, i;
3    for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
4       if(a[l1] <= a[l2]) b[i] = a[l1++];
5       else b[i] = a[l2++];
6    }
7    while(l1 <= mid)    b[i++] = a[l1++];
8    while(l2 <= high)   b[i++] = a[l2++];
9    for(i = low; i <= high; i++) a[i] = b[i];
10 }
11
12 void sort(int low, int high) { /* Tri recursif de a[low .. high]  */
13    if(low < high) {
14       int mid = (low + high) / 2;
15       sort(low, mid);
16       sort(mid+1, high);
17       merging(low, mid, high);
18   }
19 }
```

Question 11

1. On suppose que les sous-tableaux `a[low .. high]` et `b[low .. high]` sont déjà dans le cache. Combien de défauts de cache fait `merging` ?
2. Même question en supposant qu'aucun élément des 2 sous-tableaux n'est déjà dans le cache.
3. En déduire une majoration du nombre de défauts de cache de `sort` sur le modèle CO.

1. #défauts = 0 ;
2. soit k = high - low ; les lignes 4 à 9 parcourent linéairement les 3 tableaux : a[low..mid] ; a[mid+1..high] ; b[low..high]. Soit k = high - low ; alors $2\left\lceil \frac{k}{L} \right\rceil \leq$ #défaut $\leq 2\left\lceil \frac{k}{L} \right\rceil + 3$; i.e. $2k/L$ défauts de cache.
3. $2\frac{n}{Z/2} \times \frac{Z/2}{L} + \frac{2n}{L}\log_2 \frac{n}{Z/2} \simeq \frac{2n}{L}(\log_2 n - \log_2 Z)$.