TD 2 d'Algorithmique et Optimisation Discrète

Dépendances et localité (blocking)

Équipe pédagogique AOD

Résumé du cours : technique de blocking

La localité spatiale et temporelle a un impact sur les performances en particulier pour des programmes manipulant des tableaux. Lorsque cela est possible, il faut privilégier un parcours linéaire du tableau en suivant l'ordre de stockage des éléments, et ce quel que soit le nombre d de dimensions. En C, Python ou Java les tableaux sont stockés par lignes (*Row-Major*) : il faut faire attention à choisir l'ordre des dimensions pour que les boucles les plus internes accèdent des éléments stockés de manière contigüe. Par exemple, considérons un nid de $D \geq 2$ boucles :

```
T memo[n_1][n_...][n_D] ; // un tableau de valeurs
for (i_1 = 0; i_1 < n_1 ; ++i_1)
   ...
      for ( i_D = 0 ; i_D < n_D; ++i_D )  {
         ...
         memo [i_1][...][I_D] =  calcul ou mise à jour de la valeur à partir de memo[
             ... ] ...
    }
```

Ici, le stockage choisi pour le tableau `memo`$[n_1][..][n_D]$ garantit que la boucle interne sur i_D écrit des éléments contigus (localité spatiale) ; sur un cache petit par rapport à n_D et avec une taille L de ligne de cache, le nombre de défauts de cache (en écriture) est alors proche de $\left(\prod_{k=1}^{D-1} n_k\right) \times \lceil \frac{n_D}{L} \rceil \simeq \frac{\prod_{k=1}^{D} n_k}{L}$ donc asymptotiquement optimal. Mais d'autres défauts de cache sont à considérer, en particulier ceux en lecture sur les accès aux valeurs mémoisées. Si un parcours par dimensions génère trop de défauts de cache, la technique de ***blocking*** consiste à améliorer la localité en effectuant un parcours de l'espace d'itération ou du tableau ***par blocs*** tenant en cache.

Analyse des dépendances écriture-lecture L'ordre partiel entre les instructions du programme se représente par un graphe dit graphe de dépendances (ou graphe *dataflow*) : ce graphe orienté sans circuit (DAG) décrit les dépendances *écriture-lecture* entre instructions. Les nœuds sont les instructions (ou bloc d'instructions comme par exemple l'appel à une fonction, on parle alors de graphe macro dataflow) ; il y a un arc de l'instruction i_1 à l'instruction i_2 ssi l'instruction i_2 *lit* une valeur en mémoire qui peut avoir été *écrite* par l'instruction i_1. La fermeture transitive de ce graphe est l'ordre partiel que toute exécution du programme doit vérifier pour en respecter la sémantique. Parmi tous les ordonnancements qui respectent cet ordre partiel (i.e. le graphe), il s'agit alors de trouver et programmer ceux qui ont une bonne localité.

Blocking itératif cache-aware : on choisit la taille $K_1 \times \cdots \times K_D$ des blocs au mieux pour que les données accédées dans un bloc d'itération tiennent dans le cache de taille Z ; ou, de manière alternative, on peut aussi parcourir le tableau par blocs de taille $\Theta(L)$ sur chaque dimension. Dans ces 2 cas, le programme utilise alors les paramètres du cache (typiquement L ou Z) pour définir la taille des blocs. Le programme par blocs s'écrit comme suit : les indices I_k parcourent de bloc en bloc et i_k à l'intérieur d'un bloc :

```
for (I_1=0 ;  I_1 < n_1 ; I_1 += K_1)
{
   int end_1 = ((I_1 + K_1 < n_1) ? I_1 + K_1 : n_1 ) ;
   ...
       for (I_D = 0 ; I_D < n_D; I_D += K_D )
       {
          int end_D = ((I_D + K_D < n_D) ? I_D + K_D : n_D ) ;
          computeBlock( I_1, end_1, ... I_D, end_D ) ;
}  ... }

void computeBlock( int begin_1, int end_1, ..., int begin_D, int end_D )
{  // Calcul séquentiel  du bloc qui tient dans le cache !
```

```
    for (i_1 = begin_1; i_1 < end_1 ; ++i_1)
          ...
          for (i_D = begin_D; i_D < end_D ; ++i_D)
          {  ...
             // accès à memo [i_1][...][I_D] ...
}         }
```

Blocking récursif cache-oblivious : Une découpe récursive de l'espace d'itération en blocs de grande taille selon la plus grande dimension permet l'écriture d'un programme *cache-oblivious* indépendant des paramètres du cache et qui se comporte de manière similaire pour toutes valeurs de L et Z : une fois qu'un bloc tient dans le cache, les autres découpes récursives de ce bloc le gardent en cache (localité spatiale et temporelle). Ainsi, cette découpe récursive permet d'exploiter une hiérarchie de cache, en apportant implicitement une bonne localité à chaque niveau. Pour éviter le surcout arithmétique dû aux appels récursifs (comme l'empilement des paramètres des appels récursifs sur la pile), il ne faut pas oublier d'arrêter la récursivité à un seuil en dessous duquel on procède à un calcul itératif sur le bloc (ou par blocs de petite taille).

```
#define S ....  // Seuil d'arrêt récursif à définir

// Sur les indices : un suffixe b correspond à begin, e à end.
void blockingRec(int begin_1, int end_1, ..., int begin_D, int end_D )
{
    int n_1 = end_1 - begin_1 ; ... ; int n_D = end_D - begin_D ;
    if (( n_1 <= S ) && ... && (n_D <= S))
      computeBlock( begin_1, end_1, ... begin_D, end_D ) ;
    else // Découpe récursive
    { // ici par exemple, découpe en 2 de la plus grande dimension notée k
      int mid_k = (begin_k + end_k) / 2 ;
      blockingRec(int begin_1, int end_1,..., begin_k, mid_k,...,begin_D, end_D);
      blockingRec(int begin_1, int end_1,..., mid_k,   end_k,...,begin_D, end_D);
}  }
```

Blocking et parallélisme : Outre la localité, de tels programmes par blocs sont en général bien adaptés au parallélisme : la structuration en blocs a permis de regrouper ensemble les calculs portant sur les mêmes instructions et les mêmes données ; un effet de bord qui peut apparaître (et en pratique qui apparait souvent en particulier en programmation dynamique) est de mettre en évidence des blocs de calcul (de grain assez gros) n'ayant pas de dépendances entre eux et pouvant s'exécuter donc en parallèle. Ce parallélisme de gros grain peut être exploité efficacement en pratique. En cache aware, le niveau de granularité est celui du bloc défini par K en fonction de taille Z du cache. En cache oblivious, la découpe est récursive, le niveau de granularité le plus fin étant celui défini par S le seuil de découpe récursive. En contrôlant la récursivité, on peut adapter le grain, le plus gros étant au niveau des appels récursifs les plus externes.

En dégageant des blocs de calcul récursifs parallèles et relativement gros en nombre d'opérations, ce type de parallélisme peut être facilement exploité dans des environnements de programmation parallèle adaptés aux calcul récursifs. Par exemple, en OpenMP, on rend parallèle les 2 appels récursifs ci-dessus en écrivant :

```
      #pragma omp task   // pour paralléliser le calcul (la moitié, récursivement)
      { blockingRec(int begin_1, int end_1,..., begin_k, mid_k,...,begin_D, end_D);
      }
      blockingRec(int begin_1, int end_1,..., mid_k,   end_k,...,begin_D, end_D);
```

En pratique, on limite parfois le parallélisme à des blocs de taille supérieure à un seuil (i.e. `mid_K - end_K > S_PAR`).

1 Parcours de tableaux cache oblivious (40')

Question 1 Soit une matrice A de taille $n \times m$; le programme ci-dessous calcule les tableau S de taille n (resp. T de taille m) qui contient la valeur maximale de chaque ligne (resp. minimale de chaque colonne) de A :

$$\forall i = 0 \dots n-1 : S_i = \max_{j=0\dots m-1} A_{i,j} \qquad \forall j = 0 \dots m-1 : T_j = \min_{i=0\dots n-1} A_{i,j}$$

```
// Version par indices


S[0] = T[0] = A[0][0] ;
int j;
for (j=1; j<m ; ++j) // cas i=0
{
  T[j]= A[0][j] ;
  if (S[0] < A[0][j]) S[0] = A[0][j];
}
int i;
for (i=1; i<n ; ++i)  // cas i>=1
{
  S[i] = A[i][0] ;
  if (T[0] > A[i][0]) T[0] = A[i][0];
  int j;
  for (j=1; j<m ; ++j) // cas j>=1
  {
    double v = A[i][j] ;
    if (S[i] < v) S[i] = v
    if (T[j] > v) T[j] = v ;
  }
}
```

```
// Version par itérateur (++pointeur)
double* ptA = (double*)A ;
double* endS = S+n; double* endT = T+m;
S[0] = T[0] = A[0][0] ;
double* ptT;
for(ptT=T+1, ++ptA; ptT<endT; ++ptT, ++ptA)
{
  *ptT = *ptA ;
  if (S[0] < *ptA) S[0] = *ptA ;
}
double* ptS ;
for (ptS=S+1; ptS<endS; ++ptS )
{
  *ptS = *ptA ;
  if  (T[0] > *ptA) T[0] = *ptA ;
  double *ptT=T ;
  for(++ptT,++ptA; ptT<endT; ++ptT,++ptA)
  {
    double v = *ptA ;
    if (*ptS < v) *ptS = v ;
    if (*ptT > v) *ptT = v ;
  }
}
```

On considère un cache de taille Z avec une taille L de ligne de cache – $Z = \Omega\left(L^2\right)$ – et politique LRU (modèle CO). On suppose que les tableaux A, S et T sont alignés en mémoire (leur adresse de début est un multiple de L).

1. Quel est le nombre de *défauts de cache obligatoires* (i.e. avec Z suffisament grand pour tout contenir) ?
2. Dans la suite on suppose le cache très petit, de taille insuffisante pour stocker le tableau T (i.e. $m \gg Z$). Justifier qu'il y a $\frac{nm}{L}$ défauts sur A, $\frac{n}{L}$ sur S et $\frac{nm}{L}$ sur T ; soit en tout environ $\frac{2nm}{L} + \frac{n}{L} \simeq 2\frac{nm}{L}$ défauts de cache.
3. Le programme ci-dessus est-il : a/ cache aware ; b/ cache oblivious ; c/ ni l'un ni l'autre ?
4. Ecrire un programme *cache-aware* qui fait un nombre de défauts de cache inférieur à $(1+\varepsilon)\frac{nm}{L} + O\left(\frac{n+m}{L}\right)$ où ε est petit quand $Z \gg L^2$ est grand.
5. En déduire un programme *cache oblivious* qui fait environ le même nombre de défauts : on demande juste de décrire le principe du programme, mais pas d'écrire le programme.
6. En pratique, sur une hiérarchie mémoire, jugez-vous ce programme plus pertinent que celui de l'énoncé ?

2 Filtrage d'une image bidimensionnelle (15')

Question 2 Soit une image rectangulaire stockée dans un tableau A de $n \times m$ pixels). On calcule une image B en appliquant un filtre sur A : chaque pixel intérieur $A_{i,j}$ avec $0 < i < n-1$ et $0 < j < m-1$ est mise à jour avec la valeur de ses 4 voisins (Nord, Sud, Est, Ouest). Par exemple :

$$B_{i,j} = \frac{4 \times A_{i,j} + A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1}}{8}$$

(On ignore les conditions aux bords.)

1. Quel est le nombre de défauts de cache obligatoires sur A ?
2. Comment programmer le parcours ?
3. Quel est le nombre de défauts de ce parcours si une ligne de A ne tient pas en cache ?
4. Cache-aware : décrire un algorithme par blocs qui fait un nombre de défauts de cache proche de nm/L.
5. Cache oblivious : comment programmez-vous sur une mémoire héierarchique et quel est le nombre de défauts à chaque niveau i de la hiérarchie ?

3 Jeu de la vie et filtre de Jacobi

Un filtre par voisinage d'un tableau multidimensionnel consiste à calculer pour chaque élément du tableau la moyenne pondérée de ses voisins. Exemples en dimension 2 :

— le jeu de la vie : l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines ;
— la résolution de l'équation de la chaleur : par la méthode de Jacobi sur le laplacien discrétisé ;
— la détection des contours d'une image : un contour est caractérisé par un changement brutal de la valeur d'une pixel et peut être estimé avec les 4 valeurs voisines.

On considère ici l'application d'un filtre multi-pas à un tableau A circulaire de taille n : la valeur A_i^t au pas t est calculée à partir des valeurs $A_{i-1}^{t-1}, A_i^{t-1}, A_{i+1}^{t-1}$ au pas $t-1$. Par exemple, un filtre médian s'écrit

$$A_i^t = \frac{A_{i-1}^{t-1} + A_i^{t-1} + A_{i+1}^{t-1}}{3}.$$

Le programme suivant fait le calcul en place dans A pour m pas de temps :

```
{  for (size_t t=0; t<m ; ++t)
   {  Element origine = A[0] ;
      Element old = A[0] ;
      A[0] = f( A[n-1], A[0], A[1] ) ;
      for (int i=1; i<n-1; ++i)  // pas t
      {  Element aux = A[i] ;
         A[i] = f( old, A[i], A[i+1] ) ;
         old = aux;
      }
      A[n-1] = f(old, A[n-1], origine) ;
}  }
```

Question 3 Selon que les données tiennent ou non en cache, analyser le nombre de défauts de cache sur un cache de taille Z chargé par ligne de cache de taille L.

Question 4 Pour $0 \leq i < n$ et $0 \leq t < m$, on note (i,t) l'instruction qui écrit A_i^t. Ainsi, dans le programme ci-dessus au pas t, $(i, 2t)$ – resp. $(i, 2t+1)$ — est l'instruction qui calcule `A[`i`]` – resp. `tmp[`i`]` –.

1. Dessiner le graphe de dépendances entre instructions : il y a un arc de (i_1, t_1) à (i_2, t_2) ssi (i_2, t_2) lit une valeur écrite par (i_1, t_1).
2. Quelle est la profondeur de ce graphe – ou temps parallèle du programme – en nombre d'instructions (i,t) ?
3. Pour $n > 5$, parmi les instructions suivantes, lesquelles sont parallèles : $(1,1); (3,1); (3,2)$?
4. Quelles instructions $(j, 0)$ précèdent l'instruction (i,t) ?

Question 5 Proposer un ordonnancement des instructions cache aware : indiquer sur un schéma comment regrouper les instructions et donner le nombre de défauts de cache (on ne demande pas le programme).

Question 6 Donner le principe d'un programme cache oblivious ; quel est son nombre de défauts de cache ?

Question 7 Au lieu de filtrer A circulairement, on considère ici que les conditions aux bords A_0^t et A_{n-1}^t sont imposées (constantes). Cela ajoute-t-il ou enlève-t-il des dépendances entre instructions ? Quel est l'impact pour une programmation cache oblivious ?

Question 8 On filtre maintenant un tableau bidimensionnel $n_1 \times n_2$ (i.e. une image) en supposant que la mise à jour d'un élément utilise les valeurs de ses 4 voisins (Nord, Est, Sud, Ouest) ; les conditions aux bords de l'image sont fixées. Quel est le principe d'un algorithme cache oblivious et combien ferait-il de défauts de cache :

1. si $m = 1$ (i.e. il y a une seule passe sur l'image) ?
2. si m est grand ?