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 ?

1. défauts obligatoires $= \left\lceil \frac{nm}{L} \right\rceil + \left\lceil \frac{n}{L} \right\rceil + \left\lceil \frac{m}{L} \right\rceil$ si A, S et T sont alignés.
2. A est parcouru une fois dans le sens de stockage ; idem pour S ; T est parcouru n fois dans le sens du stockage.
3. cache oblivious : en effet, $Q(n,m,L,Z) = 2\frac{nm}{L} = \Theta\left(\frac{nm}{L}\right)$: le nombre de défauts de cache est de l'ordre du nombre de défauts obligatoires, donc nécessairement a au plus un facteur constant (i.e. indépendant de L et Z) de l'optimal.
4. *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 utilise une technique de blocking pour diminuer le nombre de défauts sur T quitte à augmenter le nombre de défauts sur S : on fait un calcul par *blocs de calcul* correspondant à K_l lignes et K_c colonnes avec K_l et K_c choisis pour que : d'une part toutes les données accédées dans un bloc de calcul tiennent en cache ; d'autre par minimiser les défauts de cache.
   On prend garde à préserver $\frac{nm}{L}$ défauts sur A : pour que le parcours interne d'un bloc reste raisonnable sur les niveaux de cache inférieurs, on parcourt l'intérieur du bloc de A par ligne, puis on passe au bloc suivant de A sur les mêmes lignes. Ainsi lorsqu'on accède le premier élément du bloc suivant de A, sa ligne de cache – qui est encore en cache puisque le bloc précédent tenait en cache – est rajeunie et donc restera en cache (ce sont les lignes de cache de la même ligne de A et qui précèdent celle de cet élément qui seront prioritairement écrasées dans le cache). Ainsi, il y a $\frac{nm}{L}$ défauts seulement sur A. Le programme général s'écrit :

```
for (int i=0; i<n ; ++i) S[i]= -MAXFLOAT ; // Initialisation S
for (int j=0; j<m ; ++j) T[j]= +MAXFLOAT ;// Initialisation T
for (int I=0; I<n ; I+=K_l )  // Parcours de A par blocs lignes
{  int i_end = (n < I+K_l) ? n : I+K_l ;
   for (int J=0; J<m ; J+=K_c )
   {  // Bloc de calcul interne qui doit tenir en cache
      int j_end = (m < J+K_c) ? m : J+K_c ;
      for (int i=I; i < i_end ; ++i) // Parcours interne du bloc par colonnes
      {  for (int j=J; j < j_end ; ++j)
         {  double v = A[i][j] ;
            if (S[i] < v) S[i] = v ;
            if (T[j] > v) T[j] = v ;
}  }  }  }
```

Contrainte pour qu'un bloc interne tienne en cache. Une séquence de k éléments (i.e. mots unité de cache) occupent en cache une place k en meilleur cas et $k+2L-2$ en pire cas (lorsque le le premier (resp. dernier) élément des k est le dernier (resp. premier) d'une ligne de cache. Or un bloc de calcul accède K_l sous-lignes de K_c éléments de A ainsi que K_c éléments de T et K_l de S. Donc la place mémoire occupée en cache pour un bloc est comprise entre $K_l \times K_c + K_l + K_c$ et $K_l \times (K_c + 2L - 2) + K_l + K_c + 4L - 4$. On choisit donc K_l et K_c tels que

$$K_l \times (K_c + 2L - 2) + K_l + K_c + 4L - 4 + O(1) \leq Z. \tag{1}$$

Analyse des défauts $Q(n, m, L, Z)$. Deux analyses sont faites ci-dessous.

Analyse grossière. On ne prend pas ici en compte l'ordre de parcours des blocs (les boucles sur I et J pourraient être inversées par exemple). On a $\frac{nm}{K_l K_c}$ blocs de calcul faisant chacun $K_l \frac{K_c}{L}$ défauts sur A et $\frac{K_l}{L}$ sur S et $\frac{K_c}{L}$ sur T. Donc en tout $\frac{nm}{L}\left(1 + \frac{1}{K_c} + \frac{1}{K_l}\right)$ défauts. Cette quantité est minimisée en prenant $K_l = K_c$ le plus grand possible sous la contrainte 1, soit $K_l = K_c = \sqrt{Z} - O(1)$. Le nombre de défauts de cache est alors $\frac{nm}{L}\left(1 + O\left(\frac{1}{\sqrt{Z}}\right)\right)$, soit le résultat esscompté.

Analyse plus fine en prenant en compte l'ordre `for I for J` de parcours des blocs. On a :

— $\frac{nm}{L}$ défauts sur A car 2 blocs de A parcouris consécutivement sont contigus en mémoire (pour chacune des lignes du bloc) ; un bloc tenant en cache, il n'y a donc pas défaut de cache additionnel sur A seulement des défauts obligatoires)L

— $\frac{m}{K_c}\frac{n}{L}$ défauts sur S (car le parcours par bloc de lignes fait que chaque ligne de cache de S est accédée $\frac{m}{K_c}$ fois après initialisation)

— $\frac{n}{K_l}\frac{m}{L}$ défauts sur T (car le parcours par bloc de lignes fait que chaque ligne de cache de T est accédée $\frac{n}{K_l}$ fois après initialisation)

soit au total $Q = \frac{nm}{L} + \frac{n}{L} + \frac{nm}{LK_l} = \frac{nm}{L}\left(1 + \frac{1}{K_l} + \frac{1}{m}\right)$. Pour minimiser le nombre de défauts, on choisit K_l maximum, donc K_c minimum. D'où $K_c = 1$ et $K_l(1 + L) = Z - 2L - O(1)$ soit $K_l = \frac{Z-2L-O(1)}{L+1} \simeq \frac{Z}{L+1}$. Le nombre de défauts est alors $Q = \frac{nm}{L}\left(1 + \frac{L+1}{Z} + \frac{1}{m}\right) = \frac{nm}{L}(1 + \epsilon\ )$ avec $\epsilon < 1$ car $Z = \Omega\left(L^2\right)$.
Comme $K_c = 1$, la boucle la plus interne du programme générique ci-dessus est supprimée et on aboutit au programme cache aware suivant (en intégrant l'initialisation de S et T dans les boucles sans surcoût) :

```
#define K_l  ( Z / (L+1) )
T[0] = A[0][0] ; // Initialisation T[0]
for (int I=0; I<n ; I+= K_l )  // Parcours de A par blocs lignes
{  int i_end = (n < I+K_l) ? n : I+K_l ;
   int i_begin = I ;
   // boucle calcul en cache : for (int J=0; J<m ; J+=1 ) { for (int i = i_begin ; i<
       i_end; ++i) { ...}
   for (int i=i_begin ; i < i_end ; ++i) // traitement initailisaton cas J == 0
   {  S[i] = A[i][0] ; //  Initialisation S[i]
      if ( T[0] > A[i][0] ) T[0] = A[i][0] ; // Maj T[0]
   }
   for (int J=1; J<m ; J+=1 ) // Parcours interne par colonnes du bloc lignes
   {  if (i_begin == 0) // traitement initialisation cas i_begin=0
      {  double v = ( T[J] = A[0][J] ) ; // initialisation T[J]
         if (S[0] < v) S[0] = v;         // Maj S[0]
```

```
            i_begin = 1 ;
         }
         for (int i= i_begin; i < i_end ; ++i)
         {  double v = A[i][J] ; /* Comme K_c = 1, on a j=J ! */
            if (S[i] < v) S[i] = v ;
            if (T[J] > v) T[J] = v ;
   }  }  }
```

5.

```
void AST_cacheoblivious(double*A, int n, int m, double* S, double *T )
{
   #define Grain  200  /* Pour amortir le cout d'un appel recursif par rapport à 2
       comparaisons de double  */
   int seuil = 1 + Grain / m ;

   void split_lines (int i_begin, int i_end) // procedure recursive qui splitte en blocs
       de lignes de taille S
   {  if ((i_end - i_begin) > seuil ) // Découpe recursive en 2 par lignes
      {  int i_middle = (i_begin + i_end) / 2 ;
         split_lines ( i_begin, i_middle) ;
         split_lines ( i_middle, i_last) ;
      }
      else //  Arret des appels recursifs  car nb lignes <=  seuil )
      {  // boucle calcul en cache : for (int J=0; J<m ; J+=1 ) { for (int i = i_begin ; i
          < i_end; ++i) { ...}
         // Traitement du cas J=0 avec les initialisations pour le bloc ligne
         for (int i=i_begin ; i < i_end ; ++i) // cas J == 0
         {  S[i] = A[i][0] ; //  Initialisation S[i]
            if ( T[0] > A[i][0] ) T[0] = A[i][0] ; // Maj T[0]
         }
         for (int J=1; J<m ; J+=1 ) // Parcours interne par colonnes du bloc lignes
         {  if (i_begin == 0) // traitement initialisation T[j] et maj S[0]
            {  double v = ( T[J] = A[0][J] ) ;
               if (S[0] < v) S[0] = v;
               i_begin = 1 ;
            }
            for (int i= i_begin; i < i_end ; ++i)
            {  double v = A[i][j] ; /* Comme K_c = 1, on a j=J ! */
               if (S[i] < v) S[i] = v ;
               if (T[j] > v) T[j] = v ;
   }  }  }  }
{  // Programme principal pour AST_cacheoblivious
   T[0] = A[0][0] ; // Initialisation
   split_lines ( 0, n ) ; // Appel principal
}
```

6. **Conclusion** : l'algorithme cache oblivious itératif de l'énoncé marche bien à tout niveau de la hiérarchie : sans surcoût arithmétique, il fait moins de deux fois le nombre de défauts obligatoires.
   L'algorithme cache aware a peu de surcoût arithmétique mais ne marche qu'à un seul niveau de cache et divise au plus par 2 le nombre de défauts.
   L'algorithme cache oblivious a un surcût arithmétique important qu'il faut amortir en choisissant un bon seuil d'arrêt s et ne divise qu'au plus par 2 le nombre de défauts sur les caches de taille supérieur à s^2.
   En pratique, le programme de l'énoncé semble plus pertinent.

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 ?

1. nm/L
2.
```
for (i=1; i<n; ++i)
  for (j=1; j<m; ++j)
    B(i,j) =  ( 4*A(i,j)  + A(i-1, j) + A(i+1, j) +A(i, j-1) + A(i, j+1)  ) / 8
```
3. #défauts $\simeq$ $3nm/L$
4. découpe en blocs de taille $K \times K$ qui rentre dans le cache, donc K proche de $\sqrt{Z}$ (il faut $K^2 + 2K + 2KL \leq Z$). Le nombre de défauts est $\frac{nm}{L}\left(1 + O\left(\frac{L}{\sqrt{Z}}\right)\right)$
5. découpe récursive sur la plus grande dimension : cela correspond à un parcours (récursif) en Z de l'image.

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.

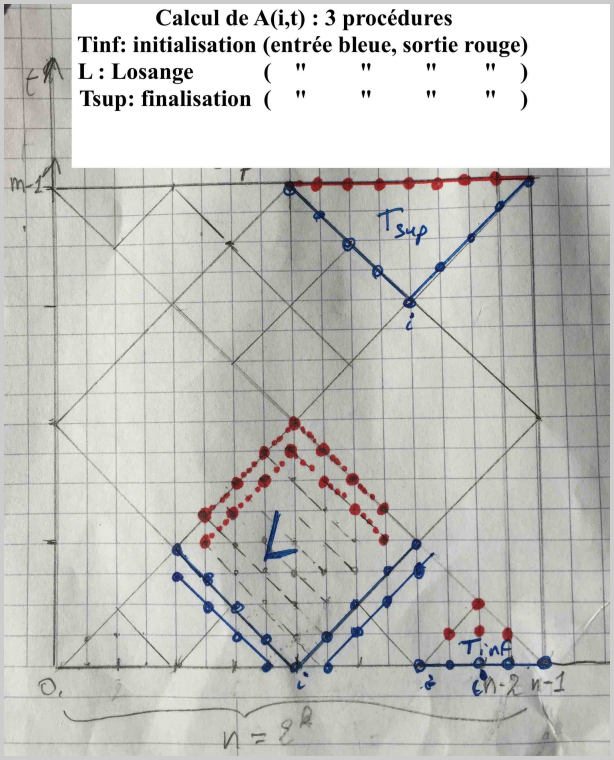
On analyse le programme ci-dessus en place dans A :
— Si les données tiennent en cache (i.e. $Z \gg n$) : $Q(n, L, Z) = \frac{n}{L} + O(1) = \Theta\left(\frac{n}{L}\right)$;
— Sinon (i.e. $Z \ll n$) : $Q(n, L, Z) = \frac{nm}{L} + O(1) = \Theta\left(\frac{nm}{L}\right)$; donc m fois plus grand !

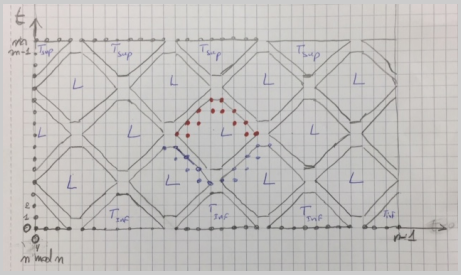
Question 4 Pour $0 \leq i < n$ et $0 \leq t < m$, on note (i, t) l'instruction qui écrit A_i^t. Ainsi, dans le programme ci-dessus au pas t, $(i, 2t)$ –resp. $(i, 2t+1)$ — est l'instruction qui calcule `A[i]` –resp. `tmp[i]` –.

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) ?

1. Chaque (i, t) a 3 arcs entrants : $((i-1) \mod n, t-1); (i, t-1); ((i+1) \mod n)$.
2. La profondeur (temps parallèle ou chmin critique) est m écritures (ou divisions par 3).
3. seul $(3, 2)$ doit s'exécuter après $(3, 1)$; de fait $(1, 1)$ est parallèle avec $(3, 1)$ et $(3, 2)$.
4. (i, t) dépend de $(j, 0)$ pour $j = (i-t) \mod n, \ldots, (i+t) \mod n$.

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).





Deux exemples de pavage disjoint possible des instructions : à gauche celui utilisé par les programmes ci-dessous.
Les 3 types de bloc d'instructions : Losange (L), Triangle inférieur (T_{inf}) et supérieur (T_{sup}).
Orientation : Nord en haut, Sud en bas, Ouest à gauche, Est à droite.
Dans le pavage de gauche, les fontieres NE et NO sont incluses dans l'élément mais pas les frontières SE et SO.

Pour simplifier la programmation, on utilise un tableau bidimensionnel A_i^t : mais le calcul se fera en place dans le tableau A en utilisant en plus un tableau temporaire de taille n.
On pave l'ensemble des instructions $A(i,t)$ pour respecter les dépendances ; le pavahe est disjoint : une même instruction $A(i,t)$ n'est pas dupliquée, son résultat est stocké -temporairement- en mémoire.
On considère le pavage indiqué à gauche dans la figure **??** : le bloc de calcul générique est un losange $K \times K$ construit à partir des 2 points inférieurs (i,t) et $(i+1,t)$: si l'instruction coin inférieur de ce losange est (i,t), alors le losange contient les instructions $(i-j,t+j),\dots(i+1+j,t+j)$ pour $j=0\dots K-1$ et les instructions $(i-j,t+2K-j),\dots(i+1+j,t+2K-j)$ pour $j=0\dots K-2$; et toutes les instructions à l'intérieur de ce losange.

- Entrées : frontières sud-est et sud-ouest : 4 tableaux de K points, 2 stockés dans le tableau en entrée A et 2 dans un tableau temporaire T de taille n. Après toute écriture de A_i^t dans la case $A[i]$, la case $T[i]$ contient A_i^{t-1}.
- Sorties : frontières nord-est et nord-ouest : 4 tableaux de K points calculés en place dans les entrées.

Les autres blocs de calcul sont :

- n/K triangles inférieurs (calculés en premier) : ils initialisent les frontières (dans A et T) à partir des valeurs initiales de A.
- n/K triangles supérieurs (calculés en dernier à la fin de l'algorithme).

Chaque bloc manipule $2K$ données en lecture et écriture (calcul en place) ; pour qu'un bloc tienne en cache on choisit $K \simeq Z/2$. Pour simplifier l'algorithme, on suppose que K divise n et $m \mod K = 1$ et que toutes les additions d'indice dans les tableaux sont faites modulo n.

```
1  //****************************** Procedure principale
2  void JacobiCacheAware(Element* A, int n, int K)
3  {  //Initalisation des tableaux T et A pour éviter les modulos
4     Element* T= (Element*) malloc( n*sizeof(Element)) ; // temporaire
5     memcpy( T, A, n*sizeof(Element) ) ;
6     for (size_t i=0; i <n ; i+=2*K)  Tinf( i, K, A, T) ; // initialisation, parallèle!
7     for( size_t t=0; t< m/K-1; ++t )  // Itérations losange par macro-pas, séquentielle
8     {  // Calcul des losanges à partir du point SW : A((t+K)%n , t+1)
9        for ( size_t i= ( ((t%2)==0) ? K : 0 ); i<n; i+=2*K ) Losange( i, K, A, T) ; // parallel
             for à t
10    }
11    for (int i= ((((m/K)%2)==0) ? 0 : K ); i <n ; i+=2*K)  Tsup(i, K, A, T) ; // parallel for
12    free(T) ;
13 }
```

```
//****************************** Bloc Losange
void Losange ( size_t i, size_t K, Element* A, Element* T )
/* Calcul du Losange carre de coté K qui contient les points de A[i-K .. i+K];
 * Entrée : - la      diagonale SW : A(i,t), A(i-1, t+1), ... A(i-K, t+K)
 *                                   stockée dans A[i .. i-K]
 *          - la sous-diagonale SW : A(i,t-1), A(i-1, t), ... A(i-K, t+K-1)
 *                                   stockée dans T[i .. i-K]
 *          - la      diagonale SE : A(i,t), A(i+1, t+1), ... A(i+K, t+K)
 *                                   stockée dans A[i .. i+K]
 *          - la sous-diagonale SE : A(i,t-1), A(i+1, t), ... A(i+K, t+K-1)
 *                                   stockée dans T[i .. i+K]
 * Sortie : - la      diagonale NW : A(i,t+2K), A(i-1, t+2K-1), ... A(i-K+1, t+K+1)
 *                                   stockée dans A[i .. i-K]
 *          - la sous-diagonale NW : A(i,t+2K-1), A(i-1, t+2K-2), ... A(i-K+1, t+K)
 *                                   stockée dans T[i .. i-K]
 *          - la      diagonale NE : A(i,t+2K), A(i+1, t+2K-1), ... A(i+K-1, t+K+1)
 *                                   stockée dans A[i .. i+K]
 *          - la sous-diagonale NE : A(i,t+2K-1), A(i+1, t+2K-2), ... A(i+K-1, t+K)
 *                                   stockée dans T[i .. i+K]
 * NB Le calcul est en place:
 * si A(i,t) est stocké en place dans A[i] alors A(i,t-1) en place dans T[i].
 */
{
   for ( size_t t=0; t<K; ++t)  // 2K diagonales de K points à calculer, de SW à NE
   {  int orig = i + t;
      for (int k=orig; k>orig-(int)K; --k) // Calcul des elements de diag et sous-diag SW issue
          de orig
      {  int j = mod(k,n) ;
         T[j]= f(T[mod((j-1), n) ], A[j], T[mod((j+1), n) ]) ; // sous-diag
         A[j]= f(A[mod((j-1), n) ], T[j], A[mod((j+1), n) ]) ; // diag
}  }  }
```

```
//****************************** Bloc Tinf
void Tinf( size_t i, size_t K, Element* A, Element* T )
/* Calcul du triangle inferieur (horizontale en bas, S), partant du point central S i
 * Entrée : - l'horizontale S : A(i-K,0),  ... A(i+K, 0)
 *                                   stockée dans A[i-K .. i+K]
 * Sortie : - la      diagonale NW : A(i-K+1, 1), A(i-K+2, 2), ... A(i, K)
 *                                   stockée dans A[i-K+1 .. i]
 *          - la sous-diagonale NW : A(i-K+1, 0), A(i-K+2, 1), ... A(i, K-1)
 *                                   stockée dans T[i-K+1 .. i]
 *          - la      diagonale NE : A(i,K), A(i+1, K-1), ... A(i+K-1, 1)
 *                                   stockée dans A[i .. i+K-1]
 *          - la sous-diagonale NE : A(i,K-1), A(i+1, K-2), ... A(i+K-1, 0)
 *                                   stockée dans T[i .. i+K-1]
 */
{  int nb = 2*K - 1  ; // nombre de points à calculer sur la ligne
   int orig = i - K+1 ; // abscisse du premier point à calculer
   for ( ; (nb > 0) ; orig+=1, nb-=2 )
   {  Element pred = A[mod(orig-1,n)]; // pour calcul en place
      for (int k=0; k <  nb; ++k ) // Mise a jour ligne
      {  int j = mod(orig+k, n) ;
         T[j]=A[j];
         A[j] = f( pred, T[j], A[mod(j+1, n)] ) ;
         pred=T[j] ;
}  }  }
```

```
//****************************** Bloc Tsup
void Tsup( size_t i, size_t K, Element* A, Element* T )
/* Calcul du triangle superieur (horizontale en haut, N) partant du point S i
 * Entrée : - la      diagonale SW : A(i,t), A(i-1, t+1), ... A(i-K, t+K)
 *                                   stockée dans A[i .. i-K]
 *          - la sous-diagonale SW : A(i,t-1), A(i-1, t), ... A(i-K, t+K-1)
 *                                   stockée dans T[i .. i-K]
 *          - la      diagonale SE : A(i,t), A(i+1, t+1), ... A(i+K, t+K)
```

```
 *                                    stockée dans A[i .. i+K]
 *          - la sous-diagonale SE : A(i,t-1), A(i+1, t), ... A(i+K, t+K-1)
 *                                    stockée dans T[i .. i+K]
 * Sortie : - l'horizontale N : A(i-K,t+K),  ... A(i+K, t+K)
 *                                    stockée dans A[i-K .. i+K]
 */
{
   for ( size_t t=0; t<K; ++t)  // 2K diagonales tronquees à calculer, de SW à NE
   {  int orig = i + t;
      int k=orig; // Les 2 diag partent de k: la 1ere diag a nbpts , la suivante nbpts-1
      int nbpts=K-t ;
      for (nbpts=K-t-1; nbpts>0; --k, --nbpts) // Calcul des elements de diag et sous-diag SW
          issue de orig
      {  int j = mod(k,n) ;
         T[j]= f(T[mod((j-1), n) ], A[j], T[mod((j+1), n) ]) ; // sous-diag
         A[j]= f(A[mod((j-1), n) ], T[j], A[mod((j+1), n) ]) ; // diag
       }
      { int j = mod(k,n) ; // int j = mod( orig-(int)K - 1, n) ;
        T[j]= A[j] ;  // sous-diag
        A[j]= f(T[mod((j-1), n) ], A[j], T[mod((j+1), n) ]) ; // sous-diag
}  }  }
```

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

On fait un pavage récursif sur le même principe : on découpe sur la plus grande dimension (n ou m) et on appelle les procédures Losange, Tinf et Tsup. Par exemple, si $n = m$, le carré est pavé comme précédemment avec un losange L de côté $n/2$, 1 triangle T_{inf} et 1 triangle T_{sup}. La découpe récursive est la suivante :

— losange L en 4 demi-losanges ;
— triangle T_{inf} en 3 demi-triangles T_{inf} et 1 demi-triangle T_{sup} ;
— triangle T_{sup} en 3 demi-triangles T_{sup} et 1 demi-triangle T_{inf} ;

Le nombre de défauts de cache est $\Theta\left(\frac{nm}{ZL}\right)$.

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 ?

Cela enlève les dépendances circulaires. Pour la programmation, il faut rajouter des demi-triangles pour les bords gauche et droit.

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 ?

1. $m = 1$, la localité est similaire, mais on peut découper en carré (plus simple qu'en losanges et triangles) : on découpe l'image récursivement en blocs sur la plus grande dimension : on parle de *Z-curve*, cf TP. Analyse du nombre de défauts : on découpe en blocs carrés de taille $K \times K$ tel qu'un bloc et ses 4 frontières (i.e. $4K$ éléments tiennent en cache : 2 des frontières sonts tockées contigues (, chacune sur une ligne i.e. chacune sur $\lceil K/L \rceil + 1$ lignes) et 2 non contigués (chacune stockée sur une colonne donc sur K lignes distinctes) : la place mémoire occupée en nombre d'éléments dans le cahe est donc environ $K^2 + 2K + 2KL$; soit $K \simeq \sqrt{Z}$ tel que le calcul tienne en cache. Le nombre de défauts est

$$Q(n_1, n_2, L, Z) \simeq \frac{n_1 n_2}{K^2}\left(\frac{K^2 + 2K + 2KL}{L}\right) \simeq \frac{n_1 n_2}{L}$$

ce qui est cacghe oblivious.

2. m grand : Le principe est le même mais il faut paver en dimension 3 cette fois, avec des losanges (ie losange 3D ie cube avec rotation de $\pi/4$) et tétraèdres). Le cube étant de volume K^3, on choisit $K = Z^{\frac{1}{3}}$; Le nombre de défauts serait $\Theta\left(\frac{n_1 n_2 m}{K^3}\frac{K^2}{L}\right) = \Theta\left(\frac{n_1 n_2 m}{LZ^{\frac{1}{3}}}\right)$.