Algorithmique et Optimisation Discrète

Séance II – Techniques de blocking et de memoïsation

Équipe pédagogique AOD

Ensimag 2ème année

Technique de blocking pour la localité

- Cas des programmes itératifs
  - Méthodologie : cache aware, cache oblivious
  - Application à des nids de boucles
- Cas des programmes récursifs
  - **Technique de memoization** pour éliminer la redondance
  - puis versions cache aware et cache oblivious

Plan du cours

Technique de blocking : présentation et code

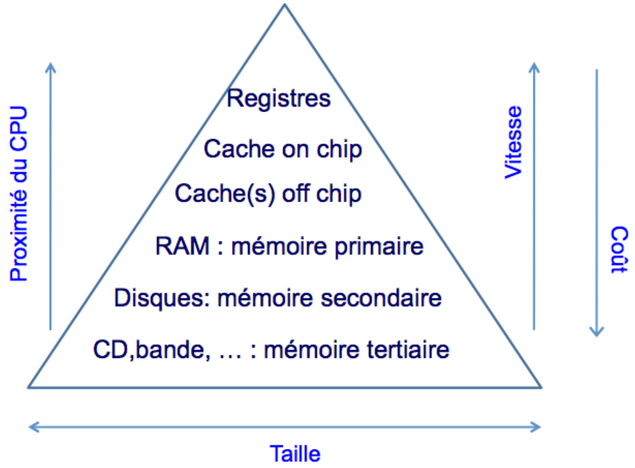
- Rappel modèle CO : cache de taille Z, ligne de taille L, police LRU
- Technique de blocking
- Programmation : exemple de la transposition de matrice

Exemples d'algorithmes cache aware/oblivious

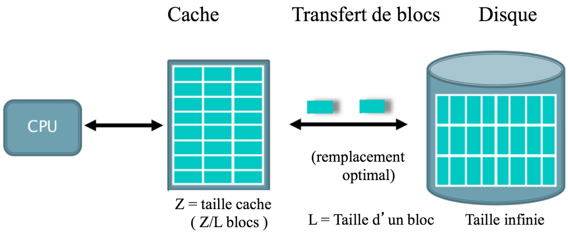
- Produit de matrices
- Méthodolologie pour le blocking
- Exemple : double boucle imbriquée

Conclusion

Hiérarchie mémoire (Rappel)

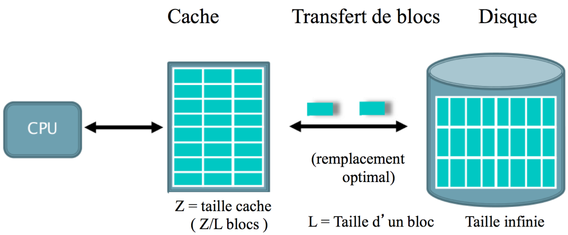


Rappel : modèle théorique CO



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

Rappel : modèle théorique CO



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

Comment améliorer la localité ?

Méthodologie en 3 étapes :

1. **Analyser** la localité de l'algorithme : calcul de $Q(n, L, Z)$
   - en supposant que tout rentre en cache : défauts *obligatoires*
   - en supposant le cache très petit : defauts superflus ?
2. ***Cache aware*** : structurer le calcul en **«blocs» d'opérations** s'exécutant « en cache »
   - manipulation de $\left[\begin{array}{l} Z/L \text{ données non contigües} \\ \text{ou } Z \text{ données contigües sur des lignes de taille } L. \end{array}\right.$
   - $\Longrightarrow$ seulement #défauts $= O(Z/L)$ pour tout le bloc de calcul !
3. ***Cache oblivious*** : rendre Z implicite
   - Découpe récursive en blocs d'opérations de plus en plus petits
   - Dès qu'un bloc manipule $\leq Z$ données, il tient en cache !
   - Attention : rendre négligeable le surcout des appels récursifs en réglant le **seuil d'arrêt** de la récursivité :
     - suffisamment petit pour tenir dans le cache $Z_{\mathbf{1}}$;
     - suffisamment grand pour amortir le surcout des appels récursifs.

Transposition de matrice : blocking cache-aware

```
for(int i = 0 ; i < m ; i++)
   for(int j = 0 ; j < n ; j++)
       B[j*m+i] = A[i*n+j];
```

- Version initiale :
  - Défauts obligatoires (si $Z \gg 2.n.m$) : $Q_1(n, m, L, Z) = 2nm/L$
  - Mais si cache limité (ie $mL > Z$) : $Q_1(n, m, L, Z) = nm + O\left(\frac{nm}{L}\right)$

Transposition de matrice : blocking cache-aware

```
for(int i = 0 ; i < m ; i++)
   for(int j = 0 ; j < n ; j++)
       B[j*m+i] = A[i*n+j];
```

- Version initiale :
  - Défauts obligatoires (si $Z \gg 2.n.m$) : $Q_1(n, m, L, Z) = 2nm/L$
  - Mais si cache limité (ie $mL > Z$) : $Q_1(n, m, L, Z) = nm + O\left(\frac{nm}{L}\right)$
- Cache **aware** : Par blocs de taille $K \simeq \sqrt{Z/2}$ (pour avoir $2K^2 = Z - 0(1)$)

$$\Longrightarrow Q_2(n, m, L, Z) \leq \frac{nm}{K^2} \times (2K \times (K/L + 1)) \simeq 2nm\left(\frac{1}{L} + \frac{1}{\sqrt{Z}}\right) = \Theta\left(\frac{nm}{L}\right)$$

```
void transposeV2 ( double*A, double*B, int m, int n) {
    for (int I=0; I < m; I += K)
      for (int J=0; J < n;  J += K) {
        // Transposition du bloc I, J
        int i_end = min( I+K, m ) ;
        int j_end = min( J+K, n ) ;
        for (int i=I; i < i_end; ++i)
          for (int j=J; j < j_end; ++j)
             B[j*m+i] = A[i*n+j];
}      }
```

Transposition de matrice : blocking cache-aware

Plus généralement, quelle taille de bloc *rectangulaire* $R_I \times R_J$ choisir ?

- Contrainte : les 2 blocs tiennent en cache, ie $R_I \times R_J + R_J - \times R_I < Z$
- nombre de défauts : $Q(m, n, L, Z, R_I, R_J) = \frac{m}{R_I}\frac{n}{R_J}$ $\left(R_I\left(\frac{R_J}{L}+1\right)+R_J\left(\frac{R_I}{L}+1\right)\right) = \frac{nm}{L}\ \left(2+\frac{L}{R_J}+\frac{L}{R_I}\right)$
- À produit $R_I \times R_J = Z$ constant, la somme $\frac{1}{R_I} + \frac{1}{R_J}$ est minimale pour $R_I = R_J$
- D'où le choix $R_I = R_J = \sqrt{Z/2} \hookrightarrow$ le nombre de défauts associé est $Q = \frac{nm}{L}\left(2 + O\left(\frac{L}{\sqrt{Z}}\right)\right)$
- Si $Z = \Omega\left(L^2\right)$, on a $Q = \Theta\left(\frac{nm}{L}\right)$ qui est asymptotiquement optimal.
- Si $Z = \omega\left(L^2\right)$, on a $Q = \frac{nm}{L}(2 + o(1))$ qui est proche du nombre de défauts obligatoires $2\frac{nm}{L}$.

Transposition de matrice : blocking cache-oblivious

```
void transposeRec( double* A, int mAb, int mAe, int nAb, int nAe,
                   double* B, int m, int n) {
    int nblines = mAe - mAb; int   nbcols = nAe - nAb;
    if (( nblines <= S ) && (nbcols <= S)) {
      int iA, jA ;
      for ( iA=mAb; iA < mAe; ++iA )
        for ( jA=nAb; jA < nAe; ++jA )
             // B[jB*m+iB] = A[iA*n+jA];
             B[jA*m+iA] = A[iA*n+jA];
    }
    else if ( nblines > nbcols ) {
        int mid = nblines / 2 ;
        transposeRec( A, mAb + mid, mAe, nAb, nAe, B, m, n);
        transposeRec( A, mAb, mAb + mid, nAb, nAe, B, m, n);
    }
    else {
        int mid = nbcols / 2 ;
        transposeRec( A, mAb, mAe, nAb + mid, nAe, B, m, n);
        transposeRec( A, mAb, mAe, nAb, nAb + mid, B, m, n);
    }
}
```

Transposition de matrice : blocking cache-oblivious

```c
void transposeRec( double* A, int mAb, int mAe, int nAb, int nAe,
                   double* B, int m, int n) {
    int nblines = mAe - mAb; int   nbcols = nAe - nAb;
    if (( nblines <= S ) && (nbcols <= S)) {
      int iA, jA ;
      for ( iA=mAb; iA < mAe; ++iA )
        for ( jA=nAb; jA < nAe; ++jA )
             // B[jB*m+iB] = A[iA*n+jA];
             B[jA*m+iA] = A[iA*n+jA];
    }
    else if ( nblines > nbcols ) {
        int mid = nblines / 2 ;
        transposeRec( A, mAb + mid, mAe, nAb, nAe, B, m, n);
```

Utilisation d'un seuil (S) pour amortir le coût de la découpe récursive.

cache oblivious : $Q_3(n, m, L, Z) = 2\frac{nm}{L} + O\left(\frac{nm}{\sqrt{Z}}\right)$

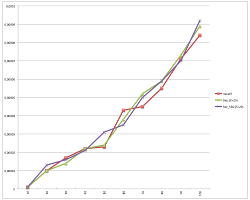
```c
        transposeRec( A, mAb, mAe, nAb, nAb + mid, B, m, n);
    }
}
```

Mesures de temps d'exécution

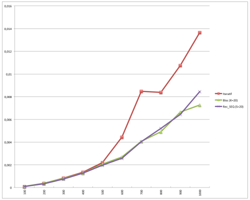
Comparaison : Itératif versus **Bloc** $(K = 20)$ versus **Récursif** $(S = 20)$

Machine : Mac OSX 10.9.5, 2.7GHz Intel Core i7, Mémoire 16Go 1600 MHz DDR3.

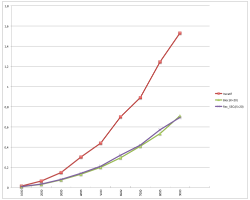
Compilateur gcc 4.7.2 -O4



10 à 100



100 à 1000

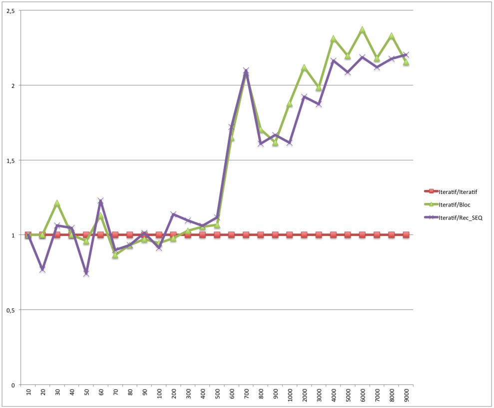


1000 à 10000

Cache-aware et cache-oblivious sont les plus performants

Mesures de temps : rapports

Temps Itératif / temps X ; avec $X \in \{$ Itératif, Bloc($K = 20$), Récursif ($S = 20$)$\}$
Tailles de 10 à 100, de 100 à 1000 et de 1000 à 10000



Transposition de matrice : récursive parallèle

Les 2 appels récusifs peuvent être faits en parallèle !
↪ Par exemple, en **OpenMP** [`gcc -fopenmp`], avec des **task**

Remarque : Ici ordonnancement glouton par défaut : lorsqu'un thread est inactif, il vole une tâche non encore traitée à un autre thread (performances prouvées pour les programmes très parallèles)
Si synchronisation (`#pragma omp wait`), utiliser annotation *untied*.

Transposition de matrice : récursive parallèle

```c
void transposeRecPAR( double* A, ...,  double* B, ... ) {
    int nblines = mAe - mAb;  int  nbcols = nAe - nAb;
    if (( nblines <= SPAR ) && (nbcols <= SPAR))
      transposeRec( A, mAb, mAe, nAb, nAe, B, m, n) ;
    else if ( nblines > nbcols ) {
        int mid = nblines / 2 ;
        #pragma omp task // pour faire cet appel récursif en parallèle
        { transposeRecPar( A, mAb + mid, mAe, nAb, nAe, B, m, n ) ; }
        transposeRecPar( A, mAb, mAb + mid, nAb, nAe, B, m, n ) ;
    }
    else {
        int mid = nbcols / 2 ;
        #pragma omp task // pour faire cet appel récursif en parallèle
        { transposeRecPar( A, mAb, mAe, nAb + mid, nAe, B, m, n ) ; }
        transposeRecPar( A, mAb, mAe, nAb, nAb + mid, B, m, n ) ;
    }
}
```

Transposition de matrice : récursive parallèle

...et pour l'appel principal

```
void transposePAR ( double*A, double*B, int m, int n) {
  if (( m <= S ) && (n <= S) { transposeIteratif( A, B, m, n ) ; }
  else if ( m+n <= SPAR)
    transposeSEQ(A, B, m, n) ;
  else {
    #pragma omp parallel { // Debut section parallele sur tous les
        threads
      #pragma omp single nowait // Un seul thread lance le calcul
      {  transposeRecPAR(A, 0, m, 0, n, B, n, m) ; }
} } }
```

Analyse de la *profondeur*=longueur du chemin critique :

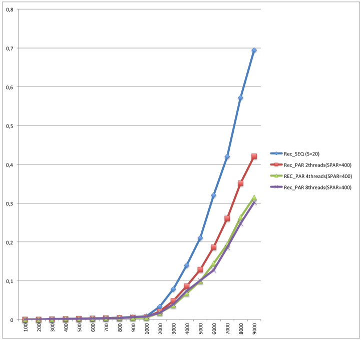
$$D(n,m) = \begin{cases} O(nm)) & \text{si } n,m \leq \text{SPAR} \\ D(n/2,m) + O(1) & \text{sinon si } n \geq m \\ D(n,m/2) + O(1) & \text{sinon} \end{cases} \implies D(n,m) = \Theta\left(\log \frac{nm}{\text{SPAR}^2}\right)$$

Performances séquentiel versus parallèle

Comparaison réc. séquentiel ($S = 20$) et par. (SPAR=400) sur 2, 4 et 8 threads.

Machine : Mac OSX 10.9.5, 2.7GHz Intel Core i7, Mémoire 16Go 1600 MHz DDR3.

Compilateur gcc -fopenmp 4.7.2 -O4



Temps mesurés

Rapport tps réc. séq. / tps réc. par.

Plan du cours

Technique de blocking : présentation et code

- Rappel modèle CO : cache de taille Z, ligne de taille L, police LRU
- Technique de blocking
- Programmation : exemple de la transposition de matrice

Exemples d'algorithmes cache aware/oblivious

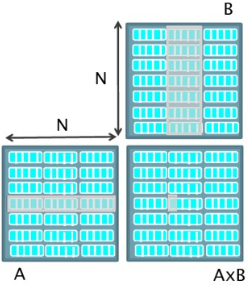
- Produit de matrices
- Méthodolologie pour le blocking
- Exemple : double boucle imbriquée

Conclusion

Produit de matrices - i,j,k

Algorithme (i, j, k) **:**

```
for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
     for (int k = 0; k < n; ++k)
       C(i,j) += A(i,k) * B(k,j)
```



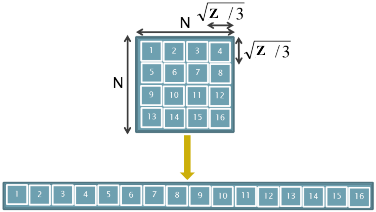
- Travail : $W_1(n) = n^3$ MultiplyAdd $= \Theta(n^3)$
- Défauts de cache : cas extrêmes :
  - si tout tient en cache, i.e. $3n^2 < Z$ alors $Q(n, L, Z) = \left(\frac{3n^2}{L}\right)$;
  - **cas général** : cache petit, une colonne ne tient pas en cache, i.e. $n > Z/L$

$$\mathrm{Q}_1(\mathrm{n}, \mathrm{L}, \mathrm{Z}) = n^2 \times \left(\frac{n}{L} + n\right) + \frac{n^2}{L} \simeq \mathrm{n}^3.$$

Produit de matrices : blocking cache-aware

Le calul du produit par blocs $R \times R$ s'écrit :

```
for (int I = 0; I < n; I+=R) {
  int imax = min (I+B, n) ;
  for (int J = 0; J < n; J+=B ) {
   int jmax = min (J+B, n) ;
   for (int K = 0; K < n; K+=B) {
    int kmax = min (K+B, n) ;
    for (int i = I; i < imax; ++i)
     for (int k = K; k < kmax; ++k)
      for (int j = J; j < jmax; ++j)
       C(i,j) += A(i,k) * B(k,j)
} } }
```

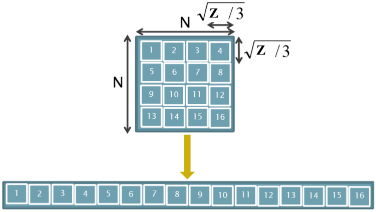


Taille de bloc $R \times R$

Produit de matrices : blocking cache-aware

Le calul du produit par blocs $R \times R$ s'écrit :

```
for (int I = 0; I < n; I+=R) {
  int imax = min (I+B, n) ;
  for (int J = 0; J < n; J+=B ) {
   int jmax = min (J+B, n) ;
   for (int K = 0; K < n; K+=B) {
    int kmax = min (K+B, n) ;
    for (int i = I; i < imax; ++i)
     for (int k = K; k < kmax; ++k)
      for (int j = J; j < jmax; ++j)
       C(i,j) += A(i,k) ⋆ B(k,j)
} } }
```



Taille de bloc $R \times R$

Comment choisir R ? (*avec contrainte de tenir en cache, i.e.* $3R^2 \leq Z$)

- Travail : $W_2(n) = n^3$ MultiplyAdd $+ \ldots = W_1(n) + \Theta\left(\frac{n^3}{R^3}\right)$
- Défauts de cache : $Q_2(n, L, Z) = \frac{n^3}{R^3} \times \frac{3R^2}{L} = 3\left(\frac{n^3}{LR}\right)$.
- Choisir R maximal minimise Q_2 et W_2 : d'où $R \simeq \sqrt{Z/3}$ et $Q_2 = \Theta\left(\frac{n^3}{L\sqrt{Z}}\right)$.

Complément : taille des blocs minimisant les défauts ?

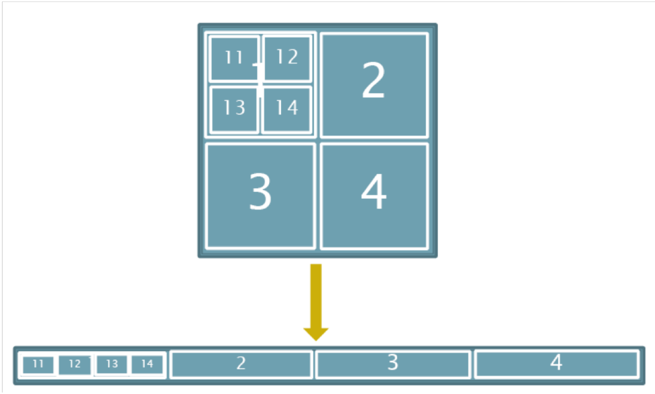
- Cas général : blocs de taille $R_I \times R_J$ sur C ; $R_I \times R_K$ sur A ; $R_K \times R_J$ sur B tels que ces 3 blocs tiennent en cache simultanément
- $\#\text{défauts} = \frac{n}{R_I}\frac{n}{R_J}\frac{n}{R_K}\left(R_I\frac{R_J}{L} + R_I\frac{R_K}{L} + R_K\frac{R_J}{L}\right) = \frac{n^3}{L}\left(\frac{1}{R_I} + \frac{1}{R_J} + \frac{1}{R_K}\right)$

$\Longrightarrow$ on choisit R_I, R_J, R_K pour minimiser $\frac{1}{R_I} + \frac{1}{R_J} + \frac{1}{R_K}$ avec $(R_I \times R_J + R_I \times R_K + R_K \times R_J \leq Z)$

- d'où $R_I = R_J = R_K \simeq \sqrt{Z/3}$.

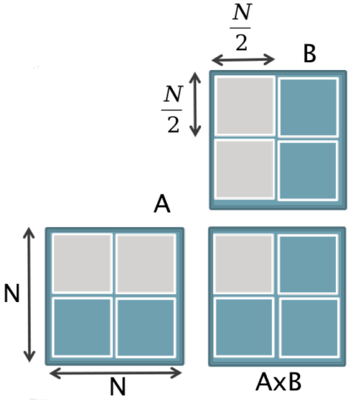
Produit de matrices - Cache oblivious

Découpe récursive jusqu'à un seuil S d'arrêt de la récursivité :



Produit de matrices - Cache oblivious

Découpe récursive jusqu'à seuil S



- $W_3(n) =$
  $n^3 \; + \text{surcoût récursivité} =$
  $W_1(n) + \Theta\left(\frac{n^3}{S^3}\right)$.
- $Q_3(n, L, Z) =$
  $$\begin{cases} 3n^2/L & \text{si } 3n^2 < Z \\ 8Q_3(n/2) + O\left(\frac{n^2}{L}\right) & \text{sinon} \end{cases}$$

$$\Longrightarrow Q_2(n, L, Z) = \Theta\left(\frac{n^3}{L\sqrt{Z}}\right).$$

Blocking : Méthodolologie générale

Les 3 étapes pour grouper les instructions afin d'améliorer la localité :

1. **Analyser les dépendances** "écriture-lecture" de données entre instructions
   - programme séquentiel : ' ;' définit un ordre total des instructions
   - mais sémantique *write before read* $\hookrightarrow$ **ordre partiel**
2. **Blocking cache-aware** : regrouper les instructions en blocs itératifs
   - en respectant l'ordre partiel des dépendances
   - un bloc d'instructions doit d'exécuter "en place" dans un cache de taille Z
3. **Blocking cache-oblivious** : découpe récursive
   - pour obtenir implicitement les blocs de taille Z du blocking cache-aware

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)   f( V[i], V[j] );    // V : tableau
```

- avec $f(a, b)$ accède a et b en R-W (sans autre effet de bord)

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)   f( V[i], V[j] );    // V : tableau
```

- avec $f(a, b)$ accède a et b en R-W (sans autre effet de bord)

- **Remarque :** schéma d'itération classique

  Tri par insertion $\hookrightarrow$ `#define f(a,b) if (a<b) swap(a,b)`

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)   f( V[i], V[j] );     // V : tableau
```

- avec $f(a, b)$ accède a et b en R-W (sans autre effet de bord)

- **Analyse défauts de cache :**

  si $n \ll Z$: $Q \simeq \frac{n}{L}$

  si $n \gg Z$: $Q \simeq \frac{n^2 - Z^2 + Z}{L} \simeq \frac{n^2}{L}$

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)  f( V[i], V[j] );      // V : tableau
```

- avec $f(a, b)$ accède a et b en R-W (sans autre effet de bord)
- **Etape 1 : analyse des dépendances** entre instructions

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)  f( V[i], V[j] );      // V : tableau
```

- avec $f(a,b)$ accède a et b en R-W (sans autre effet de bord)
- **Etape 1 : analyse des dépendances** entre instructions
  - *à l'étape i* : $V[i]$ dépend de $V[0], V[1], \ldots V[i-1]$ (dans l'ordre).

Exemple de blocking : double boucle imbriquée

```c
for (i=1 ; i<n; ++i)
  for (j=0; j<i; ++j)  f( V[i], V[j] );      // V : tableau
```

- avec $f(a,b)$ accède a et b en R-W (sans autre effet de bord)
- **Etape 1 : analyse des dépendances** entre instructions
  - *à l'étape* i : $V[i]$ dépend de $V[0], V[1], \ldots V[i-1]$ (dans l'ordre).
  - Graphe de dépendances pour $n = 5 \Longrightarrow$ parallélisme !

$$
\begin{array}{ccccccccc}
f(1,0) & & & & & & & & \\
\downarrow & & & & & & & & \\
f(2,0) & \rightarrow & f(2,1) & & & & & & \\
\downarrow & & \downarrow & & & & & & \\
f(3,0) & \rightarrow & f(3,1) & \rightarrow & f(3,2) & & & & \\
\downarrow & & \downarrow & & \downarrow & & & & \\
f(4,0) & \rightarrow & f(4,1) & \rightarrow & f(4,2) & \rightarrow & f(4,3) & & \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \\
f(5,0) & \rightarrow & f(5,1) & \rightarrow & f(5,2) & \rightarrow & f(5,3) & \rightarrow & f(5,4)
\end{array}
$$

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
  for (j=0; j<i; ++j)  f( V[i], V[j] );      // V : tableau
```

Programmation explicitant le parallélisme :

- Pour $0 \leq t \leq 2n-3$, tous les appels (i,j) avec $i+j=t$ et $0 \leq j < i$ peuvent être calculés en parallèle.
- programmation boucle externe "par diagonale" $\Longrightarrow$ boucle interne parallèle

```
for (int t=1, tmax = 2*n-3; t<= tmax ; ++t) {
   int imax = min(t+1, n) ;
   parallel for (int i=1; i < imax; ++i ) {
      int j=t-i ;  if ( j < i)  f( V[i], V[j] );
}  }
```

- *Remarque :* exécution *out of order*

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
  for (j=0; j<i; ++j)   f( V[i], V[j] );    // V : tableau
```

- **Etape 2 : blocking cache aware** $\hookrightarrow$ blocs de taille K tq $2K < Z$

```
for (int I=0; I<n; I++K )
  for (int J=0; J<=I; J=J+K) {
    int iMax=min(I+K,n);
    for (int i=I ; i< iMax; ++i) {
      int jMax = min( J+K, i) ;
      for (int j=J; j<jMax; ++j) f( V[i], V[j] );
  } }
```

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)   f( V[i], V[j] );    // V : tableau
```

- **Etape 2 : blocking cache aware** $\hookrightarrow$ blocs de taille K tq $2K < Z$

```
for (int I=0; I<n; I++K )
   for (int J=0; J<=I; J=J+K) {
      int iMax=min(I+K,n);
      for (int i=I ; i< iMax; ++i) {
         int jMax = min( J+K, i) ;
         for (int j=J; j<jMax; ++j) f( V[i], V[j] );
   } }
```

- $$Q(n, L, Z) = \sum_{I=0}^{n/K} \sum_{J=0}^{I} 2\frac{K}{L} = \Theta\left(\frac{n^2}{ZL}\right).$$

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)    f( V[i], V[j] );     // V : tableau
```

- **Etape 3 : blocking cache oblivious** $\hookrightarrow$ découpe du plus grand bloc en 2

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)   f( V[i], V[j] );     // V : tableau
```

- **Etape 3 : blocking cache oblivious** $\hookrightarrow$ découpe du plus grand bloc en 2
- **cas 1** : $[j_{begin}, j_{end}[< [i_{begin}, i_{end}[$ $\hookrightarrow$ 2 appels
  - si $(i_{end} - i_{begin}) > (j_{end} - j_{begin})$ $\hookrightarrow$ découpe $[i_{begin}, i_{end}[$ en 2
  - si $(i_{end} - i_{begin}) \leq (j_{end} - j_{begin})$ $\hookrightarrow$ découpe $[j_{begin}, j_{end}[$ en 2

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)   f( V[i], V[j] );     // V : tableau
```

- **Etape 3 : blocking cache oblivious** $\hookrightarrow$ découpe du plus grand bloc en 2
- **cas 1** : $[j_{begin}, j_{end}[<[i_{begin}, i_{end}[$ $\hookrightarrow$ 2 appels
  - si $(i_{end} - i_{begin}) > (j_{end} - j_{begin})$ $\hookrightarrow$ découpe $[i_{begin}, i_{end}[$ en 2
  - si $(i_{end} - i_{begin}) \leq (j_{end} - j_{begin})$ $\hookrightarrow$ découpe $[j_{begin}, j_{end}[$ en 2
- **cas 2** : $[j_{begin}, j_{end}[=[i_{begin}, i_{end}[$ $\hookrightarrow$ 3 appels

Exemple de blocking : double boucle imbriquée

```
for (i=1 ; i<n; ++i)
   for (j=0; j<i; ++j)   f( V[i], V[j] );    // V : tableau
```

- **Etape 3 : blocking cache oblivious** $\hookrightarrow$ découpe du plus grand bloc en 2
- **cas 1** : $[j_{begin}, j_{end}[< [i_{begin}, i_{end}[\hookrightarrow$ 2 appels
  - si $(i_{end} - i_{begin}) > (j_{end} - j_{begin}) \hookrightarrow$ découpe $[i_{begin}, i_{end}[$ en 2
  - si $(i_{end} - i_{begin}) \leq (j_{end} - j_{begin}) \hookrightarrow$ découpe $[j_{begin}, j_{end}[$ en 2
- **cas 2** : $[j_{begin}, j_{end}[= [i_{begin}, i_{end}[\hookrightarrow$ 3 appels
-

$$Q(n_1, n_2, L, Z) \leq \begin{cases} \frac{n_1+n_2}{L} + O(1) & \text{si } n_1 + n_2 < Z \\ 2Q(n_1/2, n_2, L, Z) + O(1) & \text{sinon si } n_1 > n_2 \\ 2Q(n_1, n/2, L, Z) + O(1) & \text{sinon} \end{cases}$$

D'où $Q(n, L, Z) = \Theta\left(\frac{n}{Z}\left(\frac{n}{L} + O(1)\right)\right) \simeq \frac{n^2}{ZL}$.

Exemple de blocking : double boucle imbriquée

```
void calculRec( Tab V, int bI, int eI, int bJ, int eJ ) {
  if ( (eI - bI <  S)  and  (eJ - bJ < S) ) { // Seuil
    for (int i=bI; i<=eI; ++i ) {
        int jMax = min(i, eJ) ;
        for (int j=bJ; j < jMax; ++j)  f( V[i], V[j] );
  } }
  else { // découpe du plus grand en 2
    if ( (bI == bJ) and (eI == eJ) ) {
         int middle = (eI + bI) / 2 ;
         calculRec( V, bI, middle, bJ, middle ) ;
         calculRec( V, middle, eI, bJ, middle ) ;
         calculRec( V, middle, eI, middle, eJ ) ;
    } elsif ((eI-bI) > (eJ-bJ)) {
         int middle = (eI + bI) / 2 ;
         calculRec( V, bI, middle, bJ, eJ ) ;
         calculRec( V, middle, eI, bJ, eJ ) ;
    } else  {
         int middle = (eJ + bJ) / 2 ;
         calculRec( V, bI, eI, bJ, middle ) ;
         calculRec( V, bI, eI, middle, bJ ) ;
} } }
```

Plan du cours

Technique de blocking : présentation et code

- Rappel modèle CO : cache de taille Z, ligne de taille L, police LRU
- Technique de blocking
- Programmation : exemple de la transposition de matrice

Exemples d'algorithmes cache aware/oblivious

- Produit de matrices
- Méthodolologie pour le blocking
- Exemple : double boucle imbriquée

Conclusion

Ce qu'on a vu aujourd'hui

Améliorer la localité par technique de blocking (cache oblivious)

$\Longrightarrow$ programmation en *cascade* récursif/itératif

1. A partir de l'algorithme initial :
   - Analyser les défauts $Q(n, L, Z)$ sur modèle CO
   - Analyser les dépendances lecture-écriture entre instructions :
     respecter l'ordre partiel des (macro-)instructions (graphe macro-dataflow)
2. Cache aware : regrouper les instructions en "blocs" tel que chaque bloc s'exécute dans un cache de taille Z

   $\hookrightarrow$ le nouvel ordonnancement doit respecter l'ordre partiel des dépendances
3. Cache oblivious : construire une découpe récursive de l'ensemble des instructions qui approche l'ordonnancement cache aware sans utiliser la valeur de Z

   $\hookrightarrow$ adapter le seuil d'arrêt de la récursivité pour amortir le surcout

A suivre : : application à des schémas récursifs (programmation dynamique)