Programmation parallèle (PPAR)
Cours 3 : modèles

P. Fortin
pierre.fortin @ upmc.fr

D'après le cours de J.-L. Lamotte

Master 2 Informatique - UPMC

Granularité - degré de parallélisation

- La **granularité** correspond à la taille des tâches effectuées en parallèle.
  Il existe trois types de granularité : fine, moyenne, grosse.
  - gros grain de calcul,
  - grain de calcul moyen,
  - gain de calcul fin.
- Le **degré de parallélisation** correspond aux nombres d'opérations que l'on peut effectuer en parallèle.
  - Plus le degré est important, plus on peut accélérer le calcul.
  - ATTENTION : le degré n'est pas constant au cours du déroulement d'un programme.

Exemple

Le produit de deux matrices de taille N : $C = A * B$

```
for i=1 to N
  for j=1 to N
     S=0
     for k=1 to N
        S=S+A(i,k)*B(k,j)
     C(i,j)=S
```

- Dans la multiplication de matrice, on peut paralléliser :
  - la boucle `for k=1 to N` $\longrightarrow$ granularité fine ;
  - la boucle `for j=1 to N` $\longrightarrow$ granularité moyenne ;
  - la boucle `for i=1 to N` $\longrightarrow$ granularité grosse.
- Le choix de la méthode est fait en fonction de l'architecture de la machine et des performances des communications.

Le parallélisme de données

- Il s'applique généralement à des données régulières.
- Divisions régulières des données et envoi aux processeurs effectuant des calculs identiques sur des données différentes.
- Le parallélisme de données s'implémente naturellement sur les machines MIMD-SM.

Un exemple de parallélisme de données

```
pour i=1,n
  parbegin
    a(i)=f(i) // calcul f(i) independant du tableau a
  parend
finpour
pour i=1,n
  parbegin
    b(i)=a(i)+a(n-i+1)
  parend
finpour
```


Exemple (suite)

- Chacune des deux boucles peut être effectuée en parallèle mais la 1^{iere} doit être terminée avant le début de la 2^e.
- La deuxième boucle pose le problème de l'accès à `a(i)` et `a(n-i+1)`.
- Ex : pour `i=1`, `b(1)` est calculé à partir de `a(1)` et `a(n)`.
- Sur une machine SM, la programmation ne pose pas de problème.
- L'écriture du programme correspond à un programme pour une machine SM. Le compilateur gère la répartition des données.

Exemple (suite)

- Pour les machines DM, la première boucle se parallélise bien. Les données sont partagées et envoyées à chaque processeur.
- Avec par exemple une distribution par bloc, le code devient avec p processeurs numéroté de 0 à $p-1$, p diviseur de n :

```
// reception des donnees
....
TailleBloc = n/p
pour i=1,TailleBloc
   a(i)=f(NumProc*TailleBloc+i)
finpour
```


Exemple (suite)

- Par contre, la 2^e boucle pose des problèmes dues au calcul de `a(i)+a(n-i+1)`.
- Solutions :
  1. récupération par une communication de `a(n-i+1)`, bilan :

     un calcul = une communication ;
  2. le processeur k reçoit (avant le calcul) la partie de `a` du processeur $p-k-1$ et lui envoie sa partie de `a`. Modification importante du programme.

Distribution des données : cas 1D

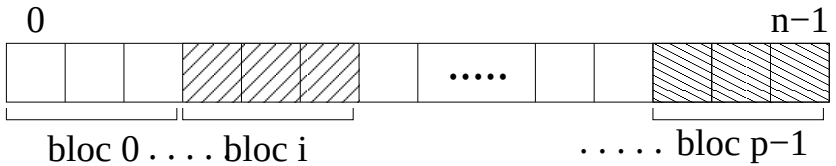
- La régularité des données permet de les distribuer facilement.

  Exemple : tableau $n \times n$ partagé sur p processeurs, chacun recevant n/p lignes.
- Les schémas de distribution les plus courant sont :
  - distribution par bloc,
  - distribution cyclique (modulo),
  - distribution par bloc cyclique.
- Soit $t(n)$, un tableau de taille n et p le nombre de processeurs (p diviseur de n).

  Chaque processeur $(P_i)_{0 \leq i \leq p-1}$ reçoit des données $\{d_j\}_{0 \leq j < n}$

Distribution des données : cas 1D (suite)

- par bloc :



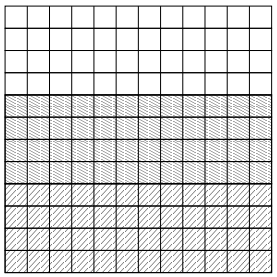
  Le processeur i possède les données $i*(n/p) \leq j < (i+1)*n/p$
- cyclique :

  Le processeur i possède les données $j \ mod \ p = i$
- par bloc cyclique :

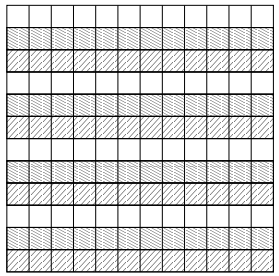
  Le processeur i possède les données $\lfloor j/b \rfloor \ mod \ p = i$.
  Répartition par bloc de taille b, b diviseur de $\frac{n}{p}$.

Distribution 1D de données 2D

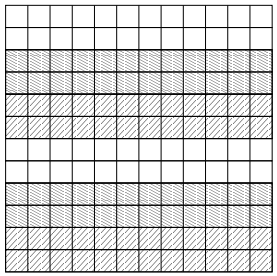
Processeurs $(P_i)_{0 \leq i \leq p-1}$



bloc



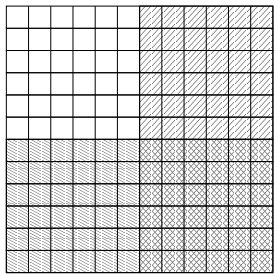
cyclique



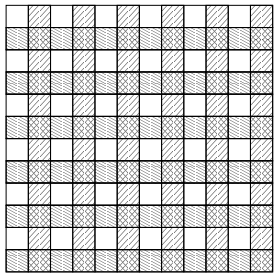
bloc cyclique

Distribution 2D de données 2D

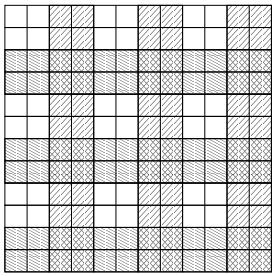
Processeurs $(P_{i,j})_{0 \leq i \leq q-1, 0 \leq i \leq q-1}$ (avec $q^2 = p$)



bloc



cyclique



bloc cyclique

Dépendance de données

- Certaines parties de programme posent des problèmes difficiles à résoudre :
- Calcul d'une somme partielle :

```
sp(1) = a(1)
pour i=2,n
  sp(i)=a(i)+sp(i-1)
```

- On est en présence d'une **dépendance de données**.
- Faire le calcul en parallèle en utilisant le `parbegin` et le `parend` revient à effectuer un AUTRE CALCUL.

Les conditions de Bernstein

- C'est une condition nécessaire et suffisante d'indépendance d'instructions basée sur les dépendances entre données.
- Soit S une séquence de calcul (d'instructions).
- Soit $L(S)$ et $E(S)$ l'ensemble des données utilisées respectivement en lecture et en écriture durant l'exécution de S.
- Deux instructions (ou séquences de calcul) S_1 et S_2 (S_1 précédant S_2 en séquentiel) sont indépendantes et peuvent être exécutées en parallèle SANS MODIFIER LE RÉSULTAT si on a les 3 conditions suivantes :
  a) $E(S_1) \cap L(S_2) = \emptyset$ (*dépendance de flot*)
  b) $L(S_1) \cap E(S_2) = \emptyset$ (*anti-dépendance*)
  c) $E(S_1) \cap E(S_2) = \emptyset$ (*dépendance de sortie*)

Condition de Bernstein : exemple

```
   |      a(0) = 0
   |      sp(0) = a(0)
   |      pour i=1,n-1
S1 |         a(i)=2*i
S2 |         sp(i)=a(i)+sp(i-1)
```

- Au $\ll j^{eme} \gg$tour de boucle, on a
  $E(S1(j)) \cap L(S2(j)) = \{\texttt{a(j)}\} \neq \emptyset$
  $\Longrightarrow$ Pas de parallélisation possible des instructions de la boucle.
- on a $E(S2(j)) \cap L(S2(j+1)) = \{\texttt{sp(j)}\} \neq \emptyset$
  $\Longrightarrow$ Pas de parallélisation possible de la boucle.

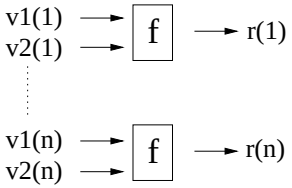
Condition de Bernstein : exemple

- Par contre on peut modifier le code pour le rendre partiellement parallèle :

```
a(0) = 0
sp(0) = a(0)
pour i=1,n-1
   parbegin
     a(i)=2*i
   parend
pour i=1,n-1
   begin
      sp(i)=a(i)+sp(i-1)
   end
```

Profils classiques de fonctions parallèles : α-schéma

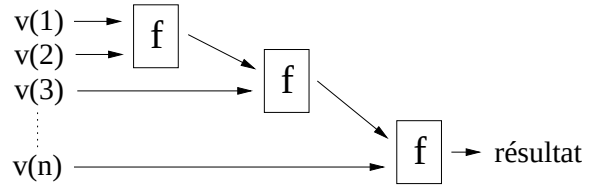
- Soit *f* une fonction de $\mathbb{R}^s \longrightarrow \mathbb{R}$.
- *f* s'applique à *s* vecteurs de taille *n*, et permet d'obtenir un vecteur résultat.
- Exemple : $\vec{r} = +(\vec{v_1}, \vec{v_2})$
- Ce type de fonction est facile à paralléliser.



Seul problème : savoir si *f* est constant en temps de calcul.

Profils classiques de fonctions parallèles : β-schéma

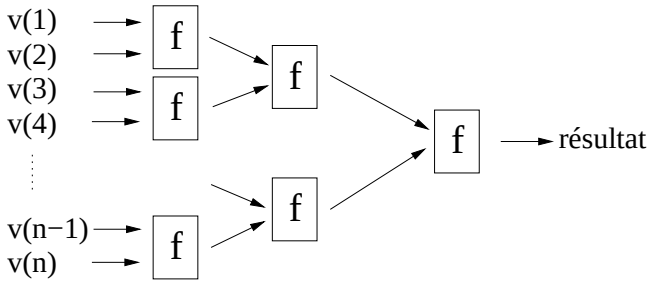
- Soit *f* une fonction de $\mathbb{R}^2 \longrightarrow \mathbb{R}$ et un vecteur unique $\vec{v}$.
- Le schéma correspond à l'application de *f* sur les deux premiers éléments du vecteur, puis ensuite *f* est appliquée au résultat précédent et à l'élément suivant de $\vec{v}$.
- Exemple : somme des éléments de $\vec{v}$



- C'est un β-schéma.
- Rédigé sous cette forme, ce calcul est non parallélisable.

Profils classiques de fonctions parallèles : β-schéma logarithmique

- Si *f* est une fonction associative alors on peut reécrire le β-schéma sous la forme :



Seul problème : savoir si *f* est constant en temps de calcul.

Profils classiques de fonctions parallèle : conclusion

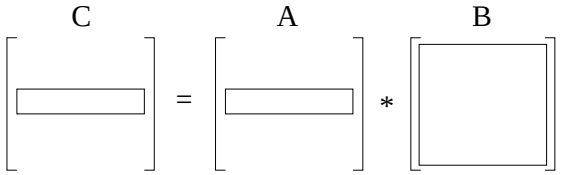
- Ces trois schémas sont assez généraux pour s'appliquer à un grand nombre d'opérations régulières sur des données régulières.
- Il faut étudier le comportement de la fonction pour connaître les temps de calcul en fonction des entrées. Cela influence le mode de répartition des données (→ cf. ≪ équilibrage de charge ≫plus loin).

Etude de cas : la multiplication de 2 matrices

- Sur machine SM $\longrightarrow$ parallélisation immédiate (attention au goulet d'étranglement pour l'accès à la mémoire).
- Architecture de la machine : MIMD-DM.
- Soit $C = A * B$, A,B,C sont des matrices carrées de n^2 éléments
- Soit $p = q^2$ le nombre de processeurs. q et n sont choisis tel que $modulo(n, p) = 0$ (et donc $modulo(n, q) = 0$)
- Nombre d'opérations en virgule flottante : $\approx 2n^3$.
- Comment répartir les données ?
- Les $C_{i,j}$ sont indépendants $\Longrightarrow$ n'importe quel processeur peut calculer les $C_{i,j}$.

Une répartition simple

- Chaque processeur reçoit n/p lignes de A et la matrice B entière.



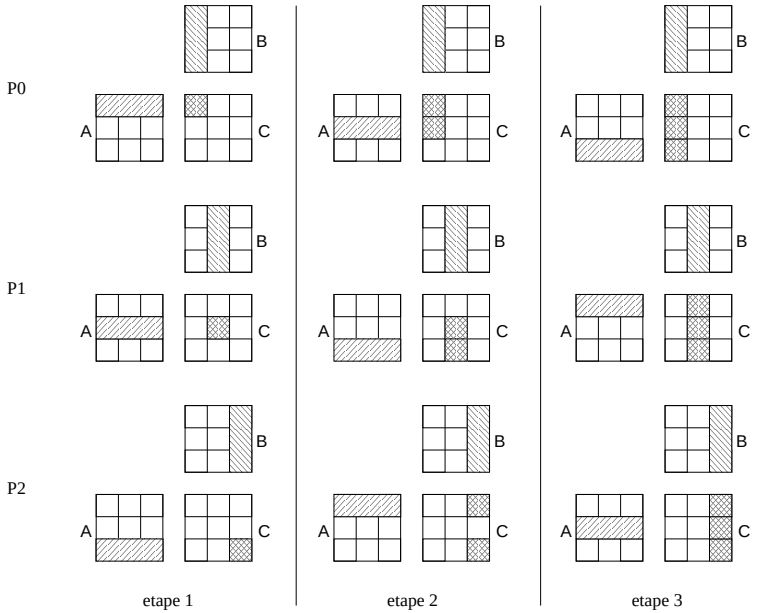
- Le processeur possède toutes les informations pour faire le calcul.
  - $2n^3/p$ opérations,
  - grosses communications en faible nombre avant le calcul,
  - nécessité d'avoir une mémoire locale très importante.
- Si ensuite on a $E = D * C$, il faut beaucoup de messages pour redistribuer D et C.

Pas de prise en compte de l'architecture du réseau.

Implémentation sur un anneau

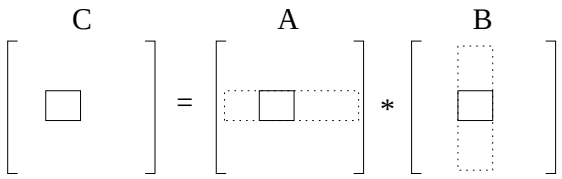
- Une autre idée : chaque processeur possède les mêmes blocs de colonnes pour B et C. B est distribuée sur l'ensemble des processeurs.
- Pour le calcul d'une colonne de C, un processeur aura besoin d'avoir toutes les lignes de A.

Multiplication de matrices sur un anneau



Produit matriciel sur un tore

- Soit $p = q^2$ processeurs numérotés $(P_{i,j})_{i,j \in [0..q-1]}$. Les processeurs sont organisés sur un tore 2D.
- Schéma de la répartition des données :

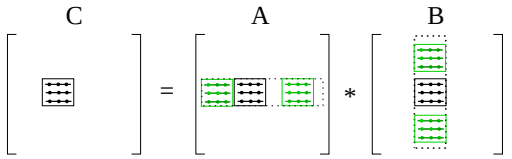


- Chaque processeur $P_{i,j}$ possède $\frac{n^2}{p}$ éléments et effectue $\frac{2n^3}{p}$ opérations.
- Il a besoin de certains blocs ($\sqrt{p}$) de A et certains blocs ($\sqrt{p}$) de B.
- Circulation des blocs de A et de B sur les 2 dimensions périodiques du tore 2D.

La circulation des blocs dans le programme

Pour le processeur $P_{i,j}$ (qui possède le bloc $C_{i,j}$) :

```
pour li=0 a (n/sqrt(p))-1
   pour co=0 a (n/sqrt(p))-1
      | c([bloc:i,j],li,co]) = 0
      | pour e=0 a sqrt(p)-1
      |     | s=0
      |     | pour d=0 a n/sqrt(p)-1
      |     |     | s = s + A([bloc:i,e],li,d)*
      |     |     |         B([bloc:e,j],d,co)
      |     | c([bloc:i,j],li,co) += s
```



Accès multiples aux mêmes blocs pour calculer le bloc de C :

- plusieurs accès aux mêmes blocs de A et de B,
- blocs accédés en même temps par différents processeurs.

Comment mieux organiser la circulation des données ?

En inversant l'ordre des boucles ?

```
Initialisation du bloc C à 0
pour e=0 a sqrt(p)-1
   pour li=0 a (n/sqrt(p))-1
      | pour co=0 a (n/sqrt(p))-1
      |     | s=0
      |     | pour d=0 a n/sqrt(p)-1
      |     |     | s = s + A([bloc:i,e],li,d)*
      |     |     |         B([bloc:e,j],d,co)
      |     | c([bloc:i,j],li,co) += s
```

- 1 seul accès à chaque bloc de A et de B,
- mais des blocs de A et de B sont toujours accédés en même temps par différents processeurs. . .

Répartition ≪ naturelle ≫

Exemple : le produit de deux matrices 3×3

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{bmatrix} =$$

$$\begin{bmatrix} A_{00}B_{00}+A_{01}*B_{10}+A_{02}*B_{20} & A_{00}B_{01}+A_{01}*B_{11}+A_{02}*B_{21} & A_{00}B_{02}+A_{01}*B_{12}+A_{02}*B_{22} \\ A_{10}B_{00}+A_{11}*B_{10}+A_{12}*B_{20} & A_{10}B_{01}+A_{11}*B_{11}+A_{12}*B_{21} & A_{10}B_{02}+A_{11}*B_{12}+A_{12}*B_{22} \\ A_{20}B_{00}+A_{21}*B_{10}+A_{22}*B_{20} & A_{20}B_{01}+A_{21}*B_{11}+A_{22}*B_{21} & A_{20}B_{02}+A_{21}*B_{12}+A_{22}*B_{22} \end{bmatrix}$$

Répartition ≪ naturelle ≫ sur 3×3 processeurs :

A_{00}, B_{00}	A_{01}, B_{01}	A_{02}, B_{02}
A_{10}, B_{10}	A_{11}, B_{11}	A_{12}, B_{12}
A_{20}, B_{20}	A_{21}, B_{21}	A_{22}, B_{22}

Seuls les éléments de la diagonale peuvent être calculés !

Comment placer les données sur un tore ?

L'algorithme de Cannon
Basé sur la modification de la boucle interne :

$$C(i,j) = C(i,j) + \sum_k A(i,k) * B(k,j)$$

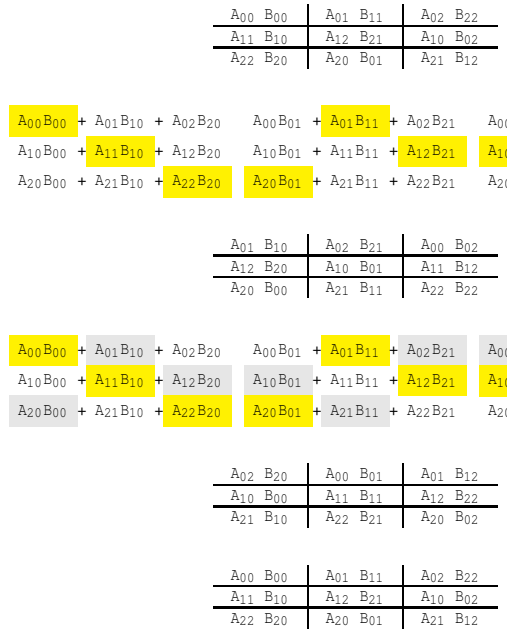
devient

$$C(i,j) = C(i,j) + \sum_k A(i,(i+j+k) mod\ \sqrt{p}) * B((i+j+k) mod\ \sqrt{p}, j)$$

Comment circulent les blocs de A et les blocs de B ?

(i+j + k) mod 3

Déroulement



Algorithme final avec envoi de messages

```
ReceptionBloc(A)
ReceptionBloc(B)
Initialisation du bloc C à 0

pour e=0 a sqrt(p)-1          % circulation des blocs
    pour li=0 a n/sqrt(p)-1
       | pour co=0 a n/sqrt(p)-1
       |    | s=0
       |    | pour d=0 à n/sqrt(p)-1
       |    |    | s = s + A(li,d)*B(d,co)
       |    | C(li,co) += s
    Decalage des blocs de A vers les processeurs de gauche
    Decalage des blocs de B vers les processeurs du haut
```


Prédiction de performance

Avec p processeurs organisés en anneau ou en tore, quel algorithme offre les meilleures performances ? Il faut étudier les performances...

Soit

- $tcom_i$: le temps d'initialisation d'une communication (**latence** du réseau)
- bp : le **débit** (bande passante) du réseau en Mo/s

Le temps de transfert t_c entre deux processeurs voisins de *nbvaleurs* nombres en double précision est :

$$t_c = tcom_i + \frac{8 nbvaleurs}{bp}$$

Soit FLOPS le nombre d'opérations (en virgule flottante) moyen par seconde.
Temps de calcul séquentiel :

$$tcal(n) = \frac{2n^3}{FLOPS}$$

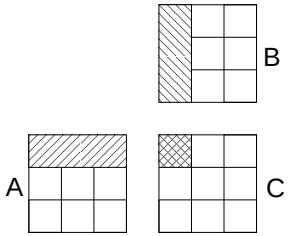
(Rappel : le calcul d'un élément C_{ij} nécessite n multiplications et n

Temps de calcul parallèle en mode bloquant

Pour l'anneau :

- A chaque étape UN processeur multiplie un bloc de taille $(\frac{n}{p}, n)$ par un bloc de taille $(n, \frac{n}{p})$ pour calculer $\frac{n^2}{p^2}$ éléments du résultat final.

  A chaque étape, l'ensemble des processeurs calcule $p.\frac{n^2}{p^2}$ éléments du résultat final.



- Temps de calcul d'un processeur pour une étape :

$$\frac{n^2}{p^2.FLOPS}.2n$$

- Temps de communication d'un processeur pour une étape :

$$tcom_i + \frac{8\frac{n^2}{p}}{bp}$$

Temps de calcul sur un anneau de p processeurs

Temps de calcul total (p étapes) :

$$p\left(\frac{n^2}{p^2.FLOPS}.2n + \left(tcom_i + \frac{8\frac{n^2}{p}}{bp}\right)\right)$$

en simplifiant

$$\frac{2n^3}{p.FLOPS} + p.tcom_i + \frac{8n^2}{bp}$$

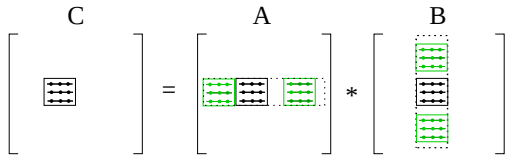
Accélération :

$$Acc(n, anneau) = \frac{\frac{2n^3}{FLOPS}}{\frac{2n^3}{p.FLOPS} + p.tcom_i + \frac{8n^2}{bp}}$$

Dans le cas idéal $Acc(n, anneau)$ tend vers p

Temps de calcul pour le tore de p processeurs

Pour le tore :



- A chaque étape UN processeur multiplie un bloc de taille $(\frac{n}{\sqrt{p}}, \frac{n}{\sqrt{p}})$ par un bloc de taille $(\frac{n}{\sqrt{p}}, \frac{n}{\sqrt{p}})$ pour calculer $\frac{n^2}{p}$ éléments du résultat final.
- Temps de calcul d'un processeur pour une étape :

$$\frac{n^2}{p.FLOPS}.2\frac{n}{\sqrt{p}}$$

- Temps de communication d'un processeur pour une étape :

$$2\left(tcom_i + \frac{8\frac{n^2}{p}}{bp}\right)$$

Temps de calcul pour le tore

Temps de calcul total ($\sqrt{p}$ étapes) :

$$\sqrt{p}\left(\frac{n^2}{p.FLOPS}.2\frac{n}{\sqrt{p}} + 2\left(tcom_i + \frac{8\frac{n^2}{p}}{bp}\right)\right)$$

en simplifiant

$$\frac{2n^3}{p.FLOPS} + 2\left(\sqrt{p}.tcom_i + \frac{8n^2}{bp.\sqrt{p}}\right)$$

Accélération :

$$Acc(n, tore) = \frac{\frac{2n^3}{FLOPS}}{\frac{2n^3}{p.FLOPS} + 2(\sqrt{p}.tcom_i + \frac{8n^2}{bp.\sqrt{p}})}$$

Dans le cas idéal $Acc(n, tore)$ tend vers p

Comparaison entre l'anneau et le tore

L'anneau :

$$\frac{2n^3}{p.FLOPS} + p.tcom_i + \frac{8n^2}{bp}$$

Le tore :

$$\frac{2n^3}{p.FLOPS} + 2\left(\sqrt{p}.tcom_i + \frac{8n^2}{bp.\sqrt{p}}\right)$$

Peut-on optimiser encore l'algorithme ?
Oui : en utilisant des communications non bloquantes pour faire le transfert des blocs de A et de B ($\rightarrow$ recouvrement communications / calcul)
A quelle condition ?
Utilisation de 2 fois plus de mémoire.
Comment le mettre en place ?

Mise en place du recouvrement des communications par du calcul

$\rightarrow$ Algorithme du produit matriciel sur un tore 2D (Cannon).

Routines de communications bloquantes :

- BlockRecv(X, P_{q_1,q_2}) : réception bloquante dans X d'un bloc envoyé par P_{q_1,q_2}
- BlockSend(X, P_{q_1,q_2}) : émission bloquante du bloc X à P_{q_1,q_2}

Routines de communications non bloquantes :

- id = IBlockRecv(X, P_{q_1,q_2}) : réception non bloquante dans X d'un bloc envoyé par P_{q_1,q_2} ; l'appel retourne l'identifiant de requête id
- id = IBlockSend(X, P_{q_1,q_2}) : émission non bloquante du bloc X à P_{q_1,q_2} ; l'appel retourne l'identifiant de requête id
- Wait(id) : attente (bloquante) de la fin de la requête d'identifiant id

Version sans recouvrement communications/calcul

Pré-condition : (q_1, q_2) : identifiants du processeur P_{q_1,q_2} ($0 \leq q_1, q_2 \leq q-1 = \sqrt{p}-1$)

1: /*Initialisation ($e=0$) : */ blocs locaux : $A = A_{q_1,(q_1+q_2) \bmod \sqrt{p}}$, $B = B_{(q_1+q_2) \bmod \sqrt{p},q_2}$, C initialisé à 0 /* correspond à C_{q_1,q_2} */
2:
3: **Pour** $e=0$ à $\sqrt{p}-1$ **faire** /* circulation des blocs */
4: **Pour** $li=0$ à $n/\sqrt{p}-1$ **faire**
5: **Pour** $co=0$ à $n/\sqrt{p}-1$ **faire**
6: $s=0$
7: **Pour** $d=0$ à $n/\sqrt{p}-1$ **faire**
8: $s = s + A(li,d) * B(d,co)$
9: **Fin pour**
10: $C(li,co) += s$
11: **Fin pour**
12: **Fin pour**
13: /*A contient actuellement $A_{q_1,(q_1+q_2+e) \bmod \sqrt{p}}$*/
14: BlockSend(A, $P_{q_1,(q_2-1) \bmod \sqrt{p}}$) et BlockRecv($A$, $P_{q_1,(q_2+1) \bmod \sqrt{p}}$)
15: /*A contient désormais $A_{q_1,(q_1+q_2+e+1) \bmod \sqrt{p}}$*/
16:
17: /*B contient actuellement $B_{(q_1+q_2+e) \bmod \sqrt{p},q_2}$*/
18: BlockSend(B, $P_{(q_1-1) \bmod \sqrt{p},q_2}$) et BlockRecv($B$, $P_{(q_1+1) \bmod \sqrt{p},q_2}$)
19: /*B contient désormais $B_{(q_1+q_2+e+1) \bmod \sqrt{p},q_2}$*/
20: **Fin pour**

Version avec recouvrement communications/calcul : utilisation du double-buffering

Pré-condition : (q_1, q_2) : identifiants du processeur P_{q_1,q_2} ($0 \leq q_1, q_2 \leq q-1 = \sqrt{p}-1$)

1: /* A et B : tableaux locaux de deux blocs */
2: /*Initialisation ($e=0$) : */ blocs locaux : $A[0] = A_{q_1,(q_1+q_2) \bmod \sqrt{p}}$, $B[0] = B_{(q_1+q_2) \bmod \sqrt{p},q_2}$, C initialisé à 0 /* correspond à C_{q_1,q_2} */
3:
4: $i=0$
5: **Pour** $e=0$ à $\sqrt{p}-1$ **faire** /* circulation des blocs */
6: id_{Ai} = IBlockSend($A[i]$, $P_{q_1,(q_2-1) \bmod \sqrt{p}}$)
7: id_{Aii} = IBlockRecv($A[1-i]$, $P_{q_1,(q_2+1) \bmod \sqrt{p}}$)
8: id_{Bi} = IBlockSend($B[i]$, $P_{(q_1-1) \bmod \sqrt{p},q_2}$)
9: id_{Bii} = IBlockRecv($B[1-i]$, $P_{(q_1+1) \bmod \sqrt{p},q_2}$)
10: **Pour** $li=0$ à $n/\sqrt{p}-1$ **faire**
11: **Pour** $co=0$ à $n/\sqrt{p}-1$ **faire**
12: $s=0$
13: **Pour** $d=0$ à $n/\sqrt{p}-1$ **faire**
14: $s = s + A[i](li,d) * B[i](d,co)$
15: **Fin pour**
16: $C(li,co) += s$
17: **Fin pour**
18: **Fin pour**
19: Wait(id_{Ai}) ; Wait(id_{Aii}) ; Wait(id_{Bi}) ; Wait(id_{Bii})
20: $i = 1-i$
21: **Fin pour**

Comment obtenir un programme parallèle efficace ?

Les grands principes :

- **localité des données** : *répartir les données de sorte que chaque processeur dispose localement d'un maximum de données à traiter*
  → très important pour machines DM : réduction des communications
- **équilibrage de charge** (*load balancing*) : *attribuer au mieux les charges de calcul en fonction des caractéristiques de chaque processeur, afin de limiter les périodes d'inactivité des processeurs*
  → machines homogènes : la charge de calcul doit être la même pour chaque processeur
- **recouvrement des communications par le calcul**

Équilibrages de charge

Charge de calcul prédictible ⇒ équilibrage de charge **statique**

- données régulières présentant toutes un même coût de calcul
  → distribution bloc, cyclique, bloc cyclique...
- données régulières présentant des coûts de calcul différents
  → utilisation d'une fonction de coût + distribution bloc, cyclique, bloc cyclique...

Charge de calcul non prédictible ⇒ équilibrage de charge **dynamique** (exemple : fractale de Mandelbrot)

- modèle maître-ouvrier
- modèle auto-régulé

Modèle maître-ouvrier (maître-esclave)

- Le maître connaît les données et le travail.
- Un ouvrier attend du maître soit une demande de calcul (l'ouvrier exécute le calcul et retourne le résultat), soit un ordre de fin.
- Cette solution a cependant des limites :
  - la mémoire locale du maître doit pouvoir parfois charger toutes les données ;
  - 2 envois de messages pour 1 calcul ⟶ nécessité d'une granularité de calcul forte pour une bonne efficacité ;
  - s'il y a trop d'ouvriers, le maître peut être un goulet d'étranglement.
- Avantages :
  - l'équilibrage de charge peut se faire en fonction de l'hétérogénéité du matériel, ou de son occupation partielle (par d'autres utilisateurs)

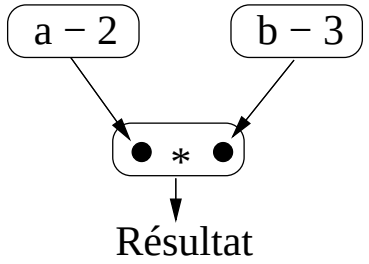
Modèle auto-régulé

Principe du « vol de travail » (*work stealing*) :

- chaque processeur gère sa propre liste de travaux à effectuer ;
- si la liste de travail d'un processeur est vide, il récupère une partie de la liste des autres processeurs.
  - \+ meilleure gestion mémoire
  - \+ tous les processeurs participent au calcul
  - – difficulté de programmation

Le parallélisme de tâches (parallélisme de contrôle)

- Au lieu de découper les données et de les répartir sur les processeurs, on découpe le code en tâches, et on étudie les **dépendances temporelles** entre les tâches :
  - quelles tâches peuvent être exécutées simultanément ?
  - quelles tâches doivent être exécutées en séquence ?
- Exemple : $(a-2)*(b-3)$



Comment effectuer le découpage et le placement ?

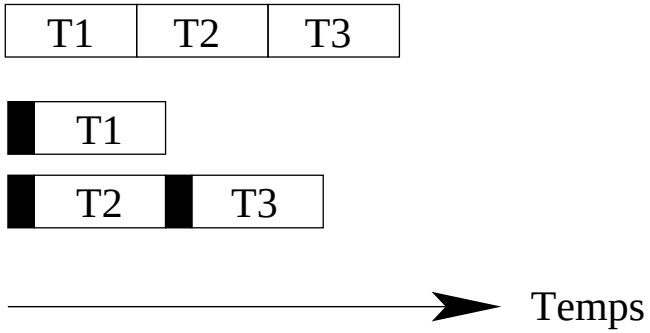
- Prendre en compte le type de machine parallèle et notamment le temps de lancement d'une tâche :
  - si le temps est long $\longrightarrow$ découpage à gros grain,
  - si le temps est court $\longrightarrow$ découpage à grain plus fin.
- Exemple : soit un programme séquentiel composé de 3 tâches T_1, T_2, T_3.
- T_1 et T_2 sont indépendants, T_3 dépend de T_1 et T_2.

```
T1
T2   T3
--------> temps
```


Exemple (suite)

- Temps de calcul d'une tâche : t_c,
- Temps de lancement d'une tâche : t_l,
- Temps du programme séquentiel : $t_s = 3t_c$,
- Temps du programme parallèle : $t_p = 2(t_l + t_c)$.
- Résolvons $t_p > t_s$ avec t_c connu :
  $2(t_l + t_c) > 3t_c$
  $2t_l > t_c \iff t_l > \frac{t_c}{2}$
- Sur cet exemple, si $t_l > \frac{t_c}{2}$ le calcul séquentiel est plus rapide.
  $\Longrightarrow$ nécessité de faire attention à la granularité du découpage.

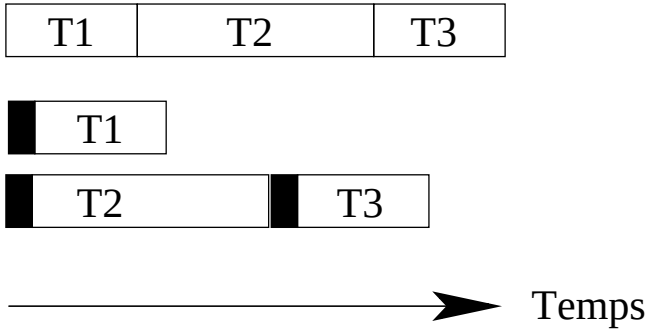
Exemple (suite)



- L'accélération est faible dans ce cas : $\frac{3}{2}$ Maxi

Exemple (suite et fin)

- L'accélération peut être nettement moins bonne si par exemple *T*2 est beaucoup plus long que *T*1.



- Le gain est ici ridicule. Il faut alors essayer de redécouper *T*2 en tâches parallèles plus petites.

Autre exemple

Algorithme d'un serveur HTTP multi-thread :

```
Boucle infinie
     Attendre une requête r d'un client
     Créer un thread qui exécute traiter_requête(r)
Fin de boucle
```

avec :

```
Procédure traiter_requête(requête r)
     Analyser la requête r
     Construire la page HTTP adaptée
     Renvoyer la page HTTP au client
Fin procédure traiter_requête
```