Table des matières

Chapitre 18

Produit parallèle de matrices

18.0 Préambule : Quel est le coût d'une communication sur un *cluster* comme `turing`?

Le programme `ring.c`

Soit le programme C 18.1, qui crée un anneau virtuel de processus et qui fait `nbTours` d'envois de messages, chaque message comportant `taille` octets.

Programme C 18.1 Procédure pour effectuer un certain nombre communications autour d'un «anneau virtuel» de processus. On fait `nbTours` d'envois de messages, chaque message comportant `taille` octets.

```
void executer( int nbTours, int taille )
{
  int numProc, nbProcs;
  MPI_Comm_rank( MPI_COMM_WORLD, &numProc );
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

  // On definit les voisins de l'anneau virtuel.
  int gauche = (numProc + nbProcs - 1) % nbProcs;
  int droite = (numProc + 1) % nbProcs;

  char tampon[taille];

  for ( int i = 0; i < nbTours; i++ ) {
    if ( numProc == 0 )  {
      MPI_Send( tampon, taille, MPI_BYTE, droite, 0,
                MPI_COMM_WORLD );
      MPI_Recv( tampon, taille, MPI_BYTE, gauche, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    } else {
      MPI_Recv( tampon, taille, MPI_BYTE, gauche, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
      MPI_Send( tampon, taille, MPI_BYTE, droite, 0,
                MPI_COMM_WORLD );
    }
  }
}

/*
 Note: le temps d'execution est mesure avec MPI_Wtime():

  double temps = -MPI_Wtime();
  executer( nbTours, taille );
  temps += MPI_Wtime();
*/
```

Effet de l'option `--map-by node`

Voici les temps pour 10 processus qui s'échangent des messages de 100 octets, exécutés sans vs. avec l'option `--map-by node` :

```
# Sans --map-by node
# nbTours    taille      temps
        4       100     0.0018
        8       100     0.0035
       16       100     0.0069
       32       100     0.0116
       64       100     0.0238
      128       100     0.0472
      256       100     0.0879
      512       100     0.1788
     1024       100     0.3384

# Avec --map-by node
# nbTours    taille      temps
        4       100     0.0053
        8       100     0.0083
       16       100     0.0200
       32       100     0.0386
       64       100     0.0763
      128       100     0.1566
      256       100     0.2934
      512       100     0.5994
     1024       100     1.1215
```

Effet de l'augmentation de la taille des messages — avec `--map-by node` pour 10 processeurs

```
# nbTours    taille      temps
        4         1     0.0049
        8         1     0.0116
       16         1     0.0195
       32         1     0.0373
       64         1     0.0730
      128         1     0.1518
      256         1     0.3001
      512         1     0.5844
     1024         1     1.1247
# nbTours    taille      temps
        4       100     0.0058
        8       100     0.0097
       16       100     0.0194
       32       100     0.0406
       64       100     0.0786
      128       100     0.1557
      256       100     0.3023
      512       100     0.5683
     1024       100     1.0711
# nbTours    taille      temps
        4     10000     0.0084
        8     10000     0.0153
       16     10000     0.0294
       32     10000     0.0581
       64     10000     0.1006
      128     10000     0.2231
      256     10000     0.4406
      512     10000     0.8675
     1024     10000     1.7541
```

Donc : Le temps augmente quand le message est «gros», mais pas de façon linéaire!

Effet de l'augmentation du nombre de processeurs — avec `--map-by node` et des messages de 1 octet

```
# nbTours    taille      temps       # 10 processeurs
        4         1     0.0049
        8         1     0.0116
       16         1     0.0195
       32         1     0.0373
       64         1     0.0730
      128         1     0.1518
      256         1     0.3001
      512         1     0.5844
     1024         1     1.1247
# nbTours    taille      temps       # 20 processeurs
        4         1     0.0119
        8         1     0.0238
       16         1     0.0442
       32         1     0.0753
       64         1     0.1591
      128         1     0.3094
      256         1     0.6017
      512         1     1.1803
     1024         1     2.1877
# nbTours    taille      temps       # 30 processeurs
        4         1     0.0162
        8         1     0.0322
       16         1     0.0623
       32         1     0.1217
       64         1     0.2486
      128         1     0.4587
      256         1     0.9043
      512         1     1.6837
     1024         1     3.3193
```

Donc : Le temps augmente de façon *presque* linéaire!

À quoi correspond le coût de l'envoi/réception d'un message d'un octet?

```
10 processeurs: 1.1247 / (1024*10) => 0.000109833984375
20 processeurs: 2.1877 / (1024*20) => 0.000106821289063
30 processeurs: 3.3193 / (1024*30) => 0.000108050130208

Moyenne
  = 0.00010882975260416666
  ≈ 1.1e-04
```

Soit le programme `heat-seq.c` qui simule la diffusion de la chaleur dans un cylindre :

```
...
double tempsEcoule = -MPI_Wtime();
for ( int k = 0; k < NSTEPS; k++ ) {
  ukp1[0] = uk[0];
  ukp1[NX-1] = uk[NX-1];
  for ( int i = 1; i < NX-1; i++ ) {
    ukp1[i] = uk[i]+(dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
  }
  temp = ukp1; ukp1 = uk; uk = temp;
}
tempsEcoule += MPI_Wtime();
printf( "NX = %d; NSTEPS = %d %10.8f\n", NX, NSTEPS, tempsEcoule );
...
```

Combien d'itérations et de points peuvent être traités en 1.1e-04 seconde?
Temps pour effectuer 7 itérations sur 1000 points :

```
$ ./heat-seq
NX = 1000; NSTEPS = 7   0.00011
```

18.1 Introduction

Ce chapitre présente trois stratégies pour effectuer le produit parallèle de matrices avec échange de messages. Ces trois stratégies reposent sur une distribution des données *par blocs — voir section suivante.*

18.2 Distribution des données par bloc et solution alternative pour le calcul du produit

Pourquoi une distribution par blocs

Les explications motivant l'approche de distribution par blocs ont été données en cours. Elles reposent sur le fait que le *ratio coûts des communication / coûts des calculs* est plus intéressant dans le cas de la distribution par blocs, donc le programme résultant est plus «*dimensionable*» («*scalable*»).

Ainsi, soit p le nombre de processeurs — un carré parfait :

- Approche par groupe de lignes adjacentes : Le ratio est proportionnel à p.
- Approche par blocs : Le ratio est proportionnel à $\sqrt{p}$.

En d'autres mots, avec une distribution par blocs, les coûts des communications augmentent moins rapidement lorsqu'on augmente le nombre de processeurs — en fonction de $\sqrt{p}$ plutôt qu'en fonction de p (augmentation linéaire).

Illustration de la distribution d'une matrice 3×3

On veut multiplier deux matrices A et B pour obtenir une matrice C. L'idée de base des stratégies est présentée avec une petite matrice 3×3. Toutefois, la définition du produit matriciel s'applique que les éléments des matrices — les $a_{i,j}, b_{i,j}$ et $c_{i,j}$ — soient des nombres... **ou des sous-matrices** :

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{pmatrix}$$

$$C = A * B = \begin{pmatrix} \sum_{k=0}^{2} a_{0,k} * b_{k,0} & \sum_{k=0}^{2} a_{0,k} * b_{k,1} & \sum_{k=0}^{2} a_{0,k} * b_{k,2} \\ \sum_{k=0}^{2} a_{1,k} * b_{k,0} & \sum_{k=0}^{2} a_{1,k} * b_{k,1} & \sum_{k=0}^{2} a_{1,k} * b_{k,2} \\ \sum_{k=0}^{2} a_{2,k} * b_{k,0} & \sum_{k=0}^{2} a_{2,k} * b_{k,1} & \sum_{k=0}^{2} a_{2,k} * b_{k,2} \end{pmatrix}$$

La figure 18.1 présente une décomposition en neuf (9) «blocs» pour neuf (9) processus d'une matrice 3×3, où chaque bloc indique les produits qui doivent être effectués et additionnés pour calculer le résultat de C_{ij} dont est responsable le

processus P_{ij} — pour une matrice de plus grande taille, il suffirait d'intepréter les aij comme des sous-matrices, i.e., de vrais «gros blocs» d'éléments.

Plus précisément, le processus P_{ij} est «responsable» des éléments d'index i, j de chacune des matrices, i.e., il doit initialement stocker les éléments appropriés des matrices de départ A et B, et calculer/stocker l'élément correspondant de la matrice résultat C. Dans la figure 18.1, les cercles en pointillés indiquent l'élément a et l'élément b stockés par le processus P_{ij} ; pour tous les autres éléments, des communications avec d'autres processus sont donc nécessaires.

Idée générale des stratégies

L'idée génerale derrière toutes les stratégies étudiées est d'effectuer les différents produits et sommes requis *en différentes phases*. Ces phases peuvent être comprises en organisant les boucles de calcul du produit matriciel d'une façon différente de celle utilisée habituellement.

Soit des matrices A et B de taille $N \times N$ pour lesquelles ont veut calculer $C = A \times B$.

Le programme C 18.2 présente la solution «**standard**» habituelle. Les programmes C 18.3 et 18.4, quant à eux, présentent une solution alternative où les boucles ont été organisées de façon différente — **la boucle la plus interne est devenue la boucle externe**.

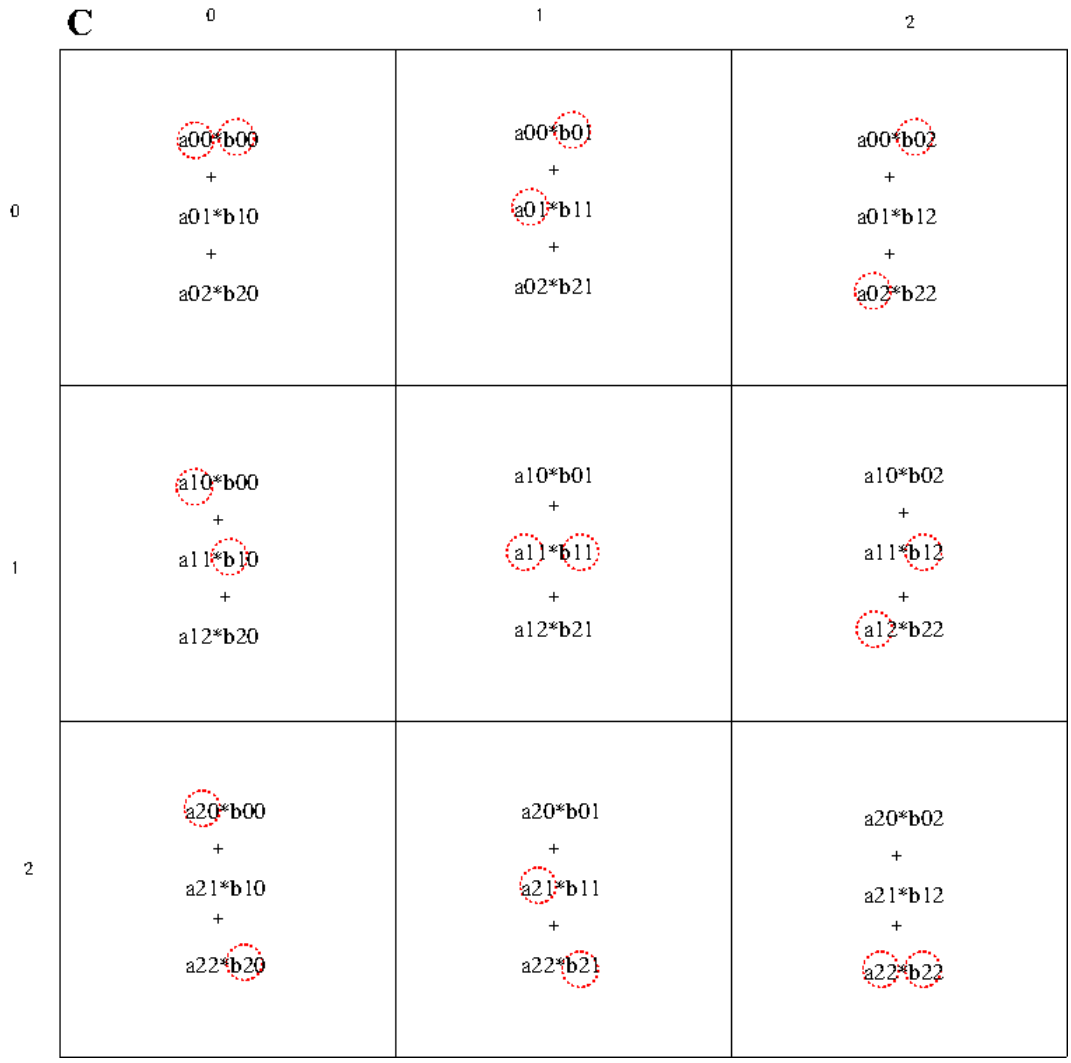


Figure 18.1: Matrice résultat `C` : calculs (produits et sommes) à effectuer pour une matrice 3×3 (avec 9 blocs, pour 9 processus). Les items **en pointillés** indiquent des éléments qui sont **locaux** au processus, à cause de la distribution par blocs. Tous les autres items sont non-locaux, donc doivent être obtenus via des communications.

Programme C 18.2 Solution «**standard**» pour le produit matriciel — l'itération sur `k` est dans la boucle interne.

```
for ( int i = 0; i < N; i++ ) {
  for ( int j = 0; j < N; j++ ) {
    C[i,j] = 0.0;
    for ( int k = 0; k < N; k++ ) {
      C[i,j] += A[i,k] * B[k,j];
    }

  }
}
```

Programme C 18.3 Solution «**alternative**» pour le produit matriciel avec réorganisation des boucles — **la boucle interne devient la boucle externe**, mais on doit faire une passe préalable d'initialisation de `C`.

```
for ( int i = 0; i < N; i++ ) {
  for ( int j = 0; j < N; j++ ) {
    C[i,j] = 0.0;
  }
}

for ( int k = 0; k < N; k++ ) {
  for ( int i = 0; i < N; i++ ) {
    for ( int j = 0; j < N; j++ ) {
      C[i,j] += A[i,k] * B[k,j];
    }
  }
}
```

Programme C 18.4 Solution «**alternative**» pour le produit matriciel avec réorganisation des boucles — **la boucle interne devient la boucle externe** — et avec utilisation de procédures auxiliaires `initialize` et `matmul_add`.

```
void initialize( double C[][], int N, double v ) {
  for ( int i = 0; i < N; i++ )
    for ( int j = 0; j < N; j++ )
      C[i,j] = v;
}

void matmul_add( int k, double A[][], double B[][],
                 double C[][], int N ) {
  for ( int i = 0; i < N; i++ ) {
    for ( int j = 0; j < N; j++ ) {
      C[i,j] += A[i,k] * B[k,j];
    }
  }
}

initialize( C, N, 0.0 );

for ( int k = 0; k < N; k++ ) {
  matmul_add( k, A, B, C, N );
}
```

18.3 Méthode de Mattson, Sanders et Massingill [MSM05]

Ordre des calculs et communications

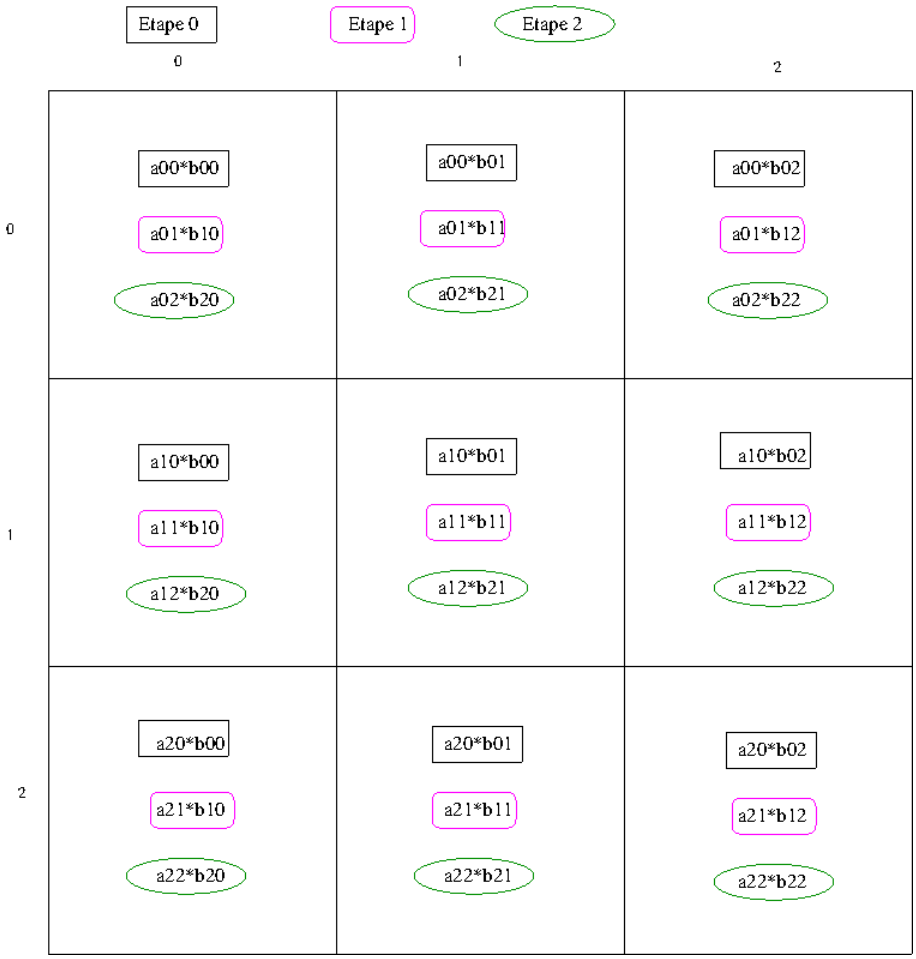


Figure 18.2: Ordre de calcul des produits dans la méthode de Mattson, Sanders et Massingil [MSM05] (voir code plus loin).

La figure 18.2 illustre quels calculs (produits) sont effectués à chacune des étapes ($k = 0, 1, 2$) du programme MPI vu en cours (adaptation de [MSM05]). Dans cette méthode de calcul, chacun des processus P_{ij} effectue, à l'étape k ($k = 0, 1, \ldots, n-1$), le produit de $a_{ik} \cdot b_{kj}$.

La figure 18.3, quant à elle, indique les communications qui doivent être effectuées *avant le début des calculs* de la zéroième étape, pour assurer que les dépendances de

données soient correctement satisfaites — une figure équivalente, décalée de façon appropriée, pourrait être produite pour les étapes 1 et 2. Les items entourés d'un petit rond sont ceux qui doivent être transmis, par le processus P_{ij}. Les figures qui suivent illustrent les communications aux étapes subséquentes.

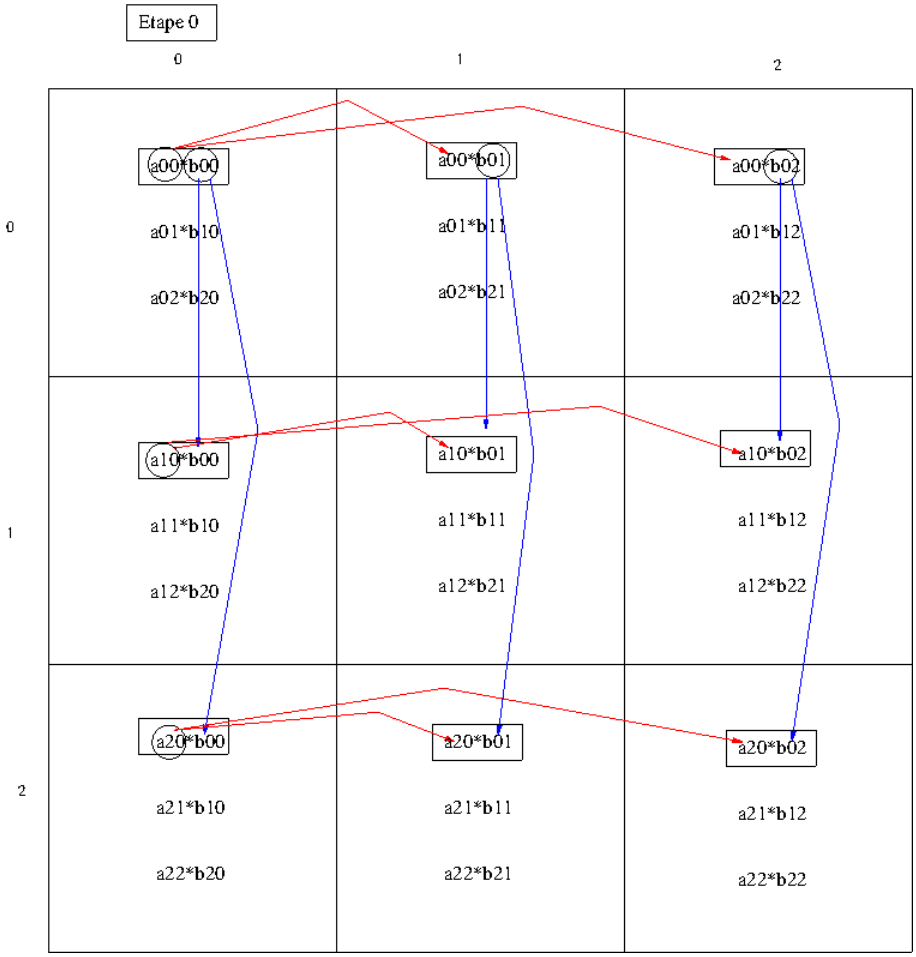


Figure 18.3: Communications à effectuer *au début de* l'étape numéro zéro (0), pour que les données requises soient disponibles (méthode Mattson, Sanders et Massingil [MSM05]).

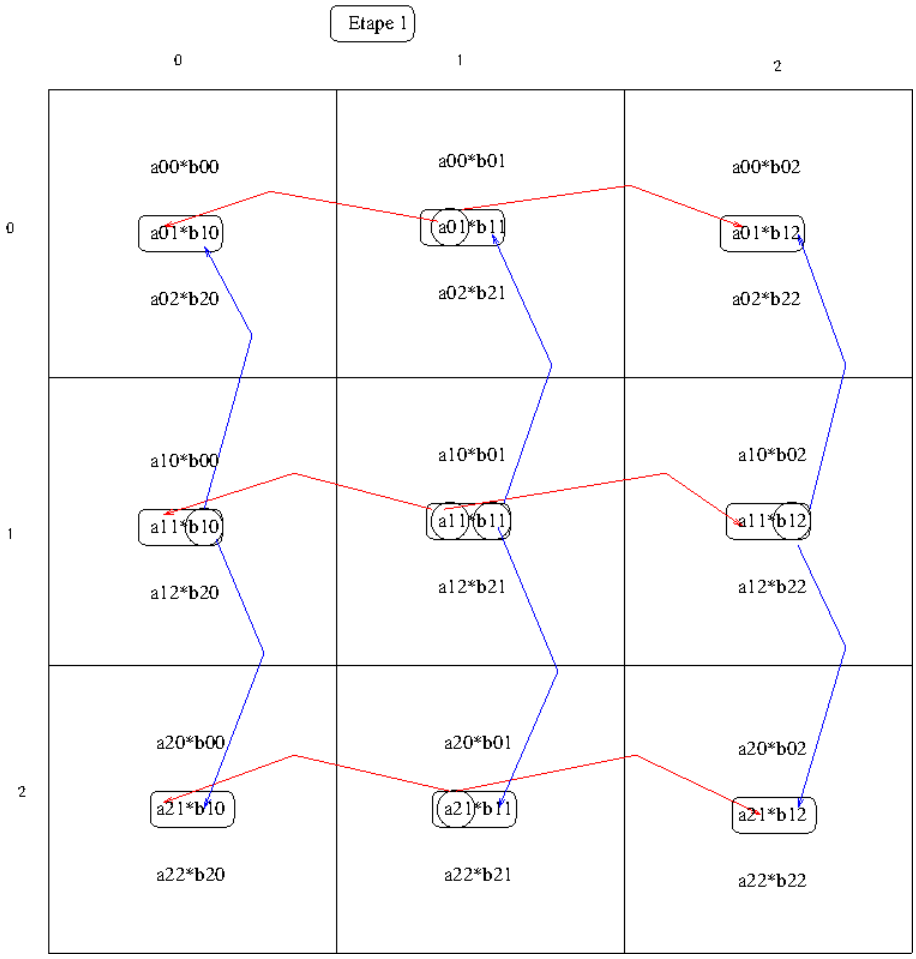


Figure 18.4: Communications à effectuer *au début de* l'étape numéro un (1), pour que les données requises soient disponibles (méthode Mattson, Sanders et Massingil [MSM05]).

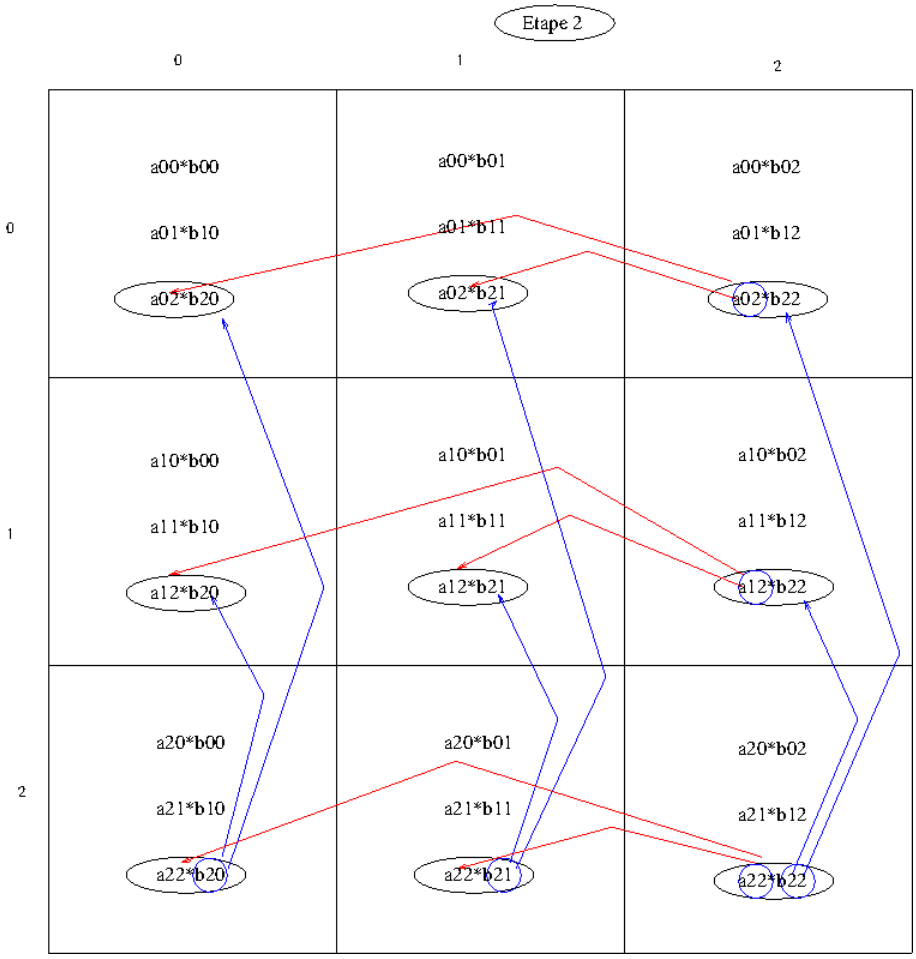


Figure 18.5: Communications à effectuer *au début de* l'étape numéro deux (2), pour que les données requises soient disponibles (méthode Mattson, Sanders et Massingil [MSM05]).

Code MPI — fichier `mm-bcast.c`

Pour la misen en oeuvre MPI, on va attribuer aux divers processus deux `ID`s additionnels, relatifs à une grille, qui vont indiquer la position du processus en terme du numéro de ligne ou de colonne dans la grille.

Par exemple, soit :

- `numProcs = 9` — nombre de processeurs
- `NB = 3` — nombre de blocs par ligne ou colonne
- `myID =` rang du processus dans `MPI_COMM_WORLD`
- `myID_i = myID / NB`
- `myID_j = myID % NB`

	myID	myID_i	myID_j
P_0	0	0	0
P_1	1	0	1
P_2	2	0	2
P_3	3	1	0
P_4	4	1	1
P_5	5	1	2
P_6	6	2	0
P_7	7	2	1
P_8	8	2	2

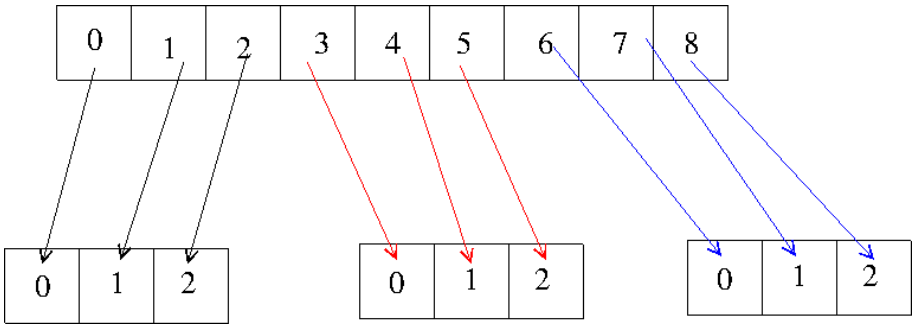
P_0	P_1	P_2
P_3	P_4	P_5
P_6	P_7	P_8

Soit alors les instructions `MPI_Comm_split` suivantes :

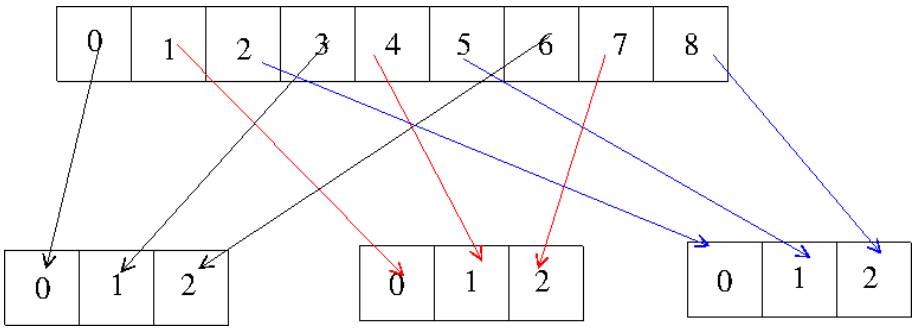
```
MPI_Comm lignes, colonnes;

MPI_Comm_split( MPI_COMM_WORLD, myID_i, MPI_UNDEFINED,
                &lignes );

MPI_Comm_split( MPI_COMM_WORLD, myID_j, MPI_UNDEFINED,
                &colonnes );
```

Ces instructions créeront alors les communicateurs ci-bas.

```
MPI_Comm_split( MPI_COMM_WORLD, myID_i, MPI_UNDEFINED, &lignes );
```




```
MPI_Comm_split( MPI_COMM_WORLD, myID_j, MPI_UNDEFINED, &colonnes );
```



Le fichier suivant donne le code MPI pour cette approche :
`http://www.labunix.uqam.ca/~tremblay/INF7235/Materiel/mult-mat-par.c`

Le code est présenté dans les pages qui suivent.

```
/*
 * Solution parallele pour le produit de matrices, avec allocation
 * dynamique des matrices et distribution par blocs.
 *
 * Chaque matrice est allouee sous forme d'un <<vecteur>> lineaire de
 * la taille appropriee. Les diverses routines doivent donc manipuler
 * explicitement les pointeurs plutot que de faire des simples
 * operations d'indexation.
 *
 * Source: Le code est tire puis adapte de "Patterns for Parallel
 * Programming", T.G. Mattson, B.A. Sanders and B.L. Massingill,
 * Addison-Wesley, 2005.
 *
 * Diverses modifications ont ete apportees pour simplifier le code,
 * notamment dans la facon d'initialiser les matrices, et ce dans le
 * but de verifier (avec des assertions) que le resultat final est
 * bien celui attendu. Le style des declarations a aussi ete quelque
 * peu modifie, notamment en utilisant le plus possible des
 * declarations locales (le plus pres du point d'utilisation). Les
 * echanges de blocs se font par l'intermediaire de Bcast a des
 * sous-groupes (memes lignes ou meme colonnnes) plutot que par des
 * envois point-a-point. Finalement, le code a ete simplifie pour
 * traiter uniquement des matrices carrees.
 *
 * Argument du programme:
 * - Facteur multiplicatif pour taille des matrices a multiplier
 *
 * Note: Pour assurer le respect de la condition que la taille soit
 *       divisible par la racine carre du nombre de processeurs,
 *       l'argument N ne determine pas la taille de facon exacte mais
 *       un facteur multiplicatif d'un nombre predefini produit des
 *       premiers entiers 2, 3, etc.
 */

#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
//
```

```
////////////////////////////////////////////////////////////////////
// Signature des operations auxiliaires sur les matrices.
////////////////////////////////////////////////////////////////////

/* Les operations qui suivent  sont liees a la representation interne utilise
 * a savoir, une matrice est representee par un vecteur (lineaire) --
 * donc il faut "jouer" de facon explicite avec l'adresse de base, les indice
 * la taille des blocs, etc.
 */

// Initialisation d'une matrice avec une valeur v partout (pour simplifier te
void initialize( double* A, int N, double V );

// Operation pour acceder a l'element A[i,j], et ce a partir de
// l'adresse de base de la matrice et de la taille des blocs.
double get( double* A, int i, int j, int monN );

// Operation qui fait le produit des blocs indiques de A et B et
// additionne le resultat au bloc correspondant de C.
void matmul_add( double* A, double* B, double* C, int monN );

////////////////////////////////////////////////////////////////////
// Operations pour debogage et test.
////////////////////////////////////////////////////////////////////

// Impression du contenu d'une matrice -- pour debogage.
void matprint( char *nom, double* A, int monN );

// Constantes pour definir le contenu des matrices A et B et assurer
// que l'on peut tester facilement, avec des assertions, que le
// produit resultant est correct.
#define V1 2.0
#define V2 3.0

// Verification du contenu d'une matrice -- pour tests.
void verifier( double* A, int Nb, double res );

// Nom d'un bloc du resultat -- pour debogage.
char *mkName( char *nom, int myID_i, int myID_j, int myID );


//
```

```
////////////////////////////////////////////////////////////////////
// Fonction auxiliaire pour la distribution des blocs de donnees.
////////////////////////////////////////////////////////////////////

void bcastBloc( double* bloc, double* buffer, int nbElements, int source,
                MPI_Comm communicator )
{
  int myID;
  MPI_Comm_rank( communicator, &myID );

  if ( myID == source )
    memcpy( buffer, bloc, nbElements * sizeof(double) );

  MPI_Bcast( buffer, nbElements, MPI_DOUBLE, source, communicator );
}

//
```

```
////////////////////////////////////////////////////////////////////
// Programme principal.
////////////////////////////////////////////////////////////////////

int main( int argc, char *argv[] )
{
  // On initialise MPI et on identifie les arguments.
  int numProcs, myID;
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &numProcs );
  MPI_Comm_rank( MPI_COMM_WORLD, &myID );

  // Lecture de la taille des matrices (carres) a traiter.  Pour
  // simplifier le traitement par blocs, on s'arrange pour que cette
  // taille soit presqu'assurement divisible par la racine carre du
  // nombre de processeurs.
  int N = 2 * 2 * 3 * 5 * 7 * atoi(argv[1]);

  // NB = Nombre de blocs selon une dimension donnee,
  //    => NB X NB = numProcs
  int NB  = (int) sqrt( (float) numProcs );
  assert(  N % NB == 0        && "Mauvaise valeur pour N ou NB: NB ne divise
  assert( NB * NB == numProcs && "Mauvais nombre de processeurs: NB*NB != N"

  // Numeros de ligne et de colonne du processus local.
  int myID_i = myID / NB;
  int myID_j = myID % NB;

  // On demarre la minuterie.
  double tempsEcoule;
  MPI_Barrier( MPI_COMM_WORLD );
  tempsEcoule = -MPI_Wtime();

  //
```

```
// On alloue l'espace local pour les blocs.
int monN = N / NB;
double* A = malloc( monN * monN * sizeof(double) );
double* B = malloc( monN * monN * sizeof(double) );
double* C = malloc( monN * monN * sizeof(double) );

// On initialise les matrices a traiter.
initialize( A, monN, V1 );
initialize( B, monN, V2 );

// On definit les sous-groupes de processus qui sont sur la meme
// ligne ou meme colonne.
MPI_Comm lignes, colonnes;
MPI_Comm_split( MPI_COMM_WORLD, myID_i, MPI_UNDEFINED, &lignes );
MPI_Comm_split( MPI_COMM_WORLD, myID_j, MPI_UNDEFINED, &colonnes );

// On effectue le produit a l'aide des NB phases.
initialize( C, monN, 0.0 );

double* Abuffer = malloc( monN * monN * sizeof(double) );
double* Bbuffer = malloc( monN * monN * sizeof(double) );

for ( int kb = 0; kb < NB; kb++ ) {
  // On effectue le transfert des donnees.
  bcastBloc( A, Abuffer, monN*monN, kb, lignes );
  bcastBloc( B, Bbuffer, monN*monN, kb, colonnes );

  // On traite le bloc de donnees.
  matmul_add( Abuffer, Bbuffer, C, monN );
}

//
```

```
  // On mesure le temps requis pour effectuer le produit.
  double tempsMax;
  tempsEcoule += MPI_Wtime();
  MPI_Reduce( &tempsEcoule, &tempsMax, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WO

  // On verifie formellement que le resultat est le bon.
  verifier( C, monN, ((double)N) * V1 * V2 );

  // On imprime le temps requis et on termine.
  if ( myID == 0 ) {
    printf( "Resultat = OK pour N = %d\n", N );
    printf( "Temps = %6.2f ms\n", 1000.0*tempsMax );
  }

  MPI_Finalize();

  return( 0 );
}


//
```

```
////////////////////////////////////////////////////////////////////
// Mise en oeuvre des diverses operations auxiliaires.
////////////////////////////////////////////////////////////////////

//
// Operations pour manipulation des matrices par bloc.
//
void initialize( double* A, int monN, double v )
{
  double* pt = A;
  for ( int i = 0; i < monN; i++ ) {
    for ( int j = 0; j < monN; j++ ) {
      *pt++ = v;
    }
  }
}

#define get( A, i, j, monN )\
  (*((A) + (i)*(monN) + (j)))

void matmul_add( double* A, double* B, double* C, int monN )
{
  double* pt = C;

  for ( int i = 0; i < monN; i++ ) {
    for ( int j = 0; j < monN; j++ ) {

      double r = 0.0;
      for ( int k = 0; k < monN; k++ ) {
        r += get(A, i, k, monN) * get(B, k, j, monN);
      }
      *pt++ += r;

    }
  }
}


//
```

```
//
// Operations pour debogage et test.
//

void matprint( char *nom, double* A, int monN )
{
  if ( nom ) {
    printf( "%s::\n", nom );
  }

  double* pt = A;
  for ( int i = 0; i < monN; i++ ) {
    for ( int j = 0; j < monN; j++ ) {
      printf( " %5.1f", *pt );
      pt++;
    }
    printf( "\n" );
  }
  fflush( stdout );
}

char *mkName( char *nom, int myID_i, int myID_j, int myID )
{
  char *n = malloc( 128 * sizeof(char) );

  sprintf( n, "%s[%d, %d]@%d", nom, myID_i, myID_j, myID );

  return( n );
}

void verifier( double* A, int monN, double res )
{
  double* pt = A;

  for ( int i = 0; i < monN; i++ ) {
    for ( int j = 0; j < monN; j++ ) {
      assert( *pt++ == res );
    }
  }
}
```

Quelques explications :

- `numProcs` = Nombre de processeurs spécifié à l'appel de `mpirun` — qui doit être un carré parfait.
- Tel qu'indiqué plus haut, la programme effectue le produit `C` = `A` × `B`, avec `A`: `N` × `N`, `B`: `N` × `N` et `C`: `N` × `N`.
- `NB` = Nombre de sous-blocs sur une ligne ou sur une colonne, donc `NB`×`NB` = `numProcs`.
- `monN` = nombre d'éléments (`double`) dans un bloc, donc `monN = N / NB`.
- `matmul_add( A, B, C, ... );` : Fait le calcul matriciel C += A * B, via la manipulation de blocs.
- `get( A, i, j, monN )` : Obtient l'élément `A[i,j]` du bloc.

Performances

La figure 18.6 présente les accélérations obtenues sur `turing.hpc.uqam.ca` (30 processeurs physiques) pour différentes tailles de matrices (`N X N`) et différents nombres de processus (donc avec des processeurs *virtuels* lorsque le nombre de processus est supérieur à 30).

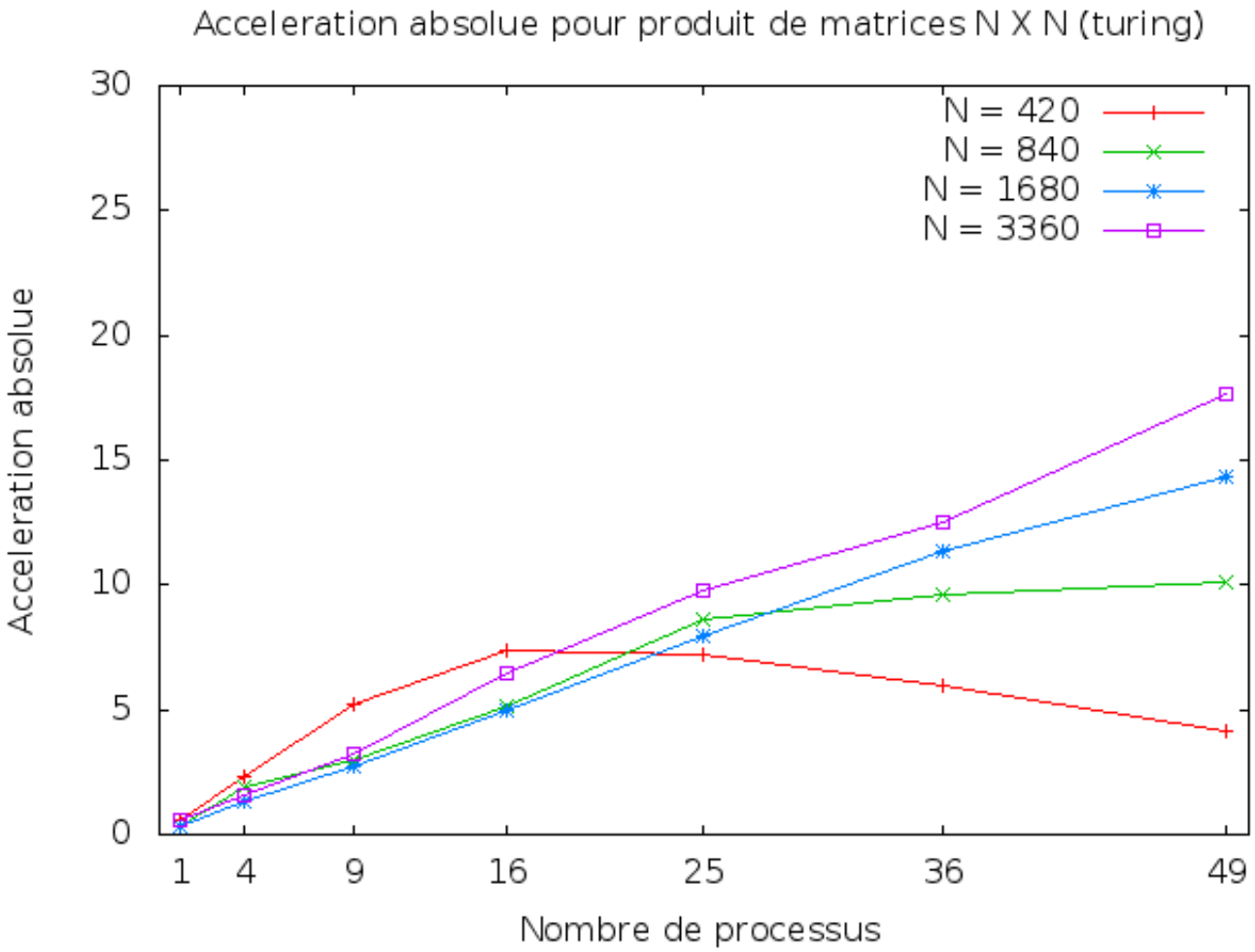


Figure 18.6: Accélérations absolues obtenues sur `turing.hpc.uqam.ca` (30 noeuds, 4 processeurs/noeud) avec la méthode de Mattson, Sanders et Massingil, et ce pour différentes tailles de matrices et différents nombres de processus. Les tailles de matrices ont été choisies pour que les blocs soient tous de taille égale, et ce pour différents nombre de processus qui sont eux-mêmes des carrés ($2^2, 3^2, 4^2, 5^6, 6^2, 7^2$).

18.4 Méthode de Fox [Pac97]

La méthode de Fox [Pac97] effectue les divers calculs de produits dans un ordre différent, comme l'illustre la figure 18.7. Plus précisément, à l'étape k ($k = 0, 1, \ldots, n-1$), le processus P_{ij} effectue le calcul suivant de c_{ij}, où $\bar{k} = (i + k)\ mod\ n$:

$$c_{ij} \texttt{ += } a_{i\bar{k}} \cdot b_{\bar{k}j}$$

Comme précédemment, les cercles pointillés indiquent des données locales au processus P_{ij} : on remarque que les b_{ij} sont déjà tous au bon endroit pour effectuer la zéroième étape, mais pas les a_{ij}. Avant d'effectuer les calculs des produits de la zéroième étape, les éléments a appropriés doivent donc être transmis aux autres processus, tel que cela est illustré par les flèches (normales) sur la figure 18.8.

En examinant la figure 18.7, on peut aussi remarquer qu'à l'étape 1, ces sont les valeurs b_{ij} utilisées à l'étape 0 qui seront utilisées pour les produits calculés à cette nouvelle étape — et ainsi de suite pour chacune des étapes subéquentes.

À une étape de l'algorithme de Fox, on va donc effectuer les opérations suivantes :

- Un des processus d'une rangée va transmettre aux autres processus de sa rangée sa valeur de a.
- Chaque processus va effectuer le produit approprié.
- Dans chacune des colonnes, on va «décaler» (vers le haut, de façon cyclique) les valeurs de b pour qu'elles soient disponibles à l'étape suivante de calcul.

Les diverses communications sont illustrées, pour la zéroième étape, à la figure 18.8, où les flèches en pointillées représentent des «décalages» (vertical, au niveau des colonnes, et cyclique). Signalons aussi qu'à l'étape subséquente, pour chacun des processus, ce sera le b reçu qui sera à nouveau transmis pour poursuivre le décalage vertical, ce qui assurera que tous les éléments b de la colonne auront ultimement été traités par chacun des processus — voir figure 18.9.

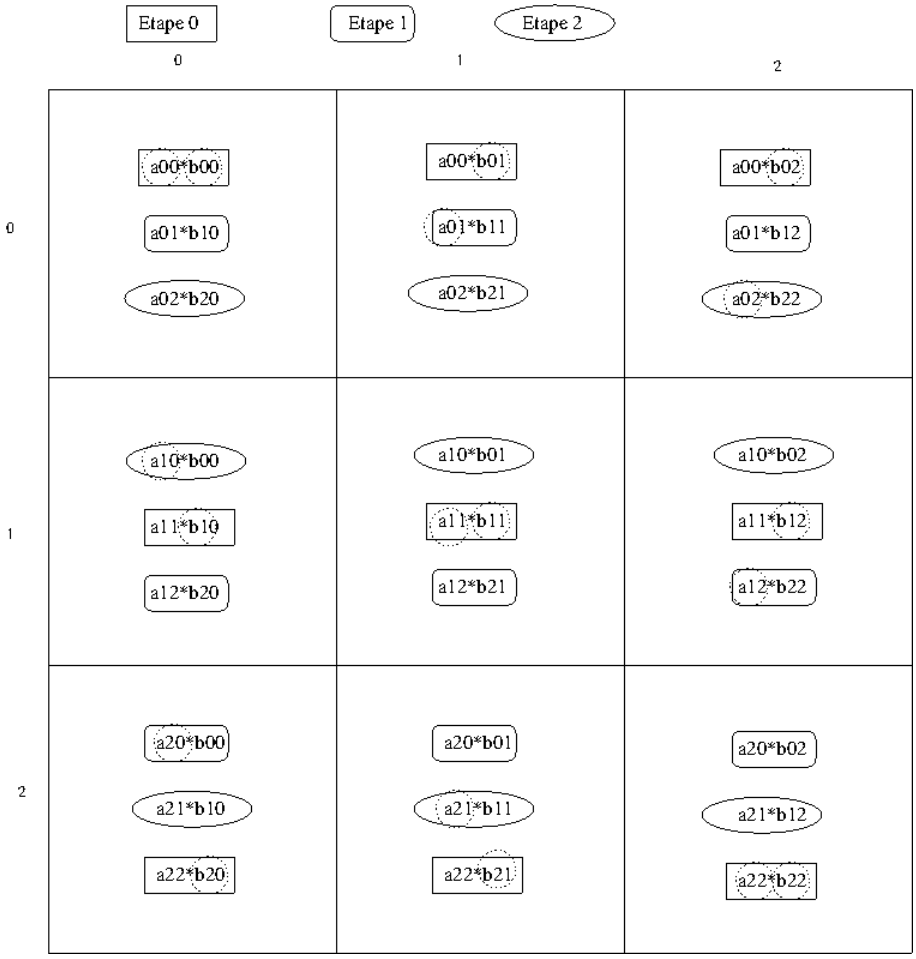


Figure 18.7: Ordre de calcul des produits dans la méthode de Fox.

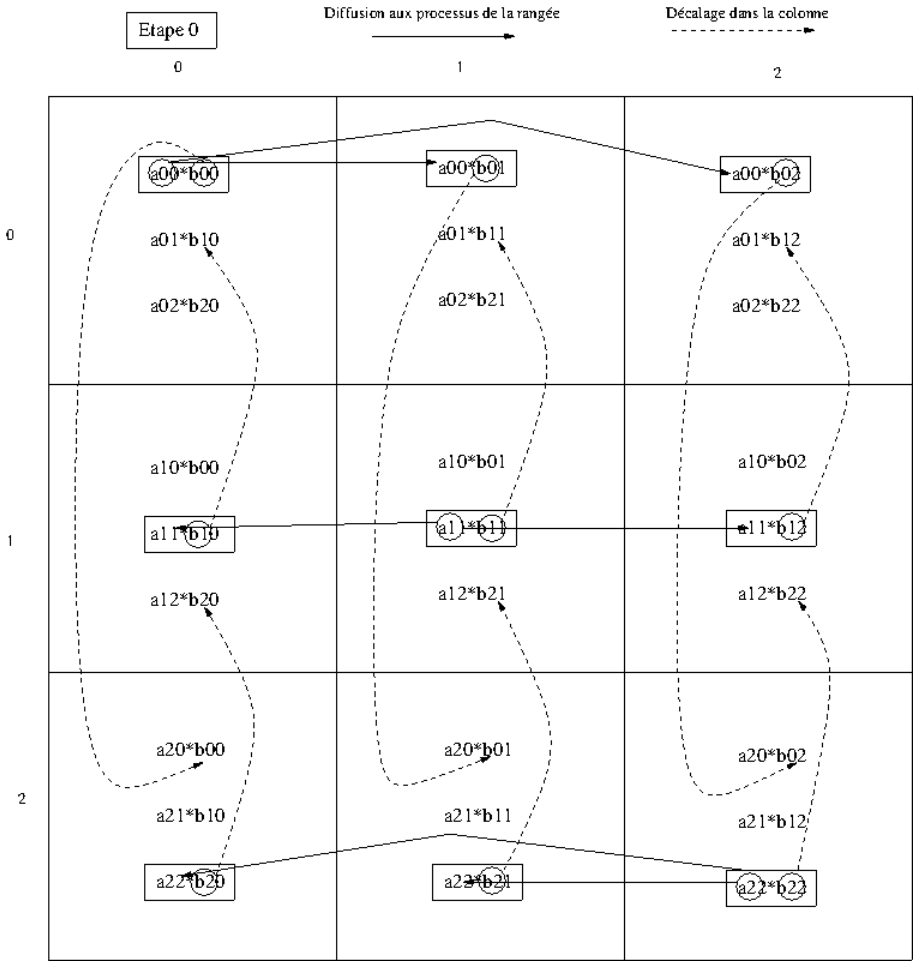


Figure 18.8: Communications lors de la zéroième étape dans la méthode de Fox.

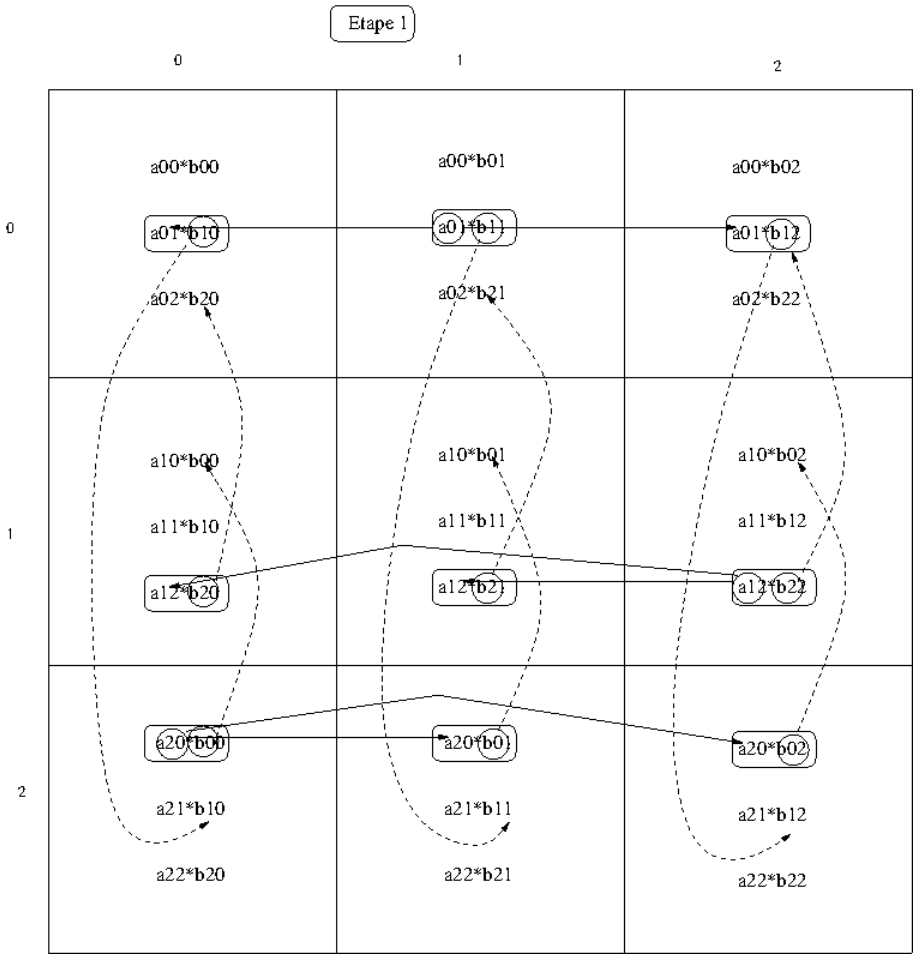


Figure 18.9: Communications lors de l'étape 1 dans la méthode de Fox.

18.5 Méthode de Cannon

La première méthode n'utilise que des diffusions (dans les rangées et dans les colonnes). La méthode de Fox utilise des diffusions dans les rangées, mais des décalages répétitifs dans les colonnes. La méthode de Cannon [Qui03], quant à elle, n'utilise que des décalages, tant au niveau des rangées que des colonnes.

Références

[MSM05] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[Pac97] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman Publ., 1997.

[Qui03] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.