Table des matières

15 Exemples de programmes MPI **2**
15.1 Petit pipeline simple avec trois processus 2
15.2 Intégration numérique 9
15.3 Fonction `mystere` 11
15.4 Programme pour simuler la diffusion de la chaleur dans un cylindre 13

Références **22**

Chapitre 15

Exemples de programmes MPI

15.1 Petit pipeline simple avec trois processus

Il s'agit ici d'une version MPI du même programme vu en Ruby ainsi qu'en Go : le premier processus génère une série de valeurs, le deuxième processus transforme ces valeurs en les multipliant par 10, le troisième processus fait la somme des valeurs.

Le programme principal version Ruby

```
c1, c2, c3, c4 = Array.new(4) { PRuby::Channel.new }

p1.go( c1, c2 )
p2.go( c2, c3 )
p3.go( c3, c4 )

c1 << 10

puts c4.get # => 550
```

Les trois processus version Ruby

```
p1 = lambda do |cin, cout|
  n = cin.get
  (1..n).each { |i| cout << i }
  cout.close
end

p2 = lambda do |cin, cout|
  cin.each { |v| cout << 10 * v }
  cout.close
end

p3 = lambda do |cin, cout|
  cout << cin.reduce(0) { |r, v| r + v }
  cout.close
end
```

Le programme principal version MPI/C

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

// Tag utilise pour la transmission de donnees ou resultats
const int DONNEES = 0;

// Tag utilise pour signaler la fin du flux, lorsque necessaire
const int EOS = 1;


...

//
```

```
int main( int argc, char *argv[] )
{
  // On initialise MPI.
  int numProc, nbProcs;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &numProc );
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

  // On 'dispatche' les trois processus, en plus du programme principal.
  if ( numProc == 0 ) {
    int x = 10;
    MPI_Send( &x, 1, MPI_INT, 1, DONNEES, MPI_COMM_WORLD );
    MPI_Recv( &x, 1, MPI_INT, 3, DONNEES, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE );
    printf( "resultat = %d\n", x );
  } else {
    switch( numProc ) {
    case 1:
      p1( 0, 2 ); break;
    case 2:
      p2( 1, 3 ); break;
    case 3:
      p3( 2, 0 ); break;
    }
  }

  MPI_Finalize();

  return( 0 );
}
```

Les trois processus version MPI/C

```
void p1( int voisinGauche, int voisinDroite )
{
  // On recoit le nombre d'elements a generer.
  int n;
  MPI_Recv( &n, 1, MPI_INT, voisinGauche, DONNEES,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );

  // On genere les elements, puis l'indicateur de fin du flux.
  for( int i = 1; i <= n; i++ ) {
    MPI_Send( &i, 1, MPI_INT, voisinDroite, DONNEES,
              MPI_COMM_WORLD );
  }
  MPI_Send( NULL, 0, MPI_BYTE, voisinDroite, EOS,
            MPI_COMM_WORLD );
}

//
```

```
void p2( int voisinGauche, int voisinDroite )
{
  int eos = 0;

  while (!eos) {
    int v;
    MPI_Status statut;

    MPI_Recv( &v, 1, MPI_INT, voisinGauche, MPI_ANY_TAG,
              MPI_COMM_WORLD, &statut );

    if ( statut.MPI_TAG == DONNEES ) {
      v *= 10;
      MPI_Send( &v, 1, MPI_INT, voisinDroite, DONNEES,
                MPI_COMM_WORLD );
    } else {
      assert( statut.MPI_TAG == EOS );
      MPI_Send( NULL, 0, MPI_BYTE, voisinDroite, EOS,
                MPI_COMM_WORLD );
      eos = 1;
    }
  }
}

//
```

```
void p3( int voisinGauche, int voisinDroite )
{
  int r = 0;
  int eos = 0;

  while (!eos) {
    int v;
    MPI_Status statut;

    MPI_Recv( &v, 1, MPI_INT, voisinGauche, MPI_ANY_TAG,
              MPI_COMM_WORLD, &statut );

    if ( statut.MPI_TAG == DONNEES ) {
      r += v;
    } else {
      assert( statut.MPI_TAG == EOS );
      MPI_Send( &r, 1, MPI_INT, voisinDroite, DONNEES,
                MPI_COMM_WORLD );
      eos = 1;
    }
  }
}
```

15.2 Intégration numérique

```
////////////////////////////////////////////////////////////////////
//
// Fonction pour integration numerique a l'aide d'une quadrature
// composite utilisant la methode des trapezes.
//
// Arguments:
// - f: la fonction a integrer
// - a: la borne inferieure de l'intervalle
// - b: la borne superieure de l'intervalle
// - nbIntervalles: le nombre d'intervalles a utiliser
//
//
// Resultat:
// - Doit etre disponible sur le processeur 0 uniquement.
//
////////////////////////////////////////////////////////////////////

//
```

```
double integrer( double (*f)(double),
                 double a, double b,
                 int nbIntervalles )
{
  int numProc, nbProcs;
  MPI_Comm_rank( MPI_COMM_WORLD, &numProc );
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

  // Chaque processus determine ses index inferieur et superieur.
  int inf = numProc * ( nbIntervalles / nbProcs );
  int sup = inf     + ( nbIntervalles / nbProcs );
  if ( numProc == (nbProcs-1) ) {
    sup = nbIntervalles;  // Cas special pour dernier processeur, si pas divi
  }

  // Chaque processus fait la sommation sur son sous-intervalle.
  double h = (b - a) / (double) nbIntervalles; // Taille du pas d'integration
  double sommeLocale = 0.0;
  for ( int i = inf; i < sup; i++ ) {
    double gauche = a + i * h;
    double droite = gauche + h;
    sommeLocale += ( f(gauche) + f(droite) ) * h / 2;
  }

  // On combine les resultats intermediaires.
  double sommeTotale;
  MPI_Reduce( &sommeLocale, &sommeTotale, 1, MPI_DOUBLE,
              MPI_SUM, 0, MPI_COMM_WORLD );

  return sommeTotale;
}
```

15.3 Fonction `mystere`

```
////////////////////////////////////
// Arguments:
// - elems: tableau d'entiers
// - nb: taille de elems
// - x: un entier
//
// On suppose que le tableau elems est initialement conserve sur le
// processeur 0, et on veut que la reponse finale soit connue sur le
// processeur 0.
//
```

```
////////////////////////////////////
int mystere( int elems[], int nb, int x )
{
  // On repartit les donnees.
  int nbProcs;
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

  int monNb = nb / nbProcs;
  int *mesElems = (int*) malloc( monNb * sizeof(int) );
  MPI_Scatter( elems, monNb, MPI_INT,
               mesElems, monNb, MPI_INT,
               0, MPI_COMM_WORLD );

  // On traite les donnees locales.
  int r = 0;
  for( int i = 0; i < monNb; i++ ) {
    if ( mesElems[i] == x ) { r += 1; }
  }

  // On calcule le resultat final.
  int rGlobal;
  MPI_Reduce( &r, &rGlobal, 1, MPI_INT, MPI_SUM, 0,
              MPI_COMM_WORLD );

  return rGlobal;
}
```

15.4 Programme pour simuler la diffusion de la chaleur dans un cylindre

Remarque : Les programmes qui suivent sont tirés textuellement du livre «*Patterns for Parallel Programming*» (T.G. Mattson, B.A. Sanders and B.L. Massingill, Addison-Wesley, 2005). Les seules différences sont l'ajout de quelques commentaires explicatifs et certaines petites modifications pour assurer une mise en page correcte.

15.4.1 Solution purement séquentielle

```
/** Solution sequentielle au probleme de la diffusion de la chaleur
    dans un cylindre.

  La formule utilisee pour l'equation de diffusion est differente,
  mais equivalente, a celle vue dans mon document "Resolution
  numerique de l'equation de diffusion de la chaleur dans un
  cylindre":

  NX: Nombre de points
  uk: vecteur des temperatures au temps t
  ukp1: vecteur des temperatures au temps t plus 1
  1.0: "longueur" du cylindre

  Posons:
    dx = 1.0 / NX
    dt = 0.5 * dx*dx

  Alors:

    ukp1[i]
    = uk[i] + dt / (dx*dx) * (uk[i+1] - 2 * uk[i] + uk[i-1])
    = uk[i] + (0.5 * dx*dx)/(dx*dx) * (uk[i+1] - 2 * uk[i] + uk[i-1])
    = uk[i] + 0.5 * (uk[i+1] - 2 * uk[i] + uk[i-1])
    = uk[i] + 0.5 * uk[i+1] - 0.5 * 2 * uk[i] + 0.5 * uk[i-1]
    = uk[i] + 0.5 * uk[i+1] - uk[i] + 0.5 * uk[i-1]
    = 0.5 * uk[i+1] + 0.5 * uk[i-1]
    = ( uk[i+1] + uk[i-1] ) / 2
*/
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NX (1000*16)
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize( double uk[], double ukp1[] )
{
  // On initialise la grille des temperatures courantes
  uk[0]    = LEFTVAL;
  uk[NX-1] = RIGHTVAL;
  for( int i = 1; i < NX-1; i++ ) {
    uk[i] = 0.0;
  }

  // On fait de meme pour l'autre grille
  // (au temps suivant: p1 => plus 1)
  for( int i = 0; i < NX; i++ ) {
    ukp1[i] = uk[i];
  }
}

void printValues( double uk[], int step )
{
  printf( "uk {%d}::\n", step );
  for ( int i = 0; i < NX; i++ ) {
    printf( "%6.2f ", uk[i] );
  }
  printf( "\n" );
}
```

```
int main( int argc, char *argv[] )
{
  double *uk = malloc( sizeof(double) * NX );
  double *ukp1 = malloc( sizeof(double) * NX );
  double *temp;

  double dx = 1.0 / NX;
  double dt = 0.5 * dx * dx;

  initialize( uk, ukp1 );

  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]);
    }

    // On interchange les deux grilles.
    temp = ukp1; ukp1 = uk; uk = temp;
  }

  // On affiche la grille finale.
  printValues( uk, NSTEPS );

  return( 0 );
}
```

15.4.2 Solution parallèle avec communications synchrones

```
void initialize( double uk[], double ukp1[],
                 int numPoints, int numProcs, int myID )
{
  // On initialise la grille des temperatures courantes...
  // pour les points non-extremes.
  for( int i = 1; i <= numPoints; i++ ) {
    uk[i] = 0.0;
  }

  // Cas speciaux: les deux extremites.
  if (myID == 0)           uk[1]          = LEFTVAL;
  if (myID == numProcs-1) uk[numPoints] = RIGHTVAL;

  // On fait de meme pour l'autre grille
  // (au temps suivant: p1 => plus 1)
  for( int i = 1; i <= numPoints; i++ ) {
    ukp1[i] = uk[i];
  }
}
```

```
int main( int argc, char *argv[] )
{
  double *uk, *ukp1;
  double *temp;

  double dx = 1.0 / NX;
  double dt = 0.5 * dx * dx;

  int numProcs, myID, leftNbr, rightNbr, numPoints;
  MPI_Status status;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myID );
  MPI_Comm_size( MPI_COMM_WORLD, &numProcs );

  leftNbr  = myID-1;
  rightNbr = myID+1;
  numPoints = NX / numProcs;

  uk   =  malloc( sizeof(double) * (numPoints+2) );
  ukp1 =  malloc( sizeof(double) * (numPoints+2) );

  initialize( uk, ukp1, numPoints, numProcs, myID );
```

```
for ( int k = 0; k < NSTEPS; ++k ) {
  // On obtient les donnees aux frontieres (cellules fantomes).
  if (myID != 0)
    MPI_Send( &uk[1], 1, MPI_DOUBLE, leftNbr, 0,
              MPI_COMM_WORLD );
  if (myID != numProcs-1)
    MPI_Send( &uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0,
              MPI_COMM_WORLD );

  if (myID != 0)
    MPI_Recv( &uk[0], 1, MPI_DOUBLE, leftNbr, 0,
              MPI_COMM_WORLD, &status);
  if (myID != numProcs-1)
    MPI_Recv( &uk[numPoints+1], 1, MPI_DOUBLE, rightNbr, 0,
              MPI_COMM_WORLD, &status );

  // On calcule les points intermediaires.
  for ( int i = 2; i < numPoints; ++i ) {
    ukp1[i] = uk[i] + (dt/(dx*dx)) * (uk[i+1] - 2*uk[i] + uk[i-1]);
  }

  // On traite les extremites... si necessaire.
  if (myID != 0) {
    int i = 1;
    ukp1[i] = uk[i]+(dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
  }
  if (myID != numProcs-1) {
    int i = numPoints;
    ukp1[i] = uk[i]+(dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
  }

  temp = ukp1; ukp1 = uk; uk = temp; // On interchange les grilles.
}
```

```
  MPI_Finalize();

  printValues( uk, NSTEPS, numPoints, myID );

  return( 0 );
}
```

15.4.3 Solution parallèle avec communications asynchrones

```
for ( int k = 0; k < NSTEPS; ++k ) {
  // On obtient les communications pour les donnees aux frontieres.
  if (myID != 0) {
    MPI_Irecv( &uk[0], 1, MPI_DOUBLE, leftNbr, 0, MPI_COMM_WORLD, &reqRecvL )
    MPI_Isend( &uk[1], 1, MPI_DOUBLE, leftNbr, 0, MPI_COMM_WORLD, &reqSendL )
  }
  if (myID != numProcs-1) {
    MPI_Irecv( &uk[numPoints+1], 1, MPI_DOUBLE, rightNbr, 0, MPI_COMM_WORLD,
    MPI_Isend( &uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0, MPI_COMM_WORLD, &r
  }

  // On calcule les points intermediaires.
  for ( int i = 2; i < numPoints; ++i ) {
    ukp1[i] = uk[i] + (dt/(dx*dx)) * (uk[i+1] - 2*uk[i] + uk[i-1]);
  }

  // On s'assure de la disponibilites des valeurs aux frontieres.
  if (myID != 0) {
    MPI_Wait( &reqRecvL, &status);
    MPI_Wait( &reqSendL, &status);
  }
  if (myID != numProcs-1) {
    MPI_Wait( &reqRecvR, &status);
    MPI_Wait( &reqSendR, &status);
  }

  //
```

```
  // On traite les extremites... si necessaire.
  if (myID != 0) {
    int i = 1;
    ukp1[i] = uk[i] + (dt/(dx*dx)) * (uk[i+1] - 2*uk[i] + uk[i-1]);
  }
  if (myID != numProcs-1) {
    int i = numPoints;
    ukp1[i] = uk[i] + (dt/(dx*dx)) * (uk[i+1] - 2*uk[i] + uk[i-1]);
  }

  temp = ukp1; ukp1 = uk; uk = temp; // On interchange les grilles.
}
```

Références