INF7235 — Programmation parallèle haute performance
Examen final (Hiver 2013)

Durée: (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Nom: _____

Remarques :

- Sauf pour la question 2, **vous devez répondre dans le cahier d'examen** qui vous est fourni.

  Pour la question 2, vous devez répondre directement sur le questionnaire d'examen (p. 6). **Vous devez donc remettre le questionnaire d'examen avec votre cahier.**

- Lorsqu'il est demandé de **justifier** vos réponses, tant la clarté, la justesse que la pertinence des explications seront évaluées — donc si vous écrivez des choses *fausses ou «qui n'ont pas rapport»*, vous serez pénalisés.

---

1. Manipulation de «matrices creuses» (20 pts)

On dit d'une matrice de nombres réels qu'elle est «*dense*» lorsque «la plupart» des éléments de la matrice *sont non nuls* ($\neq 0.0$). Par opposition, on dit d'une matrice de nombres réels qu'elle est «*creuse*» (en anglais : *sparse matrix*) lorsque «la plupart» des éléments de la matrice *sont nuls* ($= 0.0$).

De nombreux problèmes de calculs scientifiques requièrent la manipulation de matrices creuses. Par exemple, certaines formes d'équations différentielles peuvent être représentées à l'aide de matrices *tri-diagonales*, où seuls les éléments des trois diagonales principales sont non nuls, tous les autres éléments étant nuls.

L'extrait de code MPD 1 (p. 2) présente une ressource MPD, (très!) simplifiée, pour manipuler des matrices creuses : seulement trois (3) opérations sont exportées par la ressource — `set`, `get` et `somme` — et seule la mise en oeuvre de `somme` est présentée.

L'extrait de code MPD 2 (p. 3) présente deux cas de tests illustrant l'effet de ces opérations sur des matrices creuses.

Ce que vous devez faire

Supposons que les matrices creuses qu'on désire manipuler... sont vraiment creuses, c'est-à-dire que les matrices **sont de très grande taille** — `nbl` et `nbc` sont très grands — et que **peu d'éléments sont non nuls** — disons moins de 10 % — et que ces éléments non-nuls sont distribués de façon non-uniforme.

Supposons aussi qu'on travaille sur une machine semblable à `zeta`, donc une machine multi-processeurs comptant un petit nombre de processeurs — environ une dizaine de processeurs.

a. Décrivez et comparez différentes *stratégies parallèles* qui pourraient être utilisées pour mettre en oeuvre l'opération `somme`. Sous les conditions mentionnées plus haut, parmi ces diverses stratégies, **laquelle** selon vous devrait conduire à la mise en oeuvre la plus performante — en termes de temps d'exécution et d'accélération?

b. Écrivez, en MPD, le code de la procédure `somme`, et ce en utilisant la stratégie identifiée à l'item précédent.

   Remarque : Vous pouvez utiliser, sans les définir ou redéfinir, les procédures auxiliaires présentées dans l'extrait de code MPD 3 (p. 4).

```
resource MatriceCreuse

 # Definit l'element a la position (i,j) de la matrice creuse.
 op set( int i, int j, real x )

 # Retourne l'element a la position (i,j) de la matrice creuse.
 op get( int i, int j ) returns real x

 # Retourne la somme des elements de la matrice creuse.
 op somme() returns real x

body MatriceCreuse( int nbl /* nbLignes */, int nbc /* nbColonnes */ )

  ###################################################################
  # Representation des elements de la matrice creuse: liste chainee.
  ###################################################################
  type Liste = ptr Element;

  type Element = rec (
     int j;            # Numero de la colonne.
     real x;           # Element de la matrice.
     Liste suivant;    # Suivant de la liste chainee.
  )


  ###################################################################
  # Le tableau des listes d'elements, une liste pour chaque ligne.
  ###################################################################
  Liste elems[nbl];

  for [i = 1 to nbl] {
    # On cree une *tete de liste*, pour simplifier le traitement ulterieur.
    # La tete de liste est noeud bidon, ne contenant aucun element.
    elems[i] = new(Element);
    elems[i]^.suivant = null;
  }


  ###################################################################
  # Mise en oeuvre sequentielle de somme()
  ###################################################################

  procedure sommeSeq( ref Liste elems[*], int inf, int sup ) returns real r
  {
    r = 0.0;
    for [i = inf to sup] {
       ptr Element p = elems[i]^.suivant; # On saute la tete de liste.
       while ( p != null ) {
          r += p^.x;
          p = p^.suivant;
       }
    }
  }

  proc somme() returns r
  {
    r = sommeSeq( elems, 1, nbl );
  }
```

Extrait de code MPD 1: Constantes, types et opérations pour des matrices creuses.

```
procedure test0()
{
  nommerCasDeTest( "Test 0" );

  const int N = 100;

  cap MatriceCreuse m = create MatriceCreuse( N, N );

  for [i = 1 to N] {
    m.set( i, i, 1.0 );
  }

  for [i = 1 to N, j = 1 to N] {
    real attendu = 0.0 + int( i == j );
    assertRealEquals( attendu, m.get(i,j), PRECISION );
  }
}

procedure test3()
{
  nommerCasDeTest( "Test 3" );

  const int N = 300
  const int M = 400

  cap MatriceCreuse m = create MatriceCreuse( N, M );

  real r = 0.0
  for [i = 1 to N by 3, j = 1 to M by 2] {
    m.set( i, j, i*j )
    r += i*j
  }

  assertRealEquals( r, m.somme(), PRECISION );
}
```

Extrait de code MPD 2: Deux cas de tests pour les opérations sur des matrices creuses.

```
#####################################################################
# Variables et operations auxiliaires.
#####################################################################

external getenv(string[*]) returns string[2];

procedure getNbThreads() returns int nbThreads
{
   nbThreads = 1;
   if (getenv("MPD_PARALLEL") != "") {
    nbThreads = int(getenv("MPD_PARALLEL"));
  }
}

########################################

procedure inf( int i, int n, int nbThreads ) returns int r
{ r = (i-1) * (n / nbThreads) + 1; }

procedure sup( int i, int n, int nbThreads ) returns int r
{ r = i * (n / nbThreads); }

########################################

sem mutex = 1;

procedure FA( ref int x, int incr ) returns int r
{
  P( mutex );
  r = x;
  x += incr;
  V( mutex )
}

int tailleTache = 10;

# Variables pour representer le sac de taches.
int suivant, borneSup;

# Initialisation du sac de taches.
procedure initSacDeTaches( int bi, int bs )
{ suivant = bi; borneSup = bs;  }

# Retrait d'une tache du sac.
procedure obtenirTache( res int i, res int j ) returns bool disponible
{
  i = FA( suivant, tailleTache );
  j = min( i + tailleTache - 1, borneSup );
  disponible = (i <= borneSup);
}
```

Extrait de code MPD 3: Procédures auxiliaires pour la mise en oeuvre des solutions parallèles sur les matrices creuses.

2. Manipulation de polynômes en OpenMP/C (10 pts)

On veut définir, en OpenMP/C, des opérations pour des polynômes. La signature des opérations du fichier `polynomes.h` et certaines macros et fonctions du fichier `polynomes.c` sont présentées dans l'extrait de code C 1.

```
////////////////////////////////////////////
// Fichier polynomes.h
////////////////////////////////////////////
typedef struct {
  int degre;
  int* coefficients;  // Tableau alloue dynamiquement.
} Polynome;

Polynome polynome( int degre, int coefficients[] ); // Constructeur de base.

Polynome fois( Polynome p1, Polynome p2 ); // Produit.

bool egaux( Polynome p1, Polynome p2 );    // Comparaison.
```

```
////////////////////////////////////////////
// Fichier polynomes.c (extraits)
////////////////////////////////////////////
#include "polynomes.h"

// Macros et fonctions auxiliaires.
#define MIN( x, y ) ((x) < (y) ? (x) : (y))
#define MAX( x, y ) ((x) > (y) ? (x) : (y))

static Polynome allouer( int degre )
{
  Polynome p;
  p.degre = degre;
  p.coefficients = (int*) malloc( (degre+1) * sizeof(int) );
  return( p );
}
```

Extrait de code C 1: Fichier `polynomes.h` et extraits du fichier `polynomes.c`.

Complétez la mise en oeuvre des opérations présentées à la page suivante de façon **à les rendre les plus parallèles et efficaces possible.**

Pour la distribution des itérations entre les *threads*, vous devez spécifier les valeurs pour `schedule` qui vous semblent les plus appropriées. Indiquez aussi (très brièvement) pourquoi vous avez choisi d'utiliser cette valeur particulière pour `schedule`.

Rappel : La syntaxe pour `schedule` est la suivante, où `chunk` ≥ 1 (entier, optionnel) :

```
#omp for ... schedule( static [,chunk] )
#omp for ... schedule( dynamic [,chunk] )
#omp for ... schedule( guided [,chunk] )
#omp for ... schedule( runtime )
```

Vous pouvez aussi supposer que la variable d'environnement `OMP_NESTED` est égale à `TRUE`.

```
Polynome polynome( int degre, int coefficients[] )
{
  Polynome p = allouer(degre);



  for( int i = 0; i <= degre; i++ ) {

    p.coefficients[i] = coefficients[i];

  }

  return( p );
}


static int coefficient( int i, Polynome p1, Polynome p2 )
{
  int expMinR1 = MAX( 0, i-p2.degre );
  int expMaxR1 = MIN( i, p1.degre );

  int c = 0;



  for( int k = expMinR1; k <= expMaxR1; k++ ) {

    c += p1.coefficients[k] * p2.coefficients[i-k];

  }

  return( c );
}


Polynome fois( Polynome p1, Polynome p2 )
{
  Polynome p = allouer( p1.degre + p2.degre );




  for( int i = 0; i <= p.degre; i++ ) {

    p.coefficients[i] = coefficient( i, p1, p2 );

  }

  return( p );
}
```

3. Mesures de performance et loi d'Amdhal (10 pts)

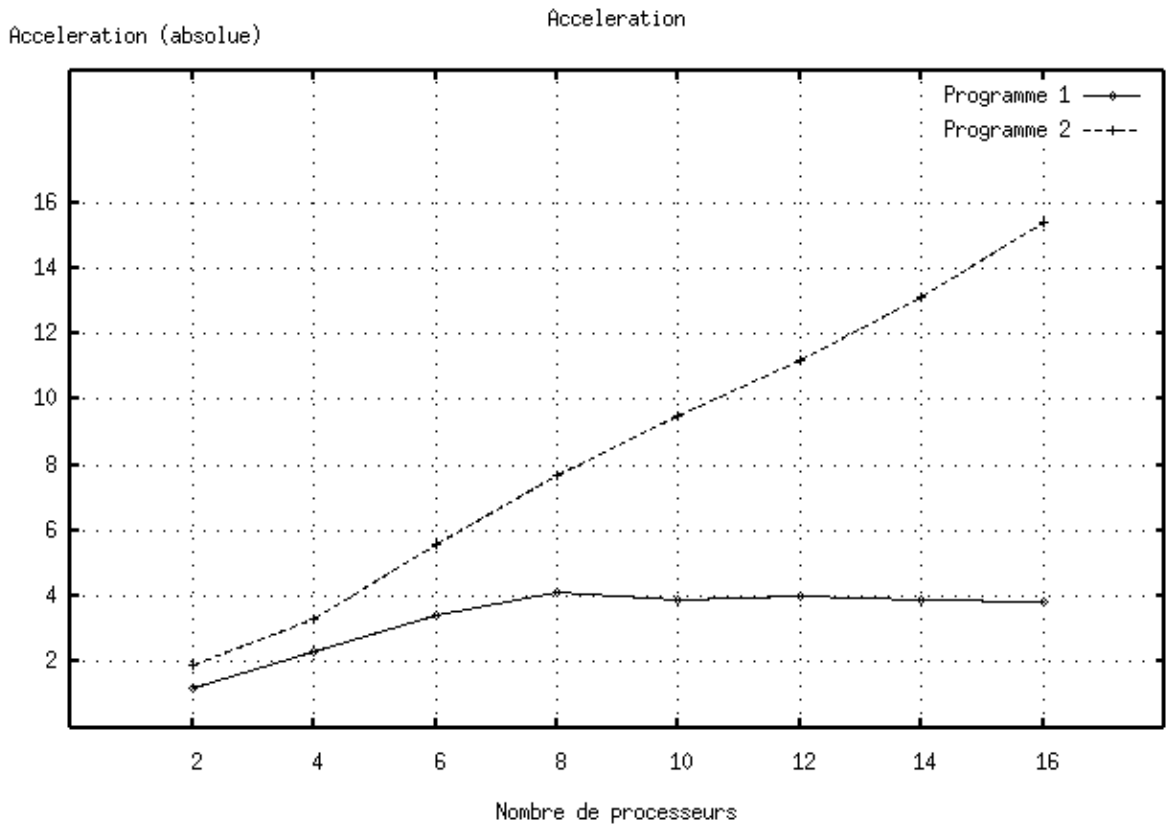

Figure 1: Graphes d'accélération pour deux programmes exécutés sur une machine multi-processeurs.

La figure 1 présente les graphes d'accélération pour deux programmes traitant le même problème — `Programme 1` (ligne continue) et `Programme 2` (ligne pointillée) — s'exécutant sur une machine multi-processeurs avec un langage de programmation parallèle avec variables partageés (à la MPD).

Ces graphes d'accélérations montrent, pour un problème de grande taille, l'effet du nombre de processeurs (*physiques*) sur l'accélération absolue obtenue.

Répondez alors aux questions ci-bas. Lorsque des valeurs numériques sont demandées, une réponse approximative est suffisante (à 5-10 % près), donc pas besoin de faire des calculs exacts et précis.

a. Quelle est la meilleure *efficacité* obtenue avec le Programme 1?

b. Quelle est la meilleure *efficacité* obtenue avec le Programme 2?

c. Pour le Programme 1, à combien peut-on estimer la fraction f du programme qui doit être exécutée **de façon purement séquentielle**?

d. Que peut-on prédire quant à l'accélération obtenue pour le Programme 1 si on l'exécutait avec 32 processeurs?

e. Que peut-on prédire quant à l'accélération obtenue pour le Programme 2 si on l'exécutait avec 32 processeurs?

Rappel sur la loi d'Amdahl : Soit f la fraction des opérations qui doivent être exécutées de façon séquentielle, avec $0 \leq f \leq 1$. Alors, l'accélération A pouvant être obtenue avec p processeurs est bornée comme suit :

$$A \leq \frac{1}{f + (1-f)/p}$$

4. Construction d'un histogramme (10 pts)

On veut écrire un programme MPI/C pour produire un histogramme. Les données à traiter, une série de nombres entiers, sont initialement connues uniquement du processus 0.

Plus précisément, chaque nombre à traiter est un entier non-négatif (donc possiblement nul). On ne connaît pas la valeur maximale de ces nombres ; on sait seulement que l'on peut allouer sans problème un tableau *dynamique* de taille appropriée pour représenter l'intervalle des différentes valeurs. Le nombre d'occurrences du nombre `k` dans l'histogramme sera alors donné par l'item à la position `k` du tableau.

Par exemple, supposons qu'on ait les 12 valeurs suivantes :

```
elems = {10, 1, 3, 3, 3, 2, 9, 1, 1, 1, 3, 10}
```

L'histogramme résultant serait alors le suivant :

```
histo = {0, 4, 1, 4, 0, 0, 0, 0, 0, 1, 2}
```

Le programme C 1 (p. 9) présente une solution séquentielle pour ce problème.

Complétez le programme MPI 1 (p. 10) pour réaliser cette tâche de façon parallèle en MPI. On suppose qu'on utilise une machine semblable à `trex_cluster`. On suppose aussi que seul le processus 0 a besoin de connaître l'histogramme final.

Remarque : Lorsque c'est possible, utilisez des opérations *collectives de communication*.

```
/*
   Programme pour calcul d'un histogramme.

   Donnee d'entree (sur la ligne de commande):
   - Un nombre entier positif

   Sortie (stdout)
   - L'histogramme est imprime
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX( x, y ) ((x) > (y) ? (x) : (y))

static void genererElems( int elems[], int nb )
{
  // Genere une serie de nombres (dans elems) qui seront ensuite analyses
  // pour la production de l'histogramme.
  ...
}

int main( int argc, char *argv[] )
{
  // On alloue l'espace pour les elements et on les genere.
  assert( argc > 1 );
  int nb = atoi( argv[1] );

  int *elems;
  elems = (int*) malloc( nb * sizeof(int) );
  genererElems( elems, nb );

  // On trouve la valeur maximale pour definir les bornes de l'histogramme.
  int valMax = 0;
  for( int i = 0; i < nb; i++ ) {
    valMax = MAX( valMax, elems[i] );
  }

  // On initialise l'histogramme.
  int *histo = (int*) malloc( (valMax+1) * sizeof(int) );
  for( int i = 0; i <= valMax; i++ ) {
    histo[i] = 0;
  }

  // On construit l'histogramme.
  for( int i = 0; i < nb; i++ ) {
    histo[elems[i]] += 1;
  }

  // On imprime les resultats.
  for( int i = 0; i <= valMax; i++ ) {
    printf( "histo[%d] = %d\n", i, histo[i] );
  }
  return( 0 );
}
```

Programme C 1: Programme séquentiel en C pour le calcul d'un histogramme.

```
int main( int argc, char *argv[] )
{
  int nbProcs;                // Nombre de processus.
  int numProc;                // Numero du processus courant.

  // On initialise MPI.
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &numProc );
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );


  // On alloue l'espace pour les elements et on les genere.
  assert( argc > 1 );
  int nb = atoi( argv[1] );
  assert( nb % nbProcs == 0 );    // Pour simplifier le probleme!

  int *elems;
  if ( numProc == 0 ) {
    elems = (int*) malloc( nb * sizeof(int) );
    genererElems( elems, nb );
  }

  // A COMPLETER... dans le cahier d'examen.









  // On imprime les resultats.
  if ( numProc == 0 ) {
    for( int i = 0; i <= valMax; i++ ) {
      printf( "histo[%d] = %d\n", i, histo[i] );
    }
  }

  MPI_Finalize();
  return( 0 );
}
```

Programme MPI 1: Programme MPI à compléter pour le calcul d'un histogramme.

5. Solution sudoku généralisé d'ordre k (10 pts)

Soit k un entier positif. Nous dirons d'une matrice $k^2 \times k^2$ qu'elle est une «solution sudoku généralisé d'ordre k» si les conditions suivantes sont satisfaites :

a. Chaque ligne de la matrice contient tous les nombres de 1 à k^2.

b. Chaque colonne de la matrice contient tous les nombres de 1 à k^2.

c. Lorsqu'on décompose la matrice en k^2 sous-matrices indépendantes de taille $k \times k$, chaque sous-matrice contient tous les nombres de 1 à k^2.

La figure 2 présente une matrice 9×9 qui est une solution sudoku généralisé d'ordre 3 — les tirets ne font pas partie de la matrice : ils ne servent qu'à mieux illustrer la troisième condition.

```
 5 3 4 | 6 7 8 | 9 1 2
 6 7 2 | 1 9 5 | 3 4 8
 1 9 8 | 3 4 2 | 5 6 7
-------+-------+------
 8 5 9 | 7 6 1 | 4 2 3
 4 2 6 | 8 5 3 | 7 9 1
 7 1 3 | 9 2 4 | 8 5 6
-------+-------+------
 9 6 1 | 5 3 7 | 2 8 4
 2 8 7 | 4 1 9 | 6 3 5
 3 4 5 | 2 8 6 | 1 7 9
```

Figure 2: Matrice 9×9 qui est une solution sudoku généralisé d'ordre 3.

Le problème et les hypothèses

On travaille sur une machine parallèle *à mémoire distribuée* — communication par échange de messages avec MPI — qui comporte p processeurs.

Soit une matrice $A : n \times n$, tel que n est un carré parfait — donc il existe k tel que $k^2 = n$. La valeur de n peut être très grande, donc n est possiblement supérieur à p de plusieurs ordres de grandeur — exprimé autrement : $p << n$.

On suppose que la matrice A est déjà distribuée entre les processeurs, et ce de façon exclusive et indépendante — en d'autres mots, il n'y a pas de chevauchement entre les éléments des différents morceaux.

On désire écrire une procédure qui va déterminer — donc vrai ou faux? — si la matrice A est une solution sudoku généralisé d'ordre k. De plus, ce résultat booléen devra être connu de l'ensemble des processus.

Ce que vous devez faire

Comparez (avantages et désavantages, forces et faiblesses) différentes façons de résoudre ce problème.

Notamment, analysez l'impact du choix de distribution de la matrice A sur le fonctionnement et l'efficacité de la procédure qui détermine si les éléments de A forment ou non une solution sudoku généralisé d'ordre k.

Si vous devez poser certaines hypothèses pour simplifier votre analyse, indiquez-les explicitement.

Concluez en indiquant *l'approche que vous suggéreriez d'utiliser, de façon à bien exploiter les ressources de la machine.*

Remarque : Vous n'avez pas à écrire de code MPI/C. Vous devez simplement décrire *les grandes lignes de vos approches*, de façon suffisante pour qu'on ait une bonne idée de la stratégie utilisée *et des communications entre processeurs qui sont requises.*