INF7235 — Programmation parallèle haute performance
Examen final (Hiver 2016)

Durée: Trois (3) heures
Documentation : Documentation personnelle autorisée (y compris *laptop*, en mode «lecture»!).

Nom: _____

Code permanent: | | | | | | | | | | | | |

1	2	3	4	5	6	Total
/10	/10	/5	/10	/10	/ 10	/55

- **Vous devez répondre directement sur le questionnaire d'examen**.
- 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 pourriez être pénalisés.
- Si nécessaire, vous pouvez utiliser, **sans la définir**, la méthode Ruby suivante :

```ruby
# Les indices pour la tranche du thread no. k
# lorsqu'on repartit uniformement n elements entre nb_threads threads.
def indices_tranche( k, n, nb_threads )
  b_inf = k * n / nb_threads
  b_sup = (k + 1) * n / nb_threads - 1

  b_inf..b_sup  # On retourne un Range des indices de la tranche.
end
```

- La déclaration «`function<bool (int)> pred`» dans l'exemple C++ (p. 8) indique un `predicat` sur un `int` — donc une fonction qui reçoit un argument `int` et qui retourne `true` **si l'argument satisfait une certaine condition décrite par le prédicat**, `false` sinon.

  Idem pour le type `Predicat` du programme OpenMP/C de la page 5.
- Concernant le type `vector<int>` en C++ (p. 8) :
  - Un objet `v` de type `vector<int>` dénote un vecteur d'entiers de longueur variable — donc semblable à un `Array` en Ruby.
  - `v.size()` = Nombre d'éléments dans le vecteur.
  - `vector<int>()` = vecteur vide — donc `vector<int>().size() == 0`.
  - `v.push_back(x)` : ajoute l'élément `x` **à la fin** du vecteur `v`.

1. Produit de polynomes en PRuby (10 pts)

Une classe `Polynome`, en Ruby, est présentée à la page 4.

Pour les deux sous-questions qui suivent, vous devez utiliser les méthodes de la bibliothèque `PRuby`.

[3] a) Donnez une mise en oeuvre **parallèle** de la méthode «==», solution parallèle **la plus simple possible** en PRuby.

```
def ==( autre )




end
```

[7] b) Donnez, à la page suivante, deux (2) mises en oeuvre parallèles de la méthode «*».

i. Pour la première mise en oeuvre, vous devez utiliser du parallélismte **itératif** de type **fork-join** (donc **pas** du parallélisme récursif!) et ce avec une répartition **statique à granularité grossière** des tâches.

ii. Pour la deuxième mise en oeuvre, vous pouvez utiliser la stratégie de votre choix, mais toujours **à granularité grossière**. Vous devez indiquer, dans le commentaire, quelle stratégie — quel **patron de programmation parallèle** — vous avez choisi d'utiliser.

iii. Indiquez (ci-bas) quelle mise en oeuvre, selon vous, serait la meilleure et expliquez brièvement pourquoi il en serait ainsi.

Dans les deux cas, pour simplifier, vous pouvez poser des hypothèses appropriées sur le nombre de *threads* et/ou les tailles de polynomes, en indiquant explicitement les conditions requises à l'aide d'appels à `assert`.

- Meilleure approche =
- Explication/justification =

```
# Repartition statique a granularite grossiere avec parallelisme fork-join.
def *( autre )









end
```

```
# Approche utilisee =
#
def *( autre )









end
```

```ruby
class Polynome
  def initialize( *coeffs )
    assert coeffs.size >= 1, "*** Il doit y avoir au moins un coefficient"

    coeffs.pop while coeffs.size > 1 && coeffs.last == 0
    @coeffs = *coeffs
  end

  def taille
    @coeffs.size
  end

  def []( k )
    assert 0 <= k && k < taille, "*** k = #{k} vs. taille = #{taille}"

    @coeffs[k]
  end

  def ==( autre )
    return false if taille != autre.taille

    (0...taille).all? { |i| self[i] == autre[i] }
  end

  def +( autre )
    return autre + self if taille > autre.taille

    coeffs = (0...autre.taille).map do |k|
      (k < taille ? self[k] : 0) + autre[k]
    end

    Polynome.new( *coeffs )
  end

  #
  # Calcul du k ieme coefficient pour le produit de p1 et p2.
  #
  def coefficient( k, p1, p2 )
    exp_min = [ 0, k-p2.taille+1 ].max
    exp_max = [ k, p1.taille-1 ].min

    (exp_min..exp_max).
      preduce(0) { |somme, i| somme + p1[i] * p2[k-i] }
  end
end
```

2. Fonctions `index_of` et `count` (OpenMP/C) (10 pts)

[6] a) On veut réaliser, en OpenMP/C, une fonction `index_of`. Cette fonction reçoit en arguments un tableau, la taille du tableau et un **prédicat**. La fonction retourne alors **un index tel que l'élément à cet index satisfait le prédicat**. Si aucun élément ne satisfait le prédicat, la fonction retourne `-1` ; si plusieurs éléments satisfont le prédicat, un des index est retourné, **au choix de l'implémenteur**.

En Ruby, Java, ou C++, le prédicat serait représenté par une lambda-expression. En C, le prédicat est plutôt représenté par un pointeur vers une fonction : voir plus bas.

Squelette de la fonction `index_of`
La définition du type `Predicat` et le corps de la fonction `index_of` sont comme suit :

```
typedef int (*Predicat)(int);

int index_of( int elems[], int n, Predicat pred )
{
  int pos = -1;

  ... segment de code indiqué à la page suivante ...

  return pos;
}
```

Exemples d'utilisation

```
int estZero( int x ) { return x == 0; }

int elems1[7] = {1, 2, 0, 3, 4, 5, 6};
assert( index_of(elems1, 7, estZero) == 2 );

int elems2[7] = {1, 2, 8, 3, 4, 5, 6};
assert( index_of(elems2, 7, estZero) == -1 );
```

Ce qu'il faut faire
Pour chacun des segments de code de la page suivante, **indiquez si la fonction produit ou non le bon résultat**. Même si la réponse produite est bonne, **indiquez s'il s'agit ou non d'une bonne stratégie, en justifiant brièvement votre réponse**. Finalement, indiquez aussi **laquelle parmi ces trois solutions vous semble la meilleure**.

i.
```
#pragma omp parallel
for( int i = 0; i < n; i++ ) {
  if ( (*pred)(elems[i]) ) {
    pos = i;
  }
}
```

ii.
```
#pragma omp parallel for
for( int i = 0; i < n; i++ ) {
  if ( (*pred)(elems[i]) ) {
    #pragma omp critical
    pos = i;
  }
}
```

iii.
```
#pragma omp parallel for
for( int i = 0; i < n; i++ ) {
  if ( (*pred)(elems[i]) ) {
    pos = i;
  }
}
```

[4] b) Soit le fragment de code suivant, qui définit une fonction qui **compte le nombre** d'eléments de `elems` qui satisfont le prédicat `pred`.

```
int count( int elems[], int n, Predicat pred )
{
  int nb = 0;


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

    if ( (*pred)(elems[i]) ) {

      nb += 1;

    }

  }

  return nb;
}
```

Supposons que le temps d'exécution d'un appel à `pred` soit **très variable** d'un élément à un autre — parfois court, parfois long.

Quelle(s) directive(s) OpenMP faudrait-il ajouter pour **paralléliser** cette fonction et obtenir un temps d'exécution qui soit le meilleur possible? Indiquez les directives à ajouter directement dans le corps de la fonction.

Note : vous devez **aussi** indiquer **explicitement** (mais pas uniquement!) la stratégie de répartition des itérations qu'il serait préférable d'utiliser — `static`, `dynamic`, `guided`, `runtime`? avec ou sans argument?

Note : Il n'est pas possible, à l'intérieur d'une boucle parallèle, d'exécuter un `return` ou `break`. (Le `return` indiqué est à l'extérieur de la boucle!)

3. Programme mystère avec réduction parallèle (TBB/C++) (5 pts)

```
// Note: Voir page 1 pour le type vector<int>.
static vector<int> foo( vector<int> elems, function<bool (int)> pred ) {
  return parallel_reduce(
      blocked_range<size_t>(0, elems.size()),

      vector<int>(), // Vecteur vide.

      [=]( blocked_range<size_t> r, vector<int> res ) {
        for( size_t k = r.begin(); k < r.end(); k++ ) {
          if( pred(elems[k]) ) {
            res.push_back( elems[k] ); // On ajoute a la fin.
          }
        }
        return res;
      },

      [=]( vector<int> r1, vector<int> r2 ) {
        for( int i = 0; i < r2.size(); i++ ) {
          r1.push_back( r2[i] ); // On ajoute a la fin.
        }
        return r1;
      }
  );
}
```

i. Qu'est-ce qui sera retourné dans la variable `r` par l'appel suivant :

```
// On suppose elems = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 }
vector<int> r = foo( elems, []( int x ) { return x % 3 == 0; } );
```

ii. De façon plus générale, que fait la fonction `foo`? Quel nom **plus significatif** pourrait-on lui donner?

4. Mesures de performances (10 pts)

[8] a) Soit une machine parallèle *à mémoire partagée* comportant 16 processeurs. On a mesuré le temps requis par un **programme parallèle** P_1 pour traiter un même ensemble de données avec différents nombres de processeurs (1, 2, 4, 8) et on a obtenu les temps suivants (en ms) :

Nb. procs	**Temps**
1	150
2	60
4	40
8	50

Le temps requis par une version **séquentielle** du programme avec les mêmes données est de 100 ms.

i. Quelle est **l'accélération relative** lorsqu'on utilise 4 processeurs?

ii. Quelle est **l'accélération absolue** lorsqu'on utilise 4 processeurs?

iii. En utilisant l'accélération **absolue**, pour quel nombre de processeurs obtient-on la meilleure **efficacité** et quelle est cette efficacité — exprimée en **pourcentage**?

iv. Quel sera le comportement du programme (accélération et efficacité) si on l'exécute sur les mêmes données mais en utilisant les 16 processeurs? Expliquez brièvement pourquoi le programme aurait ce comportement.

[2] b) Pour un problème d'une certaine taille, on a déterminé que 5 % des instructions d'un programme P_2 étaient des instructions devant être exécutées séquentiellement par un unique processeur. Par contre, tout le reste peut s'exécuter **en parallèle** (problème *embarassingly parallel*). Avec **quel nombre de processeurs** peut-on espérer obtenir une accélération **supérieure ou égale à 10** en traitant les mêmes données (c'est-à-dire en gardant la même taille de problème)?

5. SAXPY en MPI/C (mémoire distribuée) (10 pts)

Un calcul fréquemment rencontré lorsqu'on manipule des matrices de nombres points-flottants est celui du calcul *SAXPY* — «*Single-Precision A · X Plus Y*».

Plus spécifiquement, un calcul *SAXPY* reçoit en entrée un scalaire `a`, deux vecteurs `x` et `y` de même taille `n`, puis **multiplie** chaque élément `x[i]` par `a` et **ajoute** le résultat à `y[i]`. Une mise en oeuvre séquentielle en C de `saxpy` a donc l'allure suivante :

```
void saxpy( float a, float x[], float y[], int n )
{
  for ( int i = 0; i < n; i++ ) {
      y[i] += a*x[i];
  }
}
```

La procédure à la page suivante présente un squelette pour une version parallèle en MPI/C du calcul `saxpy`, procédure à laquelle deux arguments additionnels ont été ajoutés :

- Le processus `racine`, qui a initialement le scalaire `a`, les vecteurs `x` et `y` et la taille `n`. C'est aussi ce processus **vers lequel le résultat final `y` doit ensuite être retourné** ;
- Le communicateur qui doit être utilisé pour les communications durant l'exécution de `saxpy`.

Un appel de cette procédure aurait donc l'allure suivante :

```
//  Exemple d'appel

// On suppose ici que tous les processus de MPI_COMM_WORLD seront utilisés.
// On crée un communicateur clone pour ne pas interférer avec d'autres appels.
MPI_Comm comm;
MPI_Comm_dup( MPI_COMM_WORLD, &comm );

// Le scalaire a et les vecteurs x et y sont uniquement sur le processus 0.
saxpy( a, x, y, n, 0, comm );
// Le vecteur résultant y est maintenant sur le processus 0.

MPI_Comm_free( &comm );
```

Complétez la procédure `saxpy` à la page suivante. On suppose qu'on utilise une machine semblable au *cluster* utilisé durant les labos, donc un petit *cluster* comptant une trentaine de noeuds. On suppose aussi que `n` peut être **très grand!**

Vous pouvez poser des hypothèses appropriées sur le nombre de *processus* et/ou la valeur de `n`, et ce en indiquant explicitement ces hypothèses l'aide d'appels à `assert`.

```
//
// Le scalaire a et les vecteurs x et y sont uniquement sur le processus racine.
// Le vecteur final résultant devra être sur le processus racine.
//
void saxpy( float a, float x[], float y[], int n, int racine, MPI_Comm comm )
{









}
```

6. Parallélisation d'un problème de distance d'édition (10 pts)

Le problème

La section 7.1 des notes de cours présente un algorithme pour le calcul de la distance d'édition entre deux chaines ainsi que l'analyse des dépendances entre les calculs — donnant lieu à un calcul de type *wavefront*. (Pour rappel, l'algorithme **séquentiel** et les figures résultants de **l'analyse des dépendances** sont présentés à la page suivante.)

On veut développer un programme parallèle qui analysera, avec la distance d'édition, **un grand nombre** de **très longues chaines**. Les données et résultats seront comme suit :

- Argument : Une série de noms de fichier. Chaque fichier contient du texte, interprété comme une unique (possiblement très longue) chaine de caractères.
- Résultat : La distance d'édition entre le contenu **de chaque fichier** — **sans répétition**.

Par exemple, voici «l'allure» du résultat d'une exécution de ce programme sur dix fichiers `f[0-9].txt` :

```
$ mpirun -np 30 distances f0.txt f1.txt f2.txt ... f9.txt
distance(f0.txt, f1.txt) = 92304
distance(f0.txt, f2.txt) = 12320
...
distance(f0.txt, f9.txt) = 9975

distance(f1.txt, f2.txt) = 1292
...
distance(f1.txt, f9.txt) = 9924

...

distance(f7.txt, f8.txt) = 8233
distance(f7.txt, f9.txt) = 786

distance(f8.txt, f9.txt) = 3786
```

La machine et le modèle de programmation

Le programme à développer s'exécutera sur un petit *cluster*. La machine compte une trentaine de noeuds, connectés par un réseau dédié à haute vitesse. Chaque noeud est une machine **multicoeurs** comptant de 2 à 4 coeurs et se partageant l'accès à une mémoire.

Le modèle de programmation utilisé est un modèle **hybride** :

- Pour la répartition du travail **entre les noeuds**, on utilise une approche SPMD avec MPI/C.
- Pour la répartition du travail **à l'intérieur d'un noeud**, on utilise OpenMP/C.

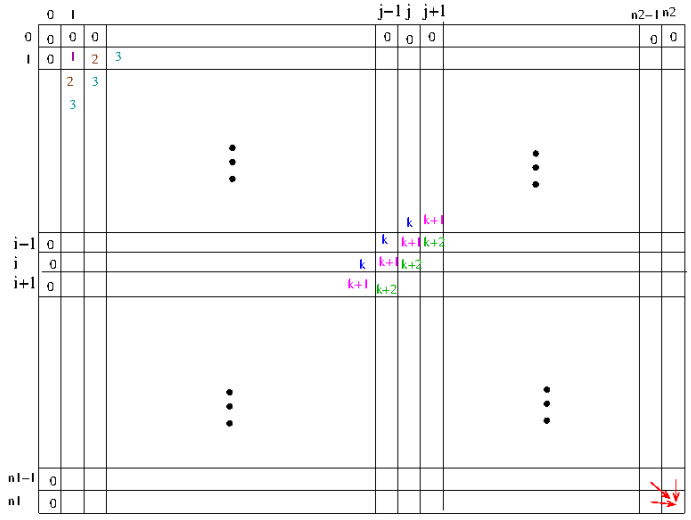
Ce que vous devez faire

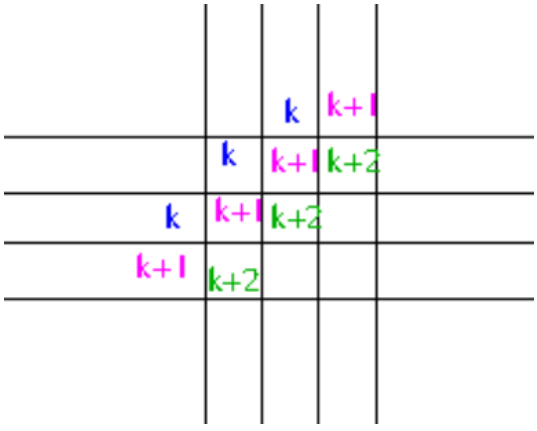
Décrivez et comparez (avantages et désavantages, forces et faiblesses) diverses façons (au moins 2!) de concevoir un programme parallèle pour résoudre ce problème.

Vous n'avez pas à écrire de code. Vous devez uniquement décrire **les grandes lignes des approches proposées** — en termes d'identification des tâches, distribution des données, communications, etc.

Vous devez aussi conclure en indiquant **l'approche que vous suggéreriez d'utiliser, de façon à bien exploiter les ressources de la machine** — autant les noeuds distribués que les processeurs multicoeurs.

```ruby
# Note: Exemple de la methode "*" sur des Ranges
#   (1..3)*(2..3) = [[1,2], [1,3], [2,2], [2,3], [3,2], [3,3]]

def distance_seq( ch1, ch2 )
  n1 = ch1.size
  n2 = ch2.size
  d = Matrice.new( n1+1, n2+1 )

  # Cas de base.
  d[0,0] = 0
  (1..n1).each do |i|
    d[i, 0] = i
  end
  (1..n2).each do |j|
    d[0, j] = j
  end

  # Cas recursifs.
  ((1..n1)*(1..n2)).each do |i, j|
    d[i, j] = [ d[i-1, j] + 1,
                d[i, j-1] + 1,
                d[i-1, j-1] + cout_subst( ch1[i], ch2[j] )
              ].min
  end

  d[n1, n2]
end
```





Note: Pour cette question, vous pouvez écrire au verso si nécessaire, ou utilisez les feuilles lignées mises à votre disposition — numérotez chacune des feuilles additionnelles et indiquez votre nom!