INF7235 — Programmation parallèle haute performance
Examen final (Hiver 2017)

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

Nom: ______________________________

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

1	2	3	4	5	6	Total
/10	/10	/10	/10	/10	/ 10	/60

- **Répondez directement sur le questionnaire**.
- 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.
- La méthode `Pruby` suivante est utilisée dans une des questions :

```
# Les bornes pour la tranche du thread no. k, lorsqu'on répartit uniformément
# n éléments entre nb_threads threads.
def bornes_de_tranche( k, n, nb_threads )
  b_inf = k * n / nb_threads
  b_sup = (k + 1) * n / nb_threads - 1

  b_inf..b_sup  # Un Range inclusif!
end
```

- En C, la déclaration suivante dénote un opérateur binaire sur des `int` — donc (un pointeur vers) une fonction qui reçoit deux arguments `int` et qui retourne un résultat `int` :

```
typedef int (*BinOp)(int, int);
```

Une fonction peut alors recevoir un argument de ce type — l'équivalent d'une *lambda*-expression :

```
int plus( int x, int y )
{ return( x + y ); }

int fois( int x, int y )
{ return( x * y ); }

int foo( BinOp op, int x )
{ return( (*op)(x, x) ); }  // Appel de l'operateur binaire op.

assert( foo(fois, 10) == 100 );
assert( foo(plus, 10) == 20  );
```

1. Mesures de performances (10 pts)

Soit une machine parallèle *à mémoire distribuée* avec 32 processeurs (un petit *cluster*). On a mesuré le temps requis par un **programme parallèle** pour traiter un même ensemble de données avec différents nombres de processeurs (1, 2, 4, 8, 16) et on a obtenu les temps ci-bas (en *ms*), alors que le temps d'exécution pour une version **séquentielle** de ce programme sur ces données est de 100 *ms*.

Nb. procs	Temps
1	120
2	80
4	30
8	20
16	12

i. Quelle est **l'accélération absolue** lorsqu'on utilise 8 processeurs?

ii. Quelle est **l'accélération relative** lorsqu'on utilise 8 processeurs?

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

iv. Peut-on prédire quel sera le comportement du programme — accélération et efficacité — si on l'exécute sur les mêmes données mais en utilisant cette fois les 32 processeurs de la machine? Expliquez brièvement pourquoi le programme aurait ce comportement.

v. Peut-on prédire quel sera le comportement du programme — accélération et efficacité — si on l'exécute avec différents nombres de processeurs (1, 2, 4, ..., 32) mais cette fois en augmentant **la taille de l'ensemble des données à traiter** — par exemple, en doublant la quantité de données à traiter? Expliquez brièvement pourquoi le programme aurait ce comportement.

2. Méthode `mystere` de la classe `Array` (10 pts)

Soit la méthode `mystere` suivante définie **dans la classe** `Array` (`bornes_de_tranche` : cf. p. 1) :

```
class Array
  def mystere
    nb_threads = PRuby.nb_threads

    futures = (0...nb_threads).map do |num_thr|
      PRuby.future do
        nb = 0
        bornes_de_tranche(num_thr, size, nb_threads).each do |k|
          nb += 1 if yield( self[k] )
        end

        nb
      end
    end

    futures.map(&:value).reduce(0, :+)
  end
end
```

---

i. Complétez les cas de test suivants pour qu'ils s'exécutent avec succès — c'est-à-dire indiquez la valeur attendue après `must_equal` pour que le test soit correct :

```
[10, 20, 30, 40].mystere { |x| x }.must_equal
```

```
[10, 20, 30, 40].mystere { |x| (x / 10).even? }.must_equal
```

```
[*1..100].mystere { |x| x <= 20 }.must_equal
```

ii. De façon plus **générale**, que fait cette méthode? Quel nom **plus significatif** peut-on lui donner?

iii. Est-ce qu'une version de la méthode `mystere` utilisant du **parallélisme de boucles** (`peach` et/ou `peach_index`) serait une bonne approche pour ce problème? Justifiez brièvement.

iv. Complétez `mystere_rec_ij` ci-bas pour que `mystere` soit réalisée avec du **parallélisme *fork-join* récursif dichotomique**. Notez que `mystere_rec_ij` n'utilise pas de *seuil* de récursion : la **récursion parallèle** se poursuit donc jusqu'au cas de base trivial (le plus simple possible).

```
# Autre mise en oeuvre de mystere: parallelisme recursif
def mystere
  return 0 if empty?
  mystere_rec_ij( 0, size - 1 ) { |x| yield( x ) }
end

def mystere_rec_ij( i, j )



end
```

3. Fonctions sur des polynomes (TBB/C++) (10 pts)

Soit $p(x)$ un polynôme de degré n :

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_{n-1}x^{n-1} + a_nx^n$$

- La `dérivée` $p'(x)$ de $p(x)$ par rapport à x est le polynôme suivant :

  $$p'(x) = a_1 + 2a_2x + 3a_3x^2 + \ldots + (n-1)a_{n-1}x^{n-2} + na_nx^{n-1}$$

  Dans le cas spécial où $p(x) = a_0$ (une constante), $p'(x) = 0$.

- Quant à la `valeur` du polynome p au point v, c'est le résultat obtenu en **évaluant** p en v :

  $$p(v) = a_0 + a_1v + a_2v^2 + a_3v^3 + \ldots + a_{n-1}v^{n-1} + a_nv^n$$

Complétez (page suivante) les fonctions `dérivée` et `valeur` pour les rendre les plus parallèles et efficaces possible, et ce **en utilisant les constructions** `parallel_for` et/ou `parallel_reduce` **avec** `blocked_range` de TBB/C++.

Note : La valeur v^k peut être calculée en utilisant la fonction standard suivante, où `v` est un `double` et `k` un entier : `std::pow(v, k)`

Rappel : Voici la représentation des `Polynomes`, ainsi que la version séquentielle de `plus` ;

```
// Representation interne = degre + tableau des coefficients.
typedef struct {
  int degre;
  double* coefficients;
} Polynome;

Polynome plus( Polynome p1, Polynome p2 )
{
  ordonnerPetitGrand( p1, p2 );
  assert( p1.degre <= p2.degre );

  Polynome p = allouer( p2.degre );
  for( int i = 0; i <= p1.degre; i++ ) {
    p.coefficients[i] = p1.coefficients[i] + p2.coefficients[i];
  }
  for( int i = p1.degre+1; i <= p2.degre; i++ ) {
    p.coefficients[i] = p2.coefficients[i];
  }

  return p;
}
```

```
// a.
Polynome dérivée( Polynome p ) {
  if ( p.degre == 1 ) {
    Polynome pPrime = allouer( 1 );
    pPrime.coefficients[0] = 0;
    return pPrime;
  }









}
```

```
// b.
double valeur( Polynome p, double v ) {









}
```

4. Fonctions sur des vecteurs (OpenMP/C) (10 pts)

Des opérations souvent utilisées lorsqu'on manipule des vecteurs de flottants sont le *SAXPY* (*Single-precision A · X Plus Y*) et le produit scalaire : `saxpy` reçoit un scalaire `a` et deux vecteurs `x` et `y` (de taille `n`), **multiplie** chaque élément `x[i]` par `a` puis **ajoute** le résultat à `y[i]` ; `produit_scalaire` est utilisé, notamment, pour combiner une ligne (`x`) et une colonne (`y`) lors d'un produit matriciel.

On veut paralléliser, **en OpenMP/C**, les fonctions ci-bas. Complétez-les pour **à les rendre les plus parallèles et efficaces possible. Indiquez aussi quelle `schedule` il faudrait utiliser!**

```
void saxpy( float a, float x[], float y[], int n )
{



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


      y[i] = a * x[i] + y[i];

  }

}
```

```
float produit_scalaire( float x[], float y[], int n )
{
  float ps = 0.0;


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


      ps += x[i] * y[i];

  }


  return ps;
}
```

5. Opérations sur des Sequences en MPI/C (mémoire distribuée) (10 pts)

Soit le type `Sequence` et les fonctions ci-bas, qui permettent de manipuler des séquences **d'entiers** (séquences non-génériques).

```
typedef struct {
  int tailleMax;  // Nombre  maximum d'éléments de la séquence.
  int nbElems;    // Nombre effectif d'éléments.
  int *elems;     // Le tableau des éléments.
} Sequence;

Sequence sequenceVide( int tailleMax )
{
  Sequence s;
  s.tailleMax = tailleMax;
  s.nbElems = 0;
  s.elems = (int*) malloc( tailleMax * sizeof(int) );
  return( s );
}

void push( Sequence* s, int x )
{
  assert( s->nbElems < s->tailleMax );
  s->elems[s->nbElems++] = x;
}
```

On veut définir, avec MPI/C, une fonction `reduce` ayant la signature ci-bas :

```
typedef int (*BinOp)(int, int);

/* Applique l'opération binaire op sur les éléments de s, en utilisant valInit comme valeur
 * initiale -- c'est cette valeur qui est retournée si s est vide.
 *
 * On suppose qu'initialement la séquence s est sur le noeud racine et
 * on veut que le résultat final soit connu aussi (et seulement!) sur le noeud racine.
 */
int reduce( Sequence s, BinOp op, int valInit, int racine )
```

Voici un exemple d'appel de cette fonction, où `RACINE` est un entier approprié.

```
int plus( int x, int y ) { return( x + y ); }
...
int myID;  MPI_Comm_rank( MPI_COMM_WORLD, &myID );
...
Sequence s;
if( myID == RACINE ) {
  s = sequenceVide(n);
  for( int i = 1; i <= n; i++ )  // s = [1, 2, ..., n-1, n]
    push( &s, i );
}

int r = reduce( s, plus, 0, RACINE ); // Utilise tous les processus de MPI_COMM_WORLD

if( myID == RACINE )
  assert( r == n * (n+1) / 2 );       // Réponse valide seulement sur RACINE!
```

Complétez la fonction `reduce` ci-bas. Vous ne pouvez/devez pas utiliser `MPI_Reduce` :[1] vous devez utiliser uniquement `MPI_{Bcast,Scatter,Gather}`. N'oubliez pas que `s` — et donc sa taille — n'est initialement connue que sur le processus `racine`, d'où la définition de `nbLocal` ci-bas!

```
int reduce( Sequence s, BinOp op, int valInit, int racine )
{
  int myID, nbProcs;
  MPI_Comm_rank( MPI_COMM_WORLD, &myID );
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

  int nbLocal;
  if( myID == racine ) {
    assert( s.nbElems % nbProcs == 0 && "s.nbElems doit etre divisible par nbProcs" );
    nbLocal = s.nbElems / nbProcs;
  }









}
```

[1] On pourrait utiliser `MPI_Reduce` **si (!?)** on utilisait `MPI_Op_create`... ce qu'on ne fera pas dans le cadre de cet examen, n'ayant pas vu cette fonction dans le cadre du cours!

6. Diffusion et distribution de la température dans une salle (10 pts)

Pour ce problème, on désirerait utiliser une machine parallèle telle que `turing`, que l'on programmerait avec MPI/C... bien que vous n'ayez pas de code à écrire!

Le problème

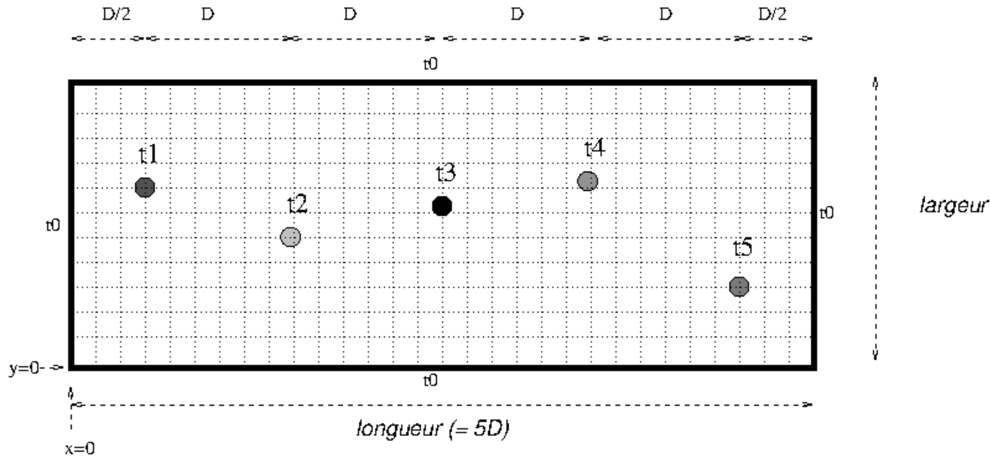

Figure 1: Salle rectangulaire avec sources de chaleur.

La figure 1 présente la configuration générale d'une salle rectangulaire ($longueur \times largeur$) avec des sources *ponctuelles* de chaleur — petits cercles, de dimension négligable (point) — disposées à différents endroits. Les lignes pointillées à l'intérieur du rectangle servent uniquement à illustrer la «discrétisation de l'espace» et ne font pas partie de la configuration de la salle (voir plus bas).

Les caractéristiques de la configuration d'une telle salle sont les suivantes :

- La salle est de forme rectangulaire et mesure $longueur \times largeur$.
- La salle possède n sources de chaleur — illustré ici avec $n = 5$. Ces sources ont des températures possiblement différentes les unes des autres ($t1, t2, \ldots$, illustrées aussi par la «couleur» des différentes sources). Il s'agit de sources *ponctuelles*, donc sans «dimension».
- En termes de position *horizontale*, les sources de chaleur sont réparties uniformément dans la salle. Si on note par $x = 0$ le coté gauche de la salle, alors les sources de chaleurs sont positionnées aux coordonnées $x = \frac{1}{2}D, \frac{3}{2}D, \frac{5}{2}D, \frac{7}{2}D, etc.$, où D est la distance entre les positions horizontales des sources de chaleur — et, donc, la longueur de la salle est nD.
- Les positions *verticales* **peuvent varier** d'une source à une autre.
- La température aux frontières de la salle, tant horizontales que verticales, est fixe.

On veut écrire un programme qui va simuler (approximer) la diffusion et distribution de la chaleur pour une telle salle. Pour ce faire, on va utiliser une approche semblable à celle vue en cours pour la diffusion de la chaleur dans un cylindre, c'est-à-dire approximation de la solution exacte par une discrétisation de l'espace (ici, une grille de points) et du temps (itération de la simulation). Toutefois, à la différence de l'exemple vu en cours, il s'agit ici d'une simulation **sur un espace à deux (2) dimensions** plutôt qu'à une seule dimension (cylindre). Dans un tel cas, l'équation à utiliser est celle dite *de Jacobi*, qui donne la température au point i, j de la grille au temps $t + 1$ en fonction de la température des voisins immédiats au temps t comme suit :

$$T_{t+1}[i, j] = \frac{T_t[i-1, j] + T_t[i+1, j] + T_t[i, j-1] + T_t[i, j+1]}{4}$$

En d'autres mots, la valeur pour un point $[i, j]$ de la grille (espace 2D) est définie par la moyenne des *quatres voisins immédiats* (nord, sud, est, ouest, ou encore gauche, droit, haut, bas) — contrairement à la simulation du cylindre où on utilisait la moyenne des deux voisins (espace 1D).

Le programme à concevoir va recevoir les arguments suivants sur la ligne de commande :

1. Longueur de la salle ;
2. Largeur de la salle ;
3. Température aux frontières de la salle (`t0`) ;
4. Nombre de sources chaleur (par exemple, 5) ;
5. Pour chaque source de chaleur, position verticale ($0 < y < largeur$) et température (`t1`, ..., `t5` de l'exemple) ;
6. Distance entre les points de simulation (discrétisation de l'espace), tant au niveau vertical qu'horizontal, i.e., distance entre les lignes pointillées de la figure. Pour simplifier, on suppose que longueur et largeur sont des multiples de cette distance ;
7. Durée de la simulation, i.e., nombre d'itérations où on va simuler l'évolution de la température.

Ce que vous devez faire

Ci-bas et page suivante, décrivez et comparez — avantages et désavantages, forces et faiblesses — diverses façons de concevoir un programme MPI pour résoudre ce problème. **Vous n'avez pas à écrire de code MPI!** Vous devez simplement décrire les grandes lignes des approches proposées, en termes de décomposition et agglomération des tâches, de communications entre les tâches, etc. Vous devez aussi conclure en indiquant **l'approche que vous suggéreriez d'utiliser, de façon à bien exploiter les ressources de la machine et réduire les communications**.