# 7 Multithreading

**Exercice 34.** Parallélisation du calcul de $\sum_{i=1}^{n} |\cos(i)|^i$

La classe `StopWatch` dans l'espace de noms `System.Diagnostics` permet de créer des chronomètres. Les méthodes `Start()` et `Restart()` et la propriété `ElapsedMilliseconds` seront utiles.

1. Ecrire une méthode statique
   `double Somme(long n0, long n1)`
   qui renvoit $\sum_{i=n0}^{n1} |\cos(i)|^i$.
   Faire un test avec $n0 = 1$ et $n1 = 10^7$. On déterminera aussi le temps de calcul.
2. Ecrire une méthode statique
   `double SommeP(long n, int p)`
   qui renvoit $\sum_{i=1}^{n} |\cos(i)|^i$ en invoquant en parallèle (utiliser `Task.Run`)
   Somme(i*(n/p)+1, (i+1)*(n/p)), $0 \le i \le p-1$. ($p$ est supposé diviser $n$)

**Exercice 35.** Tri à bulles

Pour trier un tableau, l'algorithme du tri à bulles consiste à parcourir toutes les paires d'éléments consécutifs et les intervertir s'ils ne sont pas dans le bon ordre. La procédure est répétée jusqu'à ce que le tableau soit trié. La méthode suivante met en oeuvre le tri à bulles pour un tableau d'entiers :

```
public static void Trier(int[] tab)
  {
   int k = 1;
   int aux;
   while (k>0){
    k = 0;
    for (int i = 0; i < tab.Length - 1;i++)
    {
     if (tab[i]>tab[i+1]){
      aux = tab[i];
      tab[i] = tab[i + 1];
      tab[i + 1] = aux;
      k++;
     }
    }
   }
  }
```

Afin de réaliser des tests, on donne une méthode qui fabrique la suite des entiers $0, \ldots, n-1$ permutée au hasard.

```
public static Random gen=new Random();

public static int[] permuAl(int n)
  {
   int[] permu = new int[n];
   for (int i = 0; i < n; i++) { permu[i] = i; }
   int k;
   int aux;
   for (int i = 0; i < n; i++)
   {
    k = gen.Next(0, n - i);
    aux = permu[i];
    permu[i] = permu[i + k];
    permu[i + k] = aux;
   }
   return permu;
  }
```

1. Faire un test de la méthode `Trier` pour un tableau de longueur $n = 15000$. On vérifiera que le tableau est bien trié et on donnera le temps d'exécution.
2. Ecrire un programme qui invoque deux fois en parallèle la méthode `Trier` avec le même tableau en argument. Faire un test pour vérifier si le tableau obtenu est correct.
3. Reprendre la question précédente en utilisant `lock` avec le tableau comme verrou pour remédier au problème. Que dire de la vitesse du tri ?
4. Ecrire une méthode statique
   `void Trier(int[] tab, int p, int r)`
   qui trie, par l'algorithme du tri à bulles, le tableau formé par les `tab[p*i+r]` où r est supposé compris entre 0 et p-1.
5. Ecrire une méthode `void Trier(int[] tab, int p)`
   quit met en œuvre la procédure suivante :
   - Tout d'abord trier tous les sous-tableaux `tab[p*i+r]`, $0 \leq r \leq p-1$, en parallèle.
   - Ensuite invoquer `Trier(tab)`
   Vérifier que le résultat produit est correct. Que dire de la vitesse d'exécution ?

# 8 Parallélisation de la méthode de Monte-Carlo

## 8.1 L'interface IGenerateur

Pour représenter un générateur de nombres aléatoires fournissant des réalisations indépendantes de loi donnée, on introduit l'interface suivante :

```
public interface IGenerateur{
  double NextDouble();
  IGenerateur Clone();
 }
```

La méthode `NextDouble()` a vocation de fournir un nouveau nombre "aléatoire" à chaque invocation. La méthode `Clone()` sera utile pour la parallélisation : elle doit fournir un générateur (du type `IGenerateur`) indépendant de l'objet courant mais correspondant à la même loi de probabilité. Voici un exemple concernant la loi uniforme sur $[0,1[$ :

```
 public class Gunif : Random, IGenerateur{
  static Random g = new Random();

  public Gunif(int graine) : base(graine){}

  public Gunif() { }

  public IGenerateur Clone(){
   return new Gunif(g.Next());
  }
 }
```

Remarquer que dans la méthode `Clone()`, on a pris soin d'instancier un générateur avec une graine tirée au hasard.

## 8.2 La méthode de Monte-Carlo

Etant données une fonction $f:\mathbb{R}\to\mathbb{R}$, une loi de probabilité $\nu$ sur $\mathbb{R}$ et un entier $n\geq 1$, on souhaite calculer

$$M_n=\frac{1}{n}\sum_{i=1}^{n}f(X_i)\quad \text{et}\quad V_n=\big(\frac{1}{n}\sum_{i=1}^{n}f(X_i)^2-M_n^2\big)/n$$

où $X_1,\ldots,X_n$ sont une réalisation de variables aléatoires i.i.d. de loi commune $\nu$. L'intérêt est que $M_n$ est une approximation de la moyenne $\int_{\mathbb{R}}f(x)\nu(dx)$ si elle existe. Plus précisément on sait que l'intervalle $I_n=[M_n\pm 1,96*\sqrt{V_n}]$ contient $\int_{\mathbb{R}}f(x)\nu(dx)$ avec une probabilité proche de 0,95 si $n$ est "grand".

**Exercice 36.** Ecrire une classe `MonteCarlo` munie des membres suivants :
- Un attribut `IGenerateur g`
- Un constructeur `public MonteCarlo(IGenerateur gen)` chargé d'initialiser l'attribut g
- Une méthode publique
  `Tuple<double,double> MoyVarEmpi(Func<double,double> f, ulong n)`
  qui renvoie un couple $(M_n, V_n)$ associé.

Donner une estimation de $\int_0^1 4/(1+x^2)dx$ et vérifier que l'intervalle de confiance $I_n$ contient $\pi$ 19 fois sur 20 environ.

## 8.3 Simulation de la loi gaussienne

La méthode dite de Box-Muller est basée sur le fait suivant :
Si $U$ et $V$ sont deux variables indépendantes, toutes deux de loi uniforme sur $[0,1]$, alors $X = \sqrt{2\log(V)}\cos(2\pi U)$ et $Y = \sqrt{2\log(V)}\sin(2\pi U)$ sont des gaussiennes indépendantes toutes deux de loi $\mathcal{N}(0,1)$.

**Exercice 37.** En prenant modèle sur `GUnif`, concevoir une classe `RGauss` qui dérive de `Random` et qui implémente l'interface `IGenerateur` permettant de simuler des gaussiennes de loi $\mathcal{N}(0,1)$. En particulier, on redéfinira la méthode
`protected override double Sample()`
qui est utilisée dans la classe `Random` pour la méthode `NextDouble` (donc il est inutile de redéfinir `NextDouble`).

Estimer $E[|Z|]$ où $Z \sim \mathcal{N}(0,1)$.

## 8.4 Parallélisation

**Exercice 38.** Ecrire une classe `MonteCarloP` avec les membres suivants :
- Un attribut `MonteCarlo[] tmc`
- Un constructeur `MonteCarloP(IGenerateur gen, int p)` dont le rôle est d'instancier `tmc` comme un tableau de taille $p$ dont les élements sont instanciés en utilisant des générateurs indépendants correspondant à la même loi de probabilité que l'argument `gen` (utiliser `gen.Clone()`).
- Une méthode publique
  `Tuple<double,double> MoyVarEmpi(Func<double,double> f, ulong n)`
  qui renvoie un couple $(M_n, V_n)$ associé, où les calculs sont effectués sur `p=tmc.Length` threads en parallèle.

Quel est le gain en temps de calcul pour l'estimation de $\int_0^1 4/(1+x^2)dx$ et $E[|Z|]$ (où $Z \sim \mathcal{N}(0,1)$) ?

## 8.5 La méthode du rejet

Etant donnée une variable aléatoire $X$ et un seuil $s \in \mathbb{R}$ on veut simuler la loi de $X$ sachant $\{X \geq s\}$. Pour cela il suffit de pouvoir calculer $X_T$ où

$$T = \min\{n \geq 1 : X_n \geq s\}$$

et $(X_n)_{n\geq 1}$ est une suite de variables i.i.d. de même loi que $X$.

**Exercice 39.** Ecrire une classe `RRejet:IGenerateur` possédant
— un attribut `IGenerateur g` (correspondant à la loi de $X$),
— un attribut `double seuil`
de sorte que la méthode `Next` renvoie un nombre aléatoire dont la loi est celle de $X$ sachant $\{X \geq s\}$.
Estimer $E[Z \mid Z \geq 4]$ où $Z \sim \mathcal{N}(0,1)$.

## 8.6 Parallélisation du générateur

Quand un générateur g est “lent”, on souhaite en fabriquer un plus rapide en parallèlisant.

**Exercice 40.** Définir une classe `RP : IGenerateur` avec les membres suivants :
— Un attribut `IGenerateur[] tg`
— Un attribut `Task<double>[] tt[]`
— Un constructeur `RP(IGenerateur g, int p)` qui initialise le tableau `tg` de sorte qu’il soit de taille $p$ et peuplé de copies “indépendantes” de g. Le constructeur initialise aussi le tableau `tt` de sorte que `tt[i]` correspond à l’appel de `tg[i]`.
Dans la méthode `NextDouble`, on renverra le résultat de la première tâche terminéee dans `tt` en prenant soin de relancer cette tâche avant.
Reprendre l’estimation de $E[Z \mid Z \geq 4]$ en utilisant les classes `RP` et `MonteCarlo`, puis `RP` et `MonteCarloP`. Comparer les temps de calcul.