Threads

Nous supposons que votre plate-forme de développement supporte les threads C11 et les variables et opérations atomiques. Pour cela nous vous suggérons d'utiliser la lib C alternative "musl" avec une version de gcc moderne :

```
musl-gcc -static -Wall -O3 -std=c11 -o toto toto.c
```

Pour avoir accès aux opérations atomiques et aux threads C11 il faut les includes dans votre programme :

```
#include <stdatomic.h>
#include <threads.h>
```

Le but de cet exercice est de créer plusieurs exécutables et de les comparer.

Gardez une copie de votre source pour chacun des exercices !

Exercice 1

Pour vous habituer à écrire du C modèrne, implantez deux fonctions avec les prototypes :

```
double sum2D(size_t ntot, size_t mtot, size_t n0, size_t len, double A[ntot][mtot]);
void rand2D(size_t ntot, size_t mtot, size_t n0, size_t len, double A[ntot][mtot]);
```

Ces fonction accèdent à tous les éléments des lignes $A[n_0], \ldots, A[n_0 - len]$ de la matrice A de dimension 2 avec taille n_{tot} lignes et m_{tot} colonnes.

`sum2` retourne la somme de tous ces éléments.

`rand2D` initialise ces éléments avec une valeur aléatoire.

Écrivez un `main` qui récupère n_{tot} et m_{tot} de la ligne de commande, alloue une matrice de cette taille, l'initialize avec `rand2D`, et fait la somme de tous les éléments avec `sum2D`.

Pour mémoire, les allocations de grande taille ne doivent jamais être effectués sur la pile d'une fonction. Utilisez `malloc` pour allocation sur le tas :

```
double (*A)[mtot] = malloc(sizeof(double[ntot][mtot]));
```

Et n'oubliez pas de placer un appel à `free` vers la fin de l'exécution.

Exercice 2

Parallélisez `sum2D` à l'aide de thread C11 en écrivant une fonction `sum2Dpar`. Celle-ci réalise un interface similaire à `sum2D` mais récupère aussi le nombre de thread à utiliser.

```
double sum2Dpar(size_t ntot, size_t mtot, double A[ntot][mtot], size_t threads);
```

`sum2Dpar` **n'utilisera aucune variable globale pour transmettre des informations à ses threads.**

Pour ce faire, implémentez une fonction pour le lancement d'un de ces thread

```
int sum2Dthread(void* arg);
```

qui reçoit en `arg` un pointeur opaque sur une structure de type `thread2Dinfo` qui contient l'information pour composer un appel à `sum2D`.

Testez votre fonction `sum2Dpar` avec différents nombres de threads.

Exercice 3

Parallélisez `rand2D` de façon analogue par une fonction `rand2Dpar`. Pour cela réutilisez le même type de structure `thread2Dinfo`.

Attention, les générateurs aléatoires gardent un état entre différents appels. Cet état peut être géré de façon implicite (p.ex pour `rand`) ou explicitement donnée en paramètre (p.ex la fonction POSIX `erand48`). Une parallélisation de qualité doit s'assurer que les différents threads ne se marchent pas sur les pieds (performance ou exactitude réduite) et produisent des valeurs indépendants.

Rajoutez cette fonction parallèle à votre test

Exercice 4

Exécutez votre programmes avec 1, 2, 4 et 8 thread et mésurez les temps d'exécution avec `/usr/bin/time`.

Comparez ce temps avec une exécution séquentielle de votre premier programme. Quel accélération observez vous ?