Threads

1 Appels système `pthread_create` et `pthread_join`

1. Familiarisez vous avec les appels système `pthread_create` et `pthread_join` en créant un programme qui crée n threads (n étant le premier argument du processus). Chaque thread affiche en boucle son identifiant unique, obtenu avec `pthread_self` (si nécessaire, vous pouvez ajouter un appel à la fonction `sleep` pour ne pas que les affichages ne soient pas trop rapides.). Compilez avec l'option `-l pthreads` et exécutez votre programme, puis contrôlez l'affichage pour vérifier que les threads s'exécutent bien simultanément.
2. Modifiez la fonction exécutée par les threads pour afficher un entier passé en argument de la fonction plutôt que l'identifiant du thread. Modifiez également le `main` pour que chaque thread affiche un chacun un entier distinct entre 0 et $n - 1$.

2 Exclusion mututelle

1. En vous inspirant de l'exercice précédent, écrivez un nouveau programme qui crée n thread. Chaque thread devra incrémenter une variable globale m fois (m sera le deuxième argument de votre programme). Ne mettez pas de `sleep`. À la fin du programme affichez la valeur de la variable globale depuis le `main`. Compilez et exécutez plusieurs fois votre programme pour différentes valeurs de m (10 100 1000 10000 ...). Que constatez vous?
2. Utilisez les appels système `pthread_mutex_lock` et `pthread_mutex_unlock` pour corriger le problème. Faut-il placer les primitives à l'intérieur ou à l'extérieur de la boucle `for`? Expliquez pourquoi?

3 Produit scalaire parallèle

Pour pouvoir exploiter les architectures multi-cœurs, on souhaite implementer des primitives de calcul multi-threadees. Dans cet exercice, vous allez implementer un produit scalaire parallelise grace aux threads.

1. Écrivez une fonction (non parallèle) `scalar(float v1[], float v2[], int n)` qui permet de calculer le produit scalaire entre deux vecteurs v_1 et v_2 de dimension n. Rappel, le produit scalaire se calcul avec la formule suivante.

$$v_1 * v_2 = \sum_{i=0}^{n} v_i[i] * v_2[i]$$

2. Créer une nouvelle méthode pour tester votre algorithme sur deux vecteur de taille n initialisés avec des nombres aléatoires. Pour générer des nombres aléatoires, utilisez `rand` (Voir `man 3 rand`). Note:`rand` ne génère que des entiers, vous pouvez obtenir des nombre décimaux entre 0 et 1 en divisant par `RAND_MAX`, mais attention, il y a plusieurs bus possibles. Créez une fonction pour afficher vos vecteurs, et vérifiez que tout fonctionne correctement.
3. Pour pouvoir effectuer le produit scalaire en parallèle, il faut pouvoir découper le calcul en plusieurs tâches indépendantes qui seront exécutées par différents threads. Ici, le découpage

en taches est naturel : chaque thread va prendre en charge la multiplication de $\frac{1}{t}$ valeurs (t: nombre de threads). Sans modifier la signature de la fonction `scalar` créez `t` threads qui calculent chacun une partie différente du produit scalaire (on suppose que $n \gg t$). Puis modifier le reste du programme pour calculer et afficher le résultat final.

4. Mesurez le temps d'exécution avec la commande `time`. Comparez votre implémentation parallèle avec 2 threads, avec votre implementation séquentielle. À partir de quelle taille l'utilisation de plusieurs threads fait-elle gagner du temps?