SR02 : TD 8 (Threads)
TD Machine sur une séance de 2h

Objectifs :
- Utiliser les threads POSIX
- Apprendre à paralléliser une tache en utilisant les threads

Crible d'Eratosthenes

Le but de ce TD machine est d'implémenter une version parallèle de l'algorithme du crible d'Eratosthenes en utilisant les threads POSIX.

Tout d'abord lisez sur le crible d'Eratosthenes en suivant ce lien :

http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

https://fr.wikipedia.org/wiki/Crible_d%27%C3%89ratosth%C3%A8ne

L'algorithme suivant présente une version séquentielle pour le crible d'Ératosthène :

```
Input: un entier n > 1
Soit A un tableau de valeurs booléennes, indexées de 2 à n, et initialisées toutes à vrai.
for i = 2, 3, 4, ..., √n :
  si A[i] est vrai:
    for j = i², i²+i, i²+2i, ..., n:
      A[j] := faux
Output : Maintenant tout i telle que A[i] est vrai est un nombre premier.
```

Tache 1

Regardez d'abord le pseudo code ci-dessus et répondez aux questions suivantes:
- Dérouler l'exécution de cette algorithme avec n=20
- Pourquoi la boucle de l'intérieure (la deuxième boucle) commence à i^2 et pas 0 ou i ?
- Pourquoi la première boucle s'exécute jusqu'à √n? Que faites-vous si √n n'est pas un entier ?

Tache 2

Comme premier étape pour votre version parallèle du programme, implémenter une version séquentielle pour le crible d'Ératosthène.

Tache 3

A partir de votre version séquentielle vous allez maintenant développer une version parallèle. L'idée est de paralléliser la boucle interne du pseudo code "*for j = i^2, i^2+i, i^2+2i, ..., n*", en distribuant l'exploration de "*i^2, i^2+i, i^2+2i, ..., n*" sur *k* threads. (*k* est un paramètre de votre implémentation.)

Pour éviter le coût de création répétée de threads, les *k* threads de votre programme doivent être créés en dehors de la boucle externe des algorithmes, et être réutilisés à chaque itération de la boucle externe.

Notes:

- Il y a 2 points de synchronisation dans ce code: Les threads de travail doivent attendre que le thread principal initialise le travail à effectuer (*); et le thread principal doit alors attendre que les threads de travail finissent leur part du travail (**).
- Pour répartir le travail entre les threads de travail, le thread principal indique à chacun d'eux quelle plage il faut

couvrir. Pour ce faire, le thread principal doit définir cette plage (comme paramètre dans la fonction *creat_thread*) pour chaque thread de travail avant de débloquer le thread de travail. Chaque thread doit stocker la plage qu'il doit couvrir dans les attributs appropriés.

Une fois que vous avez implémenté une version parallèle du crible d'Eratosthenes, vérifiez cette version avec différentes valeurs de k ($k = 1, 2, 3, 4, 7$).

Tache 4

L'objectif de cette dernière tâche est de comparer les performances de vos versions successives et parallèles du crible d'Eratosthenes.

Mesurez le temps nécessaire pour calculer tous les nombres premiers ci-dessous 500,000 avec:

- Votre version séquentielle;
- Votre version parallèle avec le nombre de threads variant de 1 à 7.

Tracez les résultats sur un graphique. (Vous devriez exécuter vos expériences plusieurs fois pour obtenir des moyennes représentatives, vous saurez comment calculer un intervalle de confiance sur ces valeurs?)

Répétez la même mesure pour tous les nombres premiers inférieurs à 1 000 000; 2,000,000; 4 000 000, et dessinez les courbes correspondantes sur le même graphique. Comment interprétez-vous vos résultats?

Tache 5

Analyser l'impact des deux optimisations suivantes :

- *Accélérer la boucle interne*: Votre boucle interne utilise un pas de *i*. Comment pourriez-vous le modifier pour utiliser un pas de *2i* au lieu de *i* ? (Indice: vous devrez considérer 2 comme un cas distinct.)

- *Réduction de l'espace mémoire:* votre implémentation utilise un tableau de Boolean (disons *isPrime []*) pour se souvenir quels nombres sont premiers. Cette approche utilise beaucoup d'espace: tous les nombres pairs (donc presque la moitié du tableau) sauf 2 seront mis à faux. Puisque nous savons déjà que (i) 0 et 1 ne sont pas premiers (par définition), et que (ii) tous les nombres pairs sauf 2 (i.e. 4,6,8, ...) ne sont pas premiers non plus, une approche plus efficace consiste à en utilisant un tableau qui est moitié moins grand, et en maintenant seulement un booléen pour les nombres impairs au-dessus de 3. En d'autres termes, vous pouvez utiliser *isPrime [i]* pour indiquer si k = *2 * i + 3* est premier ou non.

Barème:

Tache 1: 3 points
Tache 2: 2 points
Tache 3: 7 points
Tache 4: 2 points
Tache 5: 6 points