Laboratoire 2 - isPrime

Auteurs: Yann Merk, Adrian Barreira Romero

Description des fonctionnalités du logiciel

L'application qui a été réalisée permet, conformément au cahier des charges, de vérifier si un nombre est premier ou pas, avec les classes *PrimeNumberDetector* et *PrimeNumberDetectorMultiThread*. Elles implémentent toutes les deux l'interface *PrimeNumberDetectorInterface,* et proposent la méthode `bool isPrime(uint64_t number)`.

L'algorithme pour trouver un nombre premier est générique et peut être trouvé simplement, la logique étant fournie dans la partie 1 du labo. Le voici :

```
// Corner case
if (number <= 1)
    return false;

// Check from 2 to n-1
for (int i = 2; i <= sqrt(number); i++)
    if (number % i == 0)
        return false;

return true;
```

Choix d'implémentation

Dans la classe *PrimeNumberDetectorMultiThread* un attribut `nbThreads` est présent pour déterminer le nombre de threads lors de la création de l'objet (via constructeur). Afin que les threads puissent communiquer leur résultat en temps réel entre eux nous avons rajouté `bool hasDividerBeenFound`, qui passe à `true` dès que l'on sait qu'un nombre n'est pas réel.

Nous avons rajouté une fonction `void checkPrimeThreaded(uint64_t number, size_t offset)` qui va faire tourner l'algorithme de vérification de nombre premier similaire à la version mono thread. Les paramètres :

- **number** : le nombre à vérifier
- **offset** : un nombre de 0 à `nbThreads-1` servant à décaler le point de départ du thread afin de répartir la charge équitablement entre threads

Chaque thread va effectivement commencer de faire ses modulos depuis `2+offset`, et va faire des sauts de taille `nbThreads`, pour couvrir tous les nombres sans en traiter à double.

```
void PrimeNumberDetectorMultiThread::checkPrimeThreaded(uint64_t number, size_t offset) {
    // Check from 2 to n-1
    for (int i = offset + 2; i <= sqrt(number); i+= nbThreads)
    {
        if (number % i == 0)
        {
            hasDividerBeenFound = true;
            return;
        }
        if(hasDividerBeenFound)
            return;
    }
}
```

Ainsi, lorsque un des threads aura trouvé un diviseur, nous interrompons les itérations de tous les threads dès qu'ils voient `hasDividerBeenFound` passé à `true`. Nous avons volontairement choisi de ne pas mettre de mécanisme particulier pour attendre une potentielle écriture de ce booléen en partant du principe qu'il était preferable de laisser tourner les threads quelques temps en plus plutôt que d'augmenter le nombre d'instructions à chaque itérations.

Pour ce qui est de l'instanciation des threads, nous les stockons simplement dans un vecteur, en les lançant avec un offset incrémental.

```
bool PrimeNumberDetectorMultiThread::isPrime(uint64_t number) {
    // Corner case
    if (number <= 1)
        return false;

    // Reset
    hasDividerBeenFound = false;

    std::vector<PcoThread> threads;

    for(size_t i = 0; i < nbThreads; ++i) {
        threads.push_back(PcoThread(&PrimeNumberDetectorMultiThread::checkPrimeThreaded, this, number, i));
    }

    for(size_t i = 0; i < nbThreads; ++i) {
        threads[i].join();
    }

    return !hasDividerBeenFound;
}
```

A la fin du programme, il est obligatoire d'attendre que tous les threads aient terminé pour les nettoyer correctement, et évidemment s'assurer que toutes les valeurs inférieures ou égales à sqrt(n) aient été parcourues avant de donner un résultat.

Tests effectués

Les tests étant fournis dans différents "main", nous n'avons eu qu'à les exécuter.
main_test comportait différentes instantiations des classes et différents nombres premiers ou non premiers à trouver correctement tandis que main_bench comporte des calculs de temps d'execution pour différent nombre de threads.

Tous les tests ont été réussis ou ont eu des résultats pertinents, les nombres ont correctement été déterminés comme premiers ou non et les benchmarks semblent cohérents.

Réponses aux questions :

1. Mesurez le gain de temps produit par votre version multi-threadée en faisant varier le nombre de threads. Que remarquez-vous ?

Niveau performance l'initialisation et terminaisons des threads prends du temps. Plus la quantité de thread est élevée, plus ce temps est grand. Mais, en traitant des grands nombre il est possible de quand même gagner du temps par rapport à la version mono-threadée, car même si les threads individuellement comportent plus d'opération que la version séquentielle, leurs opérations en parallèle compense cela.

Mais pour vraiment gagner du temps il faudrait possiblement lier la quantité de thread avec la taille du nombre, quitte à même simplement tester des nombres premiers basique, p.ex tous ceux en dessous de 1000, avant de lancer un grand nombre de thread ; Il est inutile de lancer 15 threads pour un nombre qui peut déjà se faire diviser par deux.

Mais la recherche de performance n'étant pas un objectif de ce labos où ce cours, nous n'avons pas cherché à implémenter de tels solutions.

2. Le gain de vitesse est-il linéaire avec le nombre de threads ? (Indiquez le nombre de threads alloués à la VM, il peut être changé dans les settings, ainsi que le nombre de threads de votre machine). Il y a-t-il une différence pour les nombre premiers et non premiers ?

Tant que le nombre de threads est inférieur ou égal au nombre de coeurs alloué à la VM, et que le nombre à tester est un minimum conséquent il y a une amélioration linéaire, qui est très facilement explicable : au lieu d'avoir un coeur seul effectuant n opérations, avoir plusieurs threads tournant, même si leur gestion rajoute du temps fixe à l'exécution, fait que t coeurs effectuent chacuns n/t opérations, ce qui fait que les n opérations sont effectuées plus rapidement

Oui, il y a une différence entre les nombre premier ou non car quand le programme détecte un diviseur il va immédiatement s'arrêter, et retourner le résultat. Autrement, il continue jusqu'à avoir fait sqrt(n) tests avant de retourner un résultat positif.
Par exemple, pour un nombre divisible par 2 l'exécution va s'arrêter extrêmement rapidement.