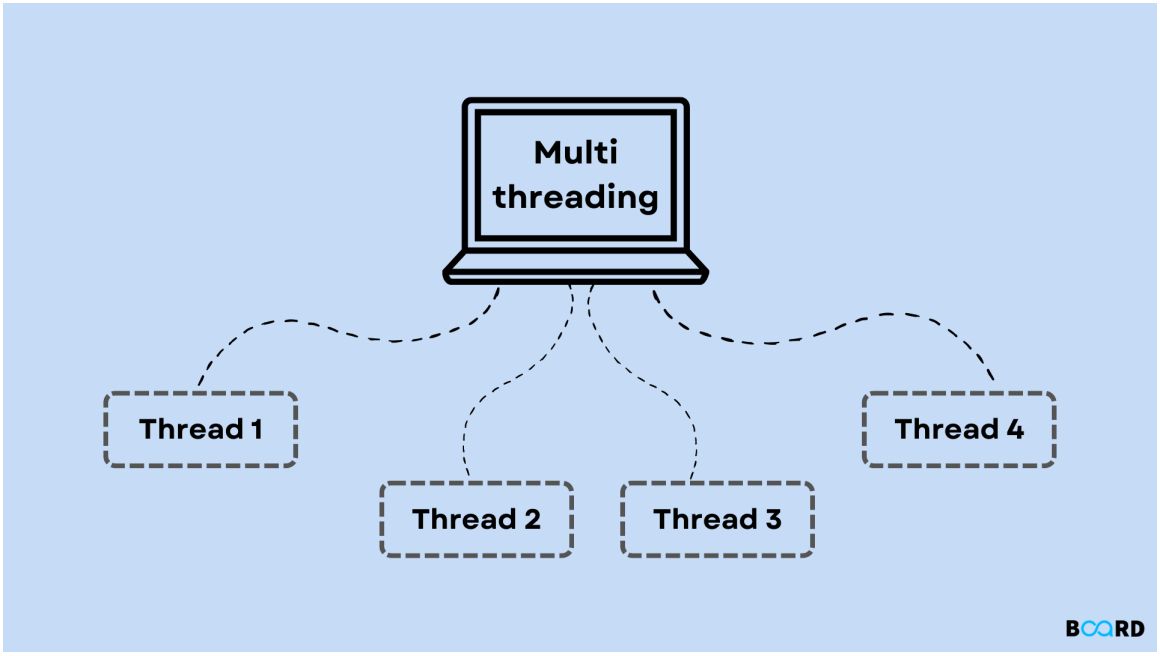
TP IDM

Génération de flux parallèles de nombres pseudo-aléatoires



CHASSAGNOL Rémi

Professeur : HILL David

18 décembre 2023

Table des matières

Table des extraits de codes

1 Introduction

L'objectif de ce TP est d'utiliser un générateur de nombres pseudo-aléatoires pour réaliser des calculs en parallèle. Pour générer les nombres nous utiliserons l'implémentation de Mersenne Twister fournie par la bibliothèque CLHEP. Nous commencerons par détailler l'installation de la bibliothèque, puis nous testerons la répétabilité des séquences de nombres générées aléatoirement ainsi que le bon fonctionnement de la sauvegarde des statuts du générateur dans des fichiers. Ensuite, nous traiterons l'exemple simple du calcul du nombre π, d'abord en séquentiel, puis en parallèle. Enfin, nous traiterons un autre exemple consistant à générer des séquences de bases nucléiques.

Pour pouvoir paralléliser les calculs nous commencerons par pré-générer des fichiers de statuts de Mersenne Twister. En effet, ce générateur produit toujours la même séquence de nombres (avec la seed fournie par Makoto Matsumoto), or on ne souhaite pas repartir du même point dans chaque thread. Pour que les threads utilisent des séquences différentes, nous utiliserons les fichiers de statuts pour lancer le générateur à partir de point différents dans la séquence. Pour ce faire, il faudra séparer les statuts d'un certain nombre d'étapes (nombre à généré) lors de la génération. Nous discuterons des limites de cette méthode dans la conclusion.

2 Utilisation de CLHEP

Dans cette première section nous allons voir comment installer la bibliothèque CLHEP. Pour simplifier la gestion des dépendances, ce projet utilise des sous-modules git.

Le clonage des sous-modules ne se fait pas automatiquement lors du clonage du projet. Pour avoir accès aux sous-modules, il faut utiliser les commandes de l'extrait de code 1.

```
1  git submodule init
2  git submodule update
```

Extrait de code 1: Clonage des sous-modules git

L'utilisation des sous-modules évite surcharger les serveurs contenant les dépôts en stockant plusieurs fois le même code. De plus, ils permettent de facilement cloner les dépendances (cela est plus simple que de télécharger la bibliothèque manuellement).

Pour compiler la bibliothèque, il faut utiliser le script `build.sh` du répertoire `lib`. Ce script va permettre de compiler et d'installer la bibliothèque (fichiers d'entêtes, bibliothèque statiques, ...) dans le répertoire `lib/CLHEP-lib`. Ensuite, on peut compiler le projet en utilisant CMake (utiliser un répertoire `build/` à la racine du projet). Malheureusement, le CMake de la bibliothèque ne fournit pas de moyen simple de compiler uniquement la partie *Random* qui est utilisée pour ce tp. La compilation peut donc être assez longue car le script compile tous les modules de la bibliothèque.
À noter que pour compiler la bibliothèque CLHEP, le script `build.sh` utilise `make` avec l'option `-j` pour paralléliser le compilation. Cette option est pratique car elle permet de compiler les fichiers plus rapidement, cependant, elle peut être consommatrice en ressources. En effet, ici, on ne spécifie pas le nombre de threads, `make` utilise toutes les ressources disponibles. Sur une machine peu puissante, il est recommandé de modifier le script soit en spécifiant une limite après l'option, soit la supprimant.

Maintenant que nous avons traité l'installation de la bibliothèque, nous allons pourvoir commencer le tp.

3 Test du générateur

Avant de commencer les calculs, assurons-nous que le générateur produit bien toujours les mêmes séquences de nombres lorsqu'il charge les fichiers de statuts.

La générations des fichier de statuts se fait avec la méthode `saveStatus` et le chargement des fichiers se fait avec `restoreStatus`. Dans la fonction `checkReproducibility`, nous générons dans un premier temps des fichiers de statuts, et chaque statuts est séparé par un certain nombre de tirages de nombres pseudo-aléatoires. Les tirages sont sauvegardés dans un tableau. Dans un second temps, nous restaurons les statuts et nous vérifions que les tirages sont les mêmes que ceux sauvegardés précédemment. Ici, nous utilisons un `assert` pour tester l'égalité des nombres. Si les nouveaux nombres générés ne sont pas égaux à ceux sauvegardés, le programme s'arrête avec un code erreur. Si le programme se termine sans échouer, le test du générateur est valide.

À noter que ce test basique ne permet pas de valider avec certitude le bon fonctionnement du générateur. Nous nous contenterons cependant de ce test étant donné le fait la répétabilité de Mercenne Twister peut être prouvée mathématiquement. Ici, nous vérifions juste que l'implémentation fonctionne sur un petit nombre de tirages.

4 Calcul de pi

Maintenant que nous avons vérifié le bon fonctionnement du générateur, nous allons réaliser un simple calcul de π en utilisant la méthode de Monte Carlo. Le calcul sera fait de façon séquentielle dans un premier temps, puis de façon parallèle dans un second.

4.1 Génération des fichiers de statuts

Pour pouvoir paralléliser les calculs, il faut préalablement générer des fichiers de statuts pour pouvoir lancer le générateur à partir d'un point donné. Sans cette étape, les calculs lancés en parallèles utiliseront la même séquence de nombre pseudo-aléatoires, ce qui rendrait la parallélisation inutile.

La génération des fichiers de statuts se fait à l'aide de la fonction `generateStatusFiles`. On peut configurer le nombre de fichiers à générer ainsi que le nombre de tirages séparant chaque statuts. Pour la suite, on utilisera les valeurs par défauts des paramètres. Il est important d'utiliser cette fonction avant de tester les autres fonctionnalité du programme car les fichiers de statuts seront utilisés dans les autres questions. Les fichiers générés sont nommés mt3_1, mt3_2, ...

4.2 Calcul séquentiel

Dans un premier temps, nous allons réaliser le calcul de π de façon séquentielle. Cela va permettre de vérifier notre algorithme et simplifiera la parallélisation.

Pour le calcul séquentiel, nous utilisons la fonction `piSequential` qui réalise plusieurs calculs de π et qui calcule la moyenne des résultats pour avoir un valeur approchée. Cette fonction prend en paramètre le nombre de réplications du calcul ainsi que le nombre de tirages pour chaque réplication. Pour chaque réplication, on restaure un statut du générateur puis on utilise la fonction `computePi`. Ici, il n'est pas vraiment nécessaire de restaurer les statuts du générateur à chaque itération, cependant, cela permet de facilement tester le code qui sera parallélisé par la suite. Le code de la fonction est visible sur l'extrait de code 2.

```cpp
void piSequential(size_t nbReplications, size_t nbDraws,
                  const std::string &fileName) {
    CLHEP::MTwistEngine mt;
    double sumPI = 0;

    timerStart();
    for (size_t i = 0; i < nbReplications; ++i) {
        mt.restoreStatus(cat(fileName, i).c_str());
        sumPI += computePi(mt, nbDraws);
    }
    timerEnd();

    std::cout << std::setprecision(10) << "pi: " << sumPI / nbReplications
              << "; time: " << timerCountMs() << std::endl;
}
```

Extrait de code 2: Fonction piSequential

La fonction permet aussi de mesurer le temps de calcul en utilisant les macros visibles sur le l'extrait de code 3. Ces macros permettent d'avoir une approximation du nombre de mili-secondes écoulées durant un calcul. Ici, on choisit une précision à la micro-seconde ce qui est très élevé étant donné le fait que l'exécution des calculs peut prendre plusieurs secondes. Cependant, cela reste intéressant s'il on souhaite faire des mesures plus précises.

```cpp
#define timerStart() auto _start = std::chrono::high_resolution_clock::now();
#define timerEnd() auto _end = std::chrono::high_resolution_clock::now();
#define timerCountMs()                                                          \
    std::chrono::duration_cast<std::chrono::microseconds>(_end - _start)        \
            .count() /                                                          \
        1000.0
```

Extrait de code 3: Macro timer

Ici, le calcul prend plus d'une minute pour s'exécuter. Dans la partie suivante, nous allons comparer ce temps de calcul avec celui obtenu avec la parallélisation.

4.3 Calcul parallèle

Dans cette section nous allons paralléliser le calcul de π avec la méthode de Monte Carlo. Nous commencerons par utiliser un script bash qui crée des processus, puis nous utiliserons les threads de la bibliothèque standard de C++.

4.3.1 Utilisation d'un script

Ici, nous allons utiliser un script bash pour paralléliser les calculs. L'objectif est de faire en sorte de pouvoir passer en argument du programme le nom d'un fichier de statut à charger de sorte à pouvoir lancer plusieurs fois le même programme en tâche de fond avec un fichier différent à chaque foi.

La fonction `piFromStatusFile` est une version simplifiée de la fonction `piSequential` qui ne lance qu'une seule fois le calcul. La fonction prend en argument le nom du fichier de statut à charger pour initialiser le générateur.

Le script utilisé pour lancer les processus est visible sur l'extrait de code 4. Ici, on commence par vérifier que le projet est bien compilé et que les fichiers de statuts ont bien été générés (dans le cas contraire, on fait le nécessaire) avec la fonction `checkSetup`. Ensuite on définit le nom des fichiers de sortie du programme (le nom est configurable avec les arguments du script), puis on lance 10 processus en tâche de fond (utilisation du `&` en fin de ligne) avec une boucle `for`. Une fois l'exécution terminée, les résultats sont stockés dans les fichiers du répertoire `out/`.

```
1  #!/usr/bin/env bash
2
3  set -euo pipefail
4
5  checkSetup() { }
6  checkSetup
7
8  # configure output file
9  outputFile=pi
10 if [ $# -eq 1 ]; then outputFile=$1; fi; mkdir -p out
11 # compute pi
12 for i in {0..9}; do
13     ./build/tp5 "./build/mt3_$i" > "./out/$outputFile$i.out" &
14 done
15 time wait; echo "done"
```

Extrait de code 4: Script compute_pi.sh

À noter qu'à la fin du script on utilise la commande `wait` qui va permettre d'attendre que tous les processus lancés en tâche de fond se terminent pour quitter le programme. On emploie aussi la commande `time` pour mesurer le temps de calcul total (on considère que le temps de lancement des processus est négligeable, on ne le prend donc pas en compte ici). Le calcul prend environ dix secondes ce qui est bien plus rapide que le temps de calcul séquentiel.

Dans cette section, nous avons utilisé un script pour paralléliser les calculs. Dans la section suivante, nous utiliserons les threads de la bibliothèque standard C++.

4.3.2 Utilisation des threads

La seconde solution utilise les threads de la bibliothèque standard de C++. Ici, on utilise la fonction `piParallel` disponible en deux versions, une utilisant directement les threads, et une autre utilisant l'abstraction `std::future`. La version utilisant `std::future` est visible sur l'extrait de code **??**. Les deux version utilisent la fonction `piFromStatusFile` présentée dans la section 4.3.1.

```
1  void piParallel(const std::string &fileName, size_t nbDraw) {
2      CLHEP::MTwistEngine mt;
3      size_t i;
4
5      timerStart();
6      /* threads scope */ {
7          std::future<void> pis[NB_REPLICATION];
8          for (i = 0; i < NB_REPLICATION; ++i) {
9              pis[i] = std::async(std::launch::async, piFromStatusFile, cat(fileName, i),
10                                 nbDraw);
11         }
12     }
13     timerEnd();
14     std::cout << "total time: " << timerCountMs() / 1000.0 << "s" << std::endl;
15 }
```

Extrait de code 5: Fonction piParallel

Pour cette solution, le temps de calcul est similaire à celui du script ce qui est logique étant donné le fait que le même nombre de threads a été utilisé.

```
./tp5 gattaca gen # génération des fichiers de statuts
./tp5 gattaca     # lancement de la simulation
```

Extrait de code 6: Lancement de la simulation GATTACA

5 Gattaca

Dans cette section, nous allons traiter un nouvel exemple de calcul parallèle. Ici, on va simuler la génération de séquences de bases nucléiques. Ces bases sont symbolisées par des lettres avec quatre possibilités A, C, G et T. Dans un premier temps, on va chercher à générer une séquence simple, "GATTACA". Ensuite, on discutera de la génération de séquences plus complexes.

Pour générer une séquence avec un certain nombre de bases, on utilise la fonction `generateSequence`. Cette fonction var permettre de générer une séquence, sous forme de chaine de caractères, avec une taille configurable.

Ensuite, on utilise la fonction `generateNSequences` qui va permettre de générer un certain nombre de séquences. Cette fonction prend en paramètre une séquence à chercher. La fonction s'arrête lorsque cette séquence est trouvée (ou lorsque le compteur atteint le nombre maximum d'itérations). Cette fonction est faite pour être parallélisée.

Enfin, la fonction `gattaca`, va lacer la fonction `generateNSequences` dans un certains nombre de threads et elle va collecter les résultats. À la fin, la fonction affiche le nombre de tirages moyen nécessaires pour trouver la séquence recherchée.

À noter qu'avant de lancer la simulation, il faut générer les fichiers de statuts du générateur. Pour ce faire, il faut utiliser les options `gattaca` et `gen` comme sur l'extrait de code 6.

Avec les règles énoncées précédemment, il y en tout $4^7 = 16,384$ possibilités lorsque l'on génère des séquences de 7 bases (GATTACA est une séquence de 7 bases). En moyenne, il faut générer $11,351$ séquences pour trouver GATTACA (avec le programme actuel), sachant que le nombre minimum d'essais trouvé par la simulation est de 94 et le nombre maximum est de $48,787$. L'écart type est d'environ $11,789$.

La séquence GATTACA très courte et nombre de possibilités est assez faible. La génération d'une séquence plus grande peut prendre beaucoup de temps. Par exemple, pour chercher une séquence de 18 bases comme `AAATTTGCGTTCGATTAG`, il y a en tout $4^{18} = 68,719,476,736$ possibilités. Ici, la plus grosse limitation vient de la génération des fichiers de statuts. En effet, la génération des fichiers se fait de manière séquentielle. Ici, si l'on souhaitait paralléliser la recherche sur N threads, il faudrait générer N fichiers de statuts avec environ 70 milliards de tirage, ce qui prendrait énormément de temps. Le problème de la génération de phrases est encore plus complexe car ici, il y a encore plus de possibilités.

6 Conclusion

En conclusion, on peut dire que les problèmes de simulation stochastiques sont facilement parallélisables. Cependant, avec des générateurs tels que Mersenne Twister, on reste limité par le fait que l'on ne peut pas facilement lancer le générateur à partir d'un certain point. La méthode employée ici consistait en la génération de fichiers de statuts, cependant pour créer ces derniers il faut tout de même faire des tirages de façon séquentielle avec le générateur ce qui peut prendre un temps conséquent.

Ce problème pousse les créateur de générateurs à trouver de nouvelles méthodes. Par exemple, Makoto Matsumoto a créé un *"générateur de générateur"* qui crée différents Mercenne Twister, et ce, de manière efficace. Malheureusement, la fiabilité de ce générateur n'a pas encore été prouvée. Avec ce type de générateurs, plus besoin de générer des fichiers de statuts, on peut directement paralléliser les calculs.