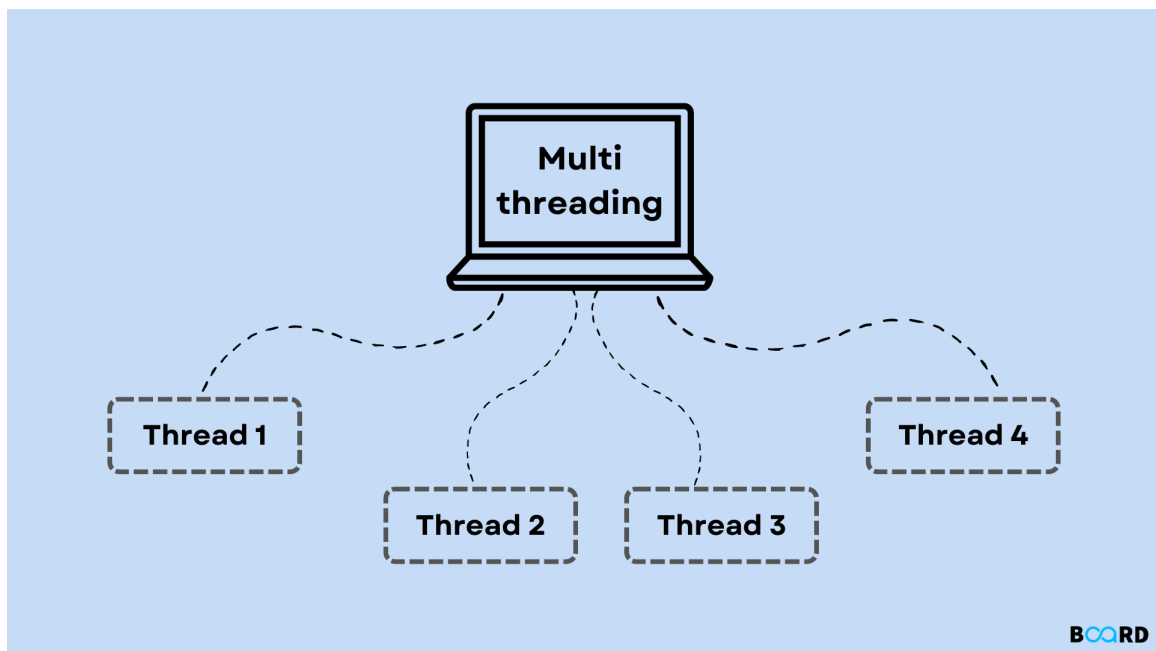


TP IDM

Génération de flux parallèles de nombres pseudo-aléatoires



CHASSAGNOL Rémi

Professeur : HILL David

1^{ER} DÉCEMBRE 2023

Table des matières

1	Introduction	2
2	Utilisation de CLHEP	3
3	Test du générateur	3
4	Calcul de pi	3
4.1	Génération des fichiers de statuts	4
4.2	Calcul séquentiel	4
4.3	Calcul parallèle	5
4.3.1	Utilisation d'un script	5
4.3.2	Utilisation des threads	7
5	Conclusion	10

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 générés aléatoirement. Ensuite, nous traiterons l'exemple simple du calcul du nombre π , d'abord en séquentiel, puis en parallèle.

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 du listing 1.

```
git submodule init
git submodule update
```

Listing 1: Synchronisation 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.

3 Test du générateur

Avant de commencer les calculs, assurons-nous que le générateur produit bien toujours les même 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 la méthode `restoreStatus`. Dans la fonction `question2`, nous générons dans un premier temps des fichiers de statuts, et chaque statuts est séparé par un certain nombre de tirage de nombre 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 même 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 Mersenne 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 temps.

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 rendra la parallélisation inutile.

La génération des fichiers de statuts se fait à l'aide de la fonction `question3`. 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 fonctionnalités 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 `question4` qui réalise plusieurs calculs de π et qui calcule la moyenne des résultats pour avoir une valeur approchée. Cette fonction prend en paramètre le nombre de réplifications du calcul ainsi que le nombre de tirages pour chaque réplification. Pour chaque réplification, 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 le listing 2.

```
void question4(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;
}
```

Listing 2: Fonction `question4`.

La fonction permet aussi de mesurer le temps de calcul en utilisant les macros visibles sur le listing 3. Ces macros permettent d'avoir une approximation du nombre de milli-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.

```
#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
```

Listing 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 en parallélisant les calculs.

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.

La fonction `question5` est une version simplifiée de la fonction `question4` qui ne lance qu'une fois le calcul. La fonction prend aussi en argument le nom du fichier de statut à charger pour initialiser le générateur.

Le script utilisé pour lancer les processus est visible sur le listing 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/`.

```
#!/usr/bin/env bash

set -euo pipefail

checkSetup() {
    # ...
}

checkSetup

# configure output file
outputFile=pi
if [ $# -eq 1 ]; then
    outputFile=$1
fi
mkdir -p out

# compute pi
echo "computing pi"
for i in {0..9}; do
    echo "./build/tp5 ./build/mt3_$i"
    ./build/tp5 "./build/mt3_$i" > "./out/$outputFile$i.out" &
done

# wait and quit
time wait
echo "done"
```

Listing 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).

```
tp5 (master) $ ./compute_pi.sh
computing pi
./build/tp5 ./build/mt3_0
./build/tp5 ./build/mt3_1
./build/tp5 ./build/mt3_2
./build/tp5 ./build/mt3_3
./build/tp5 ./build/mt3_4
./build/tp5 ./build/mt3_5
./build/tp5 ./build/mt3_6
./build/tp5 ./build/mt3_7
./build/tp5 ./build/mt3_8
./build/tp5 ./build/mt3_9

real    0m10,602s
user    1m39,013s
sys     0m0,052s
done
```

FIGURE 1 – Résultat du script lancé sur Ada.

Comme on peut le voir sur la figure 1, le calcul prend environ dix secondes ce qui est bien plus rapide que le temps de calcul séquentiel.

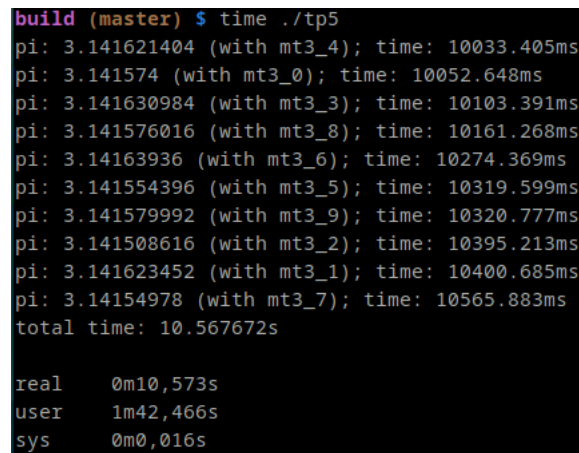
4.3.2 Utilisation des threads

La seconde solution utilise les threads de la bibliothèque standard de C++. Ici, on utilise la fonction `question6a` disponible en deux version, une utilisant directement les threads, et une autre utilisant l'abstraction `std::future`. Les deux version utilisent la fonction `question5` présentée dans la section 4.3.1.

```
void question6aFuture(const std::string &fileName, size_t nbDraw) {
    CLHEP::MTwistEngine mt;
    size_t i;

    timerStart();
    /* threads scope */ {
        std::future<void> pis[NB_REPLICATION];
        for (i = 0; i < NB_REPLICATION; ++i) {
            pis[i] = std::async(std::launch::async, question5, cat(fileName, i),
                               nbDraw);
        }
    }
    timerEnd();
    std::cout << "total time: " << timerCountMs() / 1000.0 << "s" << std::endl;
}
```

Listing 5: Fonction `question6aFuture`.



```
build (master) $ time ./tp5
pi: 3.141621404 (with mt3_4); time: 10033.405ms
pi: 3.141574 (with mt3_0); time: 10052.648ms
pi: 3.141630984 (with mt3_3); time: 10103.391ms
pi: 3.141576016 (with mt3_8); time: 10161.268ms
pi: 3.14163936 (with mt3_6); time: 10274.369ms
pi: 3.141554396 (with mt3_5); time: 10319.599ms
pi: 3.141579992 (with mt3_9); time: 10320.777ms
pi: 3.141508616 (with mt3_2); time: 10395.213ms
pi: 3.141623452 (with mt3_1); time: 10400.685ms
pi: 3.14154978 (with mt3_7); time: 10565.883ms
total time: 10.567672s

real    0m10,573s
user    1m42,466s
sys     0m0,016s
```

FIGURE 2 – Résultat du script lancé sur Ada.

Comme on peut le voir sur la figure 2, 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.

5 Conclusion

TODO