

Rapport TP Ingénierie des modèles TP #5 - Flux stochastiques – modèles du hasard - Nombres quasi-aléatoires - Parallélisme

BALLEJOS Lilian
ISIMA INP Troisième année
Spécialité F2
Année Universitaire 2023/2024

Enseignant référent : Mr David HILL
Date du rendu : 22 décembre 2023

ISIMA INP

1 rue de la Chebarde - TSA 60125 - CS 60026 - 63178 Aubière CEDEX
Tel : 04 73 40 50 00
Site web : isima.fr

Table des matières

1	Introduction	2
2	Compilation de CLHEP	2
2.1	Compilation en Séquentiel	2
2.2	Compilation en parallèle sur plusieurs threads	3
3	Découverte et compréhension de la méthode saveStatus de CLHEP	3
3.1	Explication de l'utilité	3
3.2	Démonstration	3
4	Sauvegarde de status de générateur	4
5	Approximation de PI en séquentiel	5
5.1	Utilité	5
5.2	Résultats	5
5.2.1	Qualité de PI	5
5.2.2	Temps de calcul	6
5.3	Interprétation des résultats	6
6	Approximation de PI en parallèle	6
6.1	Utilité	6
6.2	Résultats	7
6.3	Interprétation des résultats	7
7	Bibliothèques de parallélisation en CPP (pthread)	7
7.1	Explication	7
7.2	Temps de calcul	8
8	Conclusion	8

1 Introduction

Dans ce rapport de TP nous allons nous intéresser à deux concepts :

- Découverte d'une bibliothèque de taille professionnelle pour la simulation (**CLHEP**)
- Parallélisation des expériences afin de minimiser le temps de calcul de nombres quasi-aléatoires

Pour cela, nous allons utiliser la bibliothèque **CLHEP** (*Class Library for High Energy Physics*).

En nous appuyant sur la classe **CLHEP : :MTwistEngine**, nous allons refaire nos expériences effectuées l'année dernière de calcul de pseudo-PI.

Remarque : Tous les calculs ont été effectués sur ma machine personnelle (dont les caractéristiques sont disponibles sur la figure 1) et les temps d'exécution peuvent donc changer durant vos expériences.

```
1 description: CPU
2 produit: AMD Ryzen 5 7530U with Radeon Graphics
3 fabricant: Advanced Micro Devices [AMD]
4 identifiant matériel: 4
5 information bus: cpu@0
6 version: 25.80.0
7 numéro de série: Unknown
8 emplacement: FP6
9 taille: 4416MHz
10 capacité: 4546MHz
11 bits: 64 bits
12 horloge: 100MHz
13 configuration : cores=6 enabledcores=6 microcode=173015053 threads=12
```

FIGURE 1 – Configuration de mon ordinateur personnel

Où trouver le code : Tout le code de mon TP est disponible dans un répertoire public de mon GitLab ISIMA disponible à ce lien : <https://gitlab.isima.fr/liballejos/tp5-idm>

Vous trouverez dans ce répertoire comment compiler et exécuter mon programme ainsi que l'utilité de chaque dossier en lisant les **README.md** présent dans le répertoire.

2 Compilation de CLHEP

Pour avoir une première démonstration de l'utilité des threads dans les longs calculs en informatique, nous allons compiler la bibliothèque **CLHEP** de deux manières différentes.

Premièrement, nous allons compiler la bibliothèque de manière séquentielle, puis nous allons ensuite la recompiler, mais cette fois-ci en parallèle sur plusieurs threads.

2.1 Compilation en Séquentiel

Voici le temps de compilation obtenu après une compilation en séquentiel de la bibliothèque **CLHEP** (figure 2.1) :

```
1 $ time make
2 > make 33,94s user 5,81s system 99% cpu 39,76s total
```

FIGURE 2 – Temps pour une compilation séquentiel de CLHEP

On obtient donc un temps total de compilation de 39,768 secondes. Voyons maintenant obtenu avec une compilation en parallèle sur plusieurs threads.

2.2 Compilation en parallèle sur plusieurs threads

Voici le temps de compilation obtenu en parallèle (figure 4) :

```

1 $ time make -j
2 > make --jobs 104,34s user 12,59s system 1108% cpu 10,552 total

```

FIGURE 3 – Temps pour une compilation parallèle de CLHEP

On obtient donc un temps total de compilation de 10,552 secondes.

En effectuant un simple rapport :

$$\frac{t_{compile_sequentiel}}{t_{compile_parallele}} = \frac{39,768}{10,552} = 3,768764215 \quad (1)$$

On a donc une bonne première démonstration de l'efficacité des processus en parallèle, car on obtient ici un temps de compilation $\sim 3,8$ fois plus rapide en parallèle.

3 Découverte et compréhension de la méthode saveStatus de CLHEP

3.1 Explication de l'utilité

Dans cette question, nous découvrons que l'on peut sauvegarder dans un fichier avec **CLHEP** l'état dans lequel est actuellement un générateur pseudo aléatoire.

Cela nous sera très utile dans le futur pour lancer simultanément plusieurs processus d'une même fonction effectuant les mêmes tirages, mais sur un état différent du générateur pseudo aléatoire, afin de s'assurer de la bonne répartition de nos tirages.

3.2 Démonstration

On effectue donc trois séries de dix tirages en sauvegardant bien, avant chaque série, l'état du générateur pseudo aléatoire. On ré-effectue ensuite trois séries de dix tirages en restaurant l'état du générateur pseudo aléatoire avant chaque série.

On obtient ces résultats :

```
==== Performing draws ====
Performing draw n°0
756416183      87117180
3707664325     510894905
1991254702     265857433
3538995638     2455653411
393949109      3339418352

Performing draw n°1
2323704193     1830345122
120222373      1215289120
1039425486     976099804
1469530379     310884070
191307174      2813423518

Performing draw n°2
414341176      3383781080
3476225207     1970956983
3133160298     4047765271
1226388547     2959552425
1176210078     3086939694
```

FIGURE 4 – Résultats des tirages et enregistrement de l'état du générateur

```
==== Restoring status ====
Restoring status n°0
756416183      87117180
3707664325     510894905
1991254702     265857433
3538995638     2455653411
393949109      3339418352

Restoring status n°1
2323704193     1830345122
120222373      1215289120
1039425486     976099804
1469530379     310884070
191307174      2813423518

Restoring status n°2
414341176      3383781080
3476225207     1970956983
3133160298     4047765271
1226388547     2959552425
1176210078     3086939694
```

FIGURE 5 – Résultats avec restauration de l'état du générateur

En comparant les résultats des tirages sur la figure 4 et 5, on remarque qu'après restauration de l'état du générateur pseudo-aléatoire, on obtient bien les mêmes nombres pseudo aléatoire en sortie. On a donc bien réussi à sauvegarder dans un fichier l'état de notre générateur pseudo aléatoire.

4 Sauvegarde de status de générateur

Nous allons, pour le bien de la suite du TP, enregistrer l'état du générateur pseudo-aléatoire pour 10 séries de 2 000 000 000 tirages, soit un total de 20 000 000 000 de tirages. Cela nous permettra, à la fin du TP, de pouvoir effectuer en parallèle de nombreux tirages pour approximer π .

J'ai effectué plusieurs fois ces tirages pour avoir un temps moyen de calcul de ces nombreux tirages.

```
1 Time used : 74.6731 seconds
2 Time used : 73.1192 seconds
3 Time used : 72.0108 seconds
4 Time used : 74.6765 seconds
5 Time used : 71.0104 seconds
```

FIGURE 6 – Les 5 temps de calcul pour 20 000 000 000 tirages

En faisant une moyenne, on a un temps moyen pour 20 000 000 000 tirages de 73,098 secondes.

5 Approximation de PI en séquentiel

5.1 Utilité

Nous allons maintenant calculer des pseudo-PI de manière séquentielle. Cela nous donnera une idée du temps de calcul nécessaire pour une grosse simulation. Pour cela, nous allons générer 10 pseudo-PI avec la méthode de Monte Carlo et le tirage aléatoire de 1 000 000 000 points, soit le tirage de 2 000 000 000 de nombres pseudo-aléatoires (coordonnés x et y).

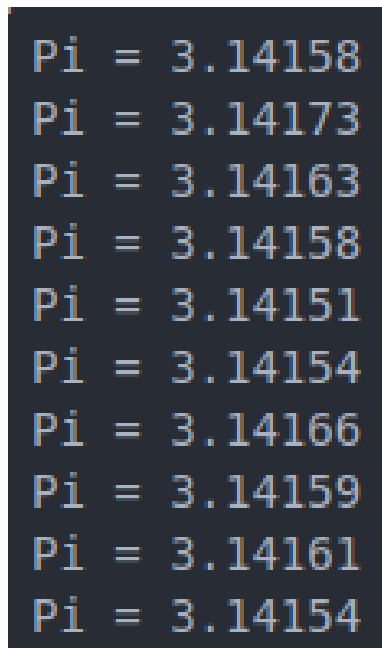
Nous effectuerons une vérification de la qualité des pseudo-PI générés. Ensuite, nous répéterons plusieurs fois l'expérience pour obtenir un temps moyen d'exécution.

5.2 Résultats

J'ai effectué l'expérience cinq fois, c'est-à-dire que j'ai lancé cinq fois mon programme. Étant donné que le générateur pseudo-aléatoire est initialisé de la même manière au démarrage, les dix sorties de PI seront les mêmes. Cependant, répéter l'expérience cinq fois permettra d'obtenir un temps d'exécution moyen.

5.2.1 Qualité de PI

Analysons tout d'abord la qualité des pseudos-PI tirés.



```
Pi = 3.14158
Pi = 3.14173
Pi = 3.14163
Pi = 3.14158
Pi = 3.14151
Pi = 3.14154
Pi = 3.14166
Pi = 3.14159
Pi = 3.14161
Pi = 3.14154
```

FIGURE 7 – Résultats des 10 pseudo-PI générés pseudo Aléatoirement

Sur la figure 7, on voit la valeur des 10 pseudo-PI calculé par mon programme avec la méthode de Monte Carlo et un tirage de 1 000 000 000 de points par pseudo-PI.

En réutilisant les fonctions créées l'année dernière pour calculer la qualité des pseudo-PI générés, nous allons pouvoir calculer la qualité de nos pseudo-PI calculés.

Le code pour effectuer cette vérification de qualité se trouve dans le dossier **PiQualityCalculation** de mon répertoire GitLab.

Voici la sortie obtenue pour nos 10 pseudo-PI générés :

```
Moyenne: 3.141597
Erreur relative: 1.000001
Erreur absolue: -0.000004
Ecart type: 0.000061
Intervalle de confiance à 95% : [3.141551 - 3.141643]
```

FIGURE 8 – Sortie de mon programme de calcul de qualité avec mes Pseudo-PI générés

On peut donc trouver dans la figure 8 une moyenne de ces 10 pseudo-PI qui vaut $\bar{x} = 3.141597$, puis l'écart type qui est de $\sigma = 6.1 * 10^{-5}$.

De plus, on a une erreur relative de 1.000001 et une erreur absolue de -0.000004

Enfin, on peut en tirer cet intervalle de confiance à 95% : $[3.141551, 3.141643]$.

5.2.2 Temps de calcul

Analysons maintenant le temps de calcul.

```
Time used : 105.353 seconds
Time used : 105.667 seconds
Time used : 111.796 seconds
Time used : 109.122 seconds
Time used : 105.405 seconds
```

FIGURE 9 – Les temps pour calculer 10 pseudo-PI avec Monte Carlo de manière séquentielle

Dans la figure 9, on a le temps d'exécution des 5 itérations d'expérience que j'ai lancé.

On obtient un temps moyen d'exécution de 107.268 secondes.

5.3 Interprétation des résultats

Nous remarquons dans cette question que le temps d'exécution pour calculer un pseudo-PI d'une qualité discutable peut être long dû à un tirage aléatoire de 20 000 000 de nombres pseudo-aléatoires.

Maintenant que nous avons enregistré dans la section 4 différents états de notre générateur pseudo-aléatoire, nous allons essayer de voir s'il est possible de calculer ces différents pseudo-PI plus rapidement en lançant les dix calculs du pseudo-PI en parallèle.

6 Approximation de PI en parallèle

6.1 Utilité

Voyons une première façon de calculer en parallèle 10 pseudo-PI.

Nous allons, pour commencer, nous appuyer sur un script *bash* (**question5.sh**) pour lancer 10 fois la fonction CPP que j'avais créée, qui calcule un pseudo-PI avec 1 milliard de points et la méthode de Monte Carlo (soit 2 milliards de tirages pseudo aléatoires).

L'avantage est que dans la section 4, nous avons déjà enregistré 10 états du générateur pseudo-aléatoire pour être sûr d'avoir 10 pseudo-PI différents malgré le lancement en parallèle du programme.

6.2 Résultats

Etant donné que nous avons enregistré les différents états de notre générateur pseudo-aléatoire de manière séquentielle et donc exactement comme lorsqu'on a effectué les tirages dans la section 5, nous n'aurons pas besoin de vérifier la qualité des pseudo-PI générés pour cette question, car ce sont les mêmes que ceux calculés dans la question précédente.

Penchons-nous maintenant sur le temps de calcul nécessaire. J'ai lancé cinq fois mon script *bash*, et voici le temps de calcul de chaque exécution.

```
All replications have been finished. Total time elapsed: 23 seconds.
All replications have been finished. Total time elapsed: 23 seconds.
All replications have been finished. Total time elapsed: 22 seconds.
All replications have been finished. Total time elapsed: 22 seconds.
All replications have been finished. Total time elapsed: 22 seconds.
```

FIGURE 10 – Les temps pour calculer 10 pseudo-PI avec Monte Carlo en parallèle

En faisant une moyenne des temps disponible sur la figure 10, on obtient un temps de calcul moyen pour la génération de 10 pseudo-PI avec la méthode de Monte Carlo (avec un tirage d'un milliard de points) de 22,6 secondes.

6.3 Interprétation des résultats

Nous avons maintenant le temps de calcul en séquentielle et en parallèle pour la même expérience.

Faisons un simple rapport pour comparer les temps de calcul :

$$\frac{t_{calcul_sequentiel}}{t_{calcul_parallele}} = \frac{107.268}{22,6} \approx 4.746 \quad (2)$$

On est donc environ 4.7 fois plus rapide pour calculer des pseudo-PI lorsqu'on le fait en parallèle plutôt qu'en séquentiel pour la même expérience !

Cela met en lumière l'importance de l'utilité des threads et du calcul en parallèle pour certains calculs très coûteux. Cet exemple nous montre bien ici la différence entre ces deux façons de calculer.

7 Bibliothèques de parallélisation en CPP (pthread)

7.1 Explication

J'ai voulu directement inclure le principe de parallélisation de calcul dans mon programme Cpp en utilisant une bibliothèque de parallélisation standard qui est **pthread**.

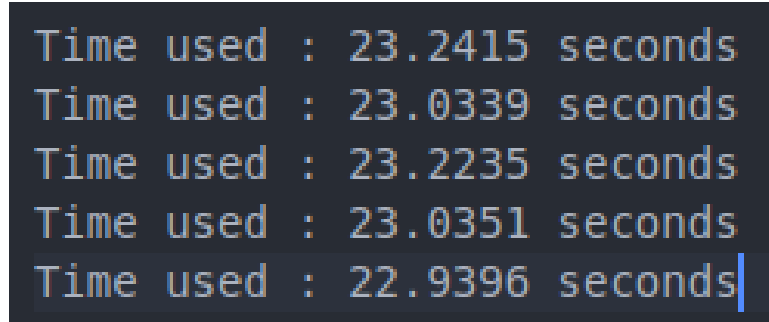
Le principe est le suivant : on crée 10 threads et on les lance en parallèle en tâche de fond, puis on attend que ces 10 threads aient fini pour afficher le temps de calcul écoulé.

On ne peut pas utiliser **clock_t**, disponible en CPP, pour calculer le temps écoulé en parallèle car il additionne le temps écoulé dans les 10 threads, et on obtient donc un temps en fin de programme proche d'un calcul en séquentiel.

J'ai donc utilisé **<chrono>** qui peut calculer seulement le temps écoulé entre l'acquisition de deux temps. Je récupère donc le temps avant l'instanciation des threads et à la fin de l'exécution de tous les threads pour trouver le temps écoulé.

7.2 Temps de calcul

J'ai lancé cinq fois mon programme Cpp pour avoir un temps moyen d'exécution. Voici les cinq temps obtenus :



```
Time used : 23.2415 seconds
Time used : 23.0339 seconds
Time used : 23.2235 seconds
Time used : 23.0351 seconds
Time used : 22.9396 seconds
```

FIGURE 11 – Les temps pour calculer 10 pseudo-PI avec Monte Carlo en parallèle avec pthread

En faisant la moyenne des résultats obtenus sur la figure 11, on obtient un temps moyen d'utilisation de 23.09472 secondes.

Si l'on calcule une nouvelle fois le rapport entre le temps de calcul en séquentiel et parallèle, on obtient :

$$\frac{t_{calcul_sequentiel}}{t_{calcul_parallele}} = \frac{107.268}{23.09472} \approx 4.64 \quad (3)$$

On a donc un temps d'exécution 4.64 fois plus rapide en parallèle !

8 Conclusion

La réalisation de ce TP sur les flux stochastiques, les modèles du hasard, les nombres quasi-aléatoires et le parallélisme nous a offert l'opportunité d'explorer plusieurs concepts clés en informatique et en simulation. Nous avons utilisé la bibliothèque **CLHEP** pour la gestion des nombres pseudo-aléatoires et étudié la compilation en parallèle afin d'améliorer les performances.

En prenant en main la bibliothèque de calcul scientifique **CLHEP**, nous avons bénéficié de générateurs pseudo-aléatoires de grande qualité, ainsi que de fonctionnalités telles que la sauvegarde d'état d'un générateur pseudo-aléatoire, particulièrement utile pour les calculs en parallèle.

L'expérience a permis de mettre en évidence l'utilité de la parallélisation des calculs, surtout pour des tâches répétitives et coûteuses en temps. L'utilisation de scripts *bash* ou d'une bibliothèque de thread directement dans le code (*pthread*) pour lancer simultanément plusieurs expériences a significativement réduit les temps de calcul, démontrant ainsi l'efficacité du parallélisme dans des contextes de calcul intensif.

Table des figures

1	Configuration de mon ordinateur personnel	2
2	Temps pour une compilation séquentiel de CLHEP	2
3	Temps pour une compilation parallèle de CLHEP	3
4	Résultats des tirages et enregistrement de l'état du générateur	4
5	Résultats avec restauration de l'état du générateur	4
6	Les 5 temps de calcul pour 20 000 000 000 tirages	4
7	Résultats des 10 pseudo-PI générés pseudo Aléatoirement	5
8	Sortie de mon programme de calcul de qualité avec mes Pseudo-PI générés . . .	6
9	Les temps pour calculer 10 pseudo-PI avec Monte Carlo de manière séquentielle	6
10	Les temps pour calculer 10 pseudo-PI avec Monte Carlo en parallèle	7
11	Les temps pour calculer 10 pseudo-PI avec Monte Carlo en parallèle avec pthread	8