Calculs de Monte Carlo à l'aide de la bibliothèque CLHEP

Calculer le volume d'une sphère de rayon 1

Ao XIE
Ingénierie des modèles et Simulation
Institut Supérieur d'Informatique, de Modélisation et de leurs Applications
22 December 2024

Introduction

La méthode de Monte Carlo est une méthode de calcul numérique basée sur l'échantillonnage aléatoire [1], qui est largement utilisée dans les domaines de la physique, de l'ingénierie et de l'informatique [2], en particulier pour les problèmes qui sont difficiles à résoudre directement par des méthodes analytiques. Dans cette expérience, la méthode de Monte Carlo est utilisée pour calculer le volume d'une sphère de rayon 1 et comparer les performances des méthodes de calcul à monothread, de calcul à multithreading et de calcul parallèle basé sur Open MP [3].

La bibliothèque CLHEP (Class Library for High Energy Physics) [4], qui fournit un support riche pour la génération de nombres aléatoires et le calcul scientifique, a été utilisée dans cette expérience pour fournir des nombres aléatoires reproductibles.

Ce rapport décrit en détail l'environnement expérimental et sa construction, le processus expérimental et les résultats finaux de cette expérience. L'analyse vise à expliquer les avantages et les limites des méthodes de calcul infranchissables dans la simulation scientifique, et à fournir une référence pour les cours ultérieurs sur le calcul parallèle.

Environnement expérimental et sa construction

Cette section est un précurseur de l'expérience, c'est-à-dire qu'elle décrit l'environnement dans lequel l'expérience sera menée et le processus de construction de cet environnement.

1. Environnement expérimental

Cette expérience a été réalisée sur un MacBook Pro 2023 équipé d'une puce Apple M2 Pro (CPU 10 cœurs, GPU 16 cœurs) et de 16 Go de RAM, fonctionnant sous macOS Sequoia 15.2.

Au niveau de l'environnement logiciel de cette expérience a utilisé la GNU Compiler Collection (GCC) installée via Homebrew, version 14.2.0. La norme C++ 2017 a également été utilisée tout au long de l'expérience.

2. Construction de l'environnement

Sur la base de l'environnement expérimental décrit ci-dessus, la présente sous-section s'attache principalement à décrire l'installation du code source de la bibliothèque CLHEP et la manière dont elle sera utilisée au cours de l'expérience finale.

Tout d'abord, le code source a été installé dans cette expérience conformément au manuel de laboratoire. Les bibliothèques sous l'architecture ARM64 ont été complètement installées dans l'expérience, et leurs fichiers de bibliothèques statiques et dynamiques générés sont montrés dans la Figure 1.

```
┌─[~/Code/ZZ3/Simu/TP5/Random]─[neil@MBP-NEIL]─[0]─[10236]
└─[:)] % lipo -info lib/libCLHEP-Random-2.1.0.0.a
Non-fat file: lib/libCLHEP-Random-2.1.0.0.a is architecture: arm64
┌─[~/Code/ZZ3/Simu/TP5/Random]─[neil@MBP-NEIL]─[0]─[10237]
└─[:)] % ls -lh lib
total 21144
-rw-r--r--@ 1 neil  staff   7.8M Dec 21 15:06 libCLHEP-Random-2.1.0.0.a
-rwxr-xr-x@ 1 neil  staff   2.5M Oct  3  2005 libCLHEP-Random-2.1.0.0.so
```

Figure 1 - Vérification de l'installation du code source

Après avoir terminé l'installation du code source et vérifié sa disponibilité, l'expérience a abandonné l'installation de la bibliothèque CLHEP à l'aide du code source. La raison en est que Homebrew peut fournir un moyen propre et facile à gérer d'installer des bibliothèques, alors que l'installation à partir du code source peut augmenter considérablement la taille d'un projet si celui-ci souhaite conserver sa méthode de génération. La bibliothèque installée est illustrée à la figure 2.

```
==> clhep: stable 2.4.7.1 (bottled), HEAD
Class Library for High Energy Physics
https://proj-clhep.web.cern.ch/proj-clhep/
Installed
/opt/homebrew/Cellar/clhep/2.4.7.1 (349 files, 9.4MB) *
  Poured from bottle using the formulae.brew.sh API on 2024-12-21 at 19:19:15
From: https://github.com/Homebrew/homebrew-core/blob/HEAD/Formula/c/clhep.rb
License: GPL-3.0-only
==> Dependencies
Build: cmake ✔
==> Options
--HEAD
        Install HEAD version
==> Analytics
install: 10 (30 days), 35 (90 days), 153 (365 days)
install-on-request: 10 (30 days), 35 (90 days), 153 (365 days)
build-error: 0 (30 days)
```

Figure 2 - Bibliothèque CLHEP installée par Homebrew

En résumé, tous les travaux de l'expérience finale ont utilisé la bibliothèque CLHEP installée par Homebrew, version 2.4.7.1.

3. Vérifier l'installation

Pour vérifier le succès de l'installation, une session de validation a été incluse dans l'expérience. Pour compléter la validation, la bibliothèque

CLHEP a été utilisée dans l'expérience pour générer 10 nombres aléatoires à l'aide d'un état et les écrire dans un fichier bin, puis lire ce fichier bin et cet état, régénérer les 10 nombres aléatoires et les comparer. Les résultats de la comparaison finale sont présentés à la figure 3.

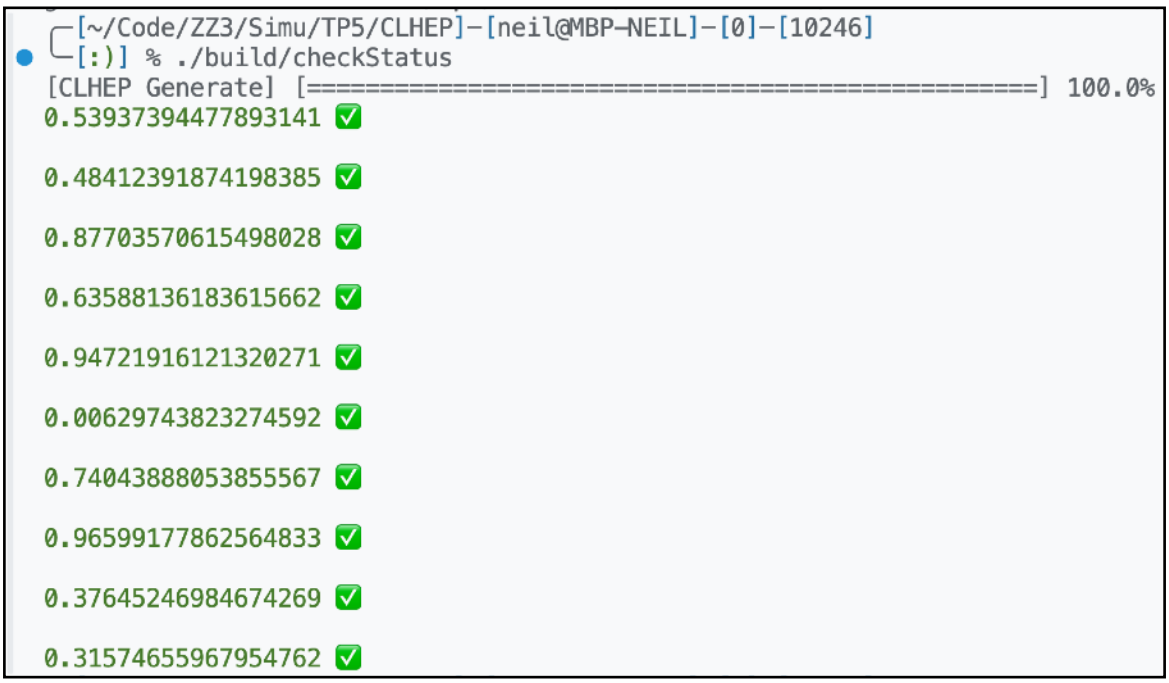

Figure 3 - Vérifier l'installation en utilisant 10 nombres aléatoires

Il convient de noter qu'en raison de la perte de précision des nombres à virgule flottante dans la pratique (comme indiqué dans le TP6 [5, 6]), les expériences ont utilisé 17 bits pour le stockage, la lecture et la validation, c'est-à-dire plus que la précision décimale des nombres à virgule flottante en double précision (15 à 16 bits).

Procédure expérimentale

Après l'installation et la validation de l'environnement ci-dessus, cette section est consacrée au processus d'expérimentation.

Toutes les expériences effectuent dix opérations de Monte Carlo, chaque opération de Monte Carlo utilisant deux milliards de points pour le calcul.

Le concept de traitement par lots a été introduit au cours des expériences, c'est-à-dire que chaque calcul de Monte Carlo a été regroupé en fonction du nombre, et le principal effet de cette approche a été de réduire l'utilisation de la mémoire pendant l'exécution du programme. Toutefois, le traitement par lots ne prend pas en charge le multithreading et s'applique à tous les calculs expérimentaux, de sorte qu'il n'affecte pas les résultats de performance de cette expérience.

Opération Monte Carlo Monothread

Dans cette étape expérimentale, l'expérience n'effectue aucune gestion des threads et surveille le nombre de threads qu'elle utilise par l'intermédiaire de l'outil de surveillance du système.

L'expérience a commencé par une étape de validation des performances, au cours de laquelle le nombre de threads utilisés et l'occupation du système ont été déterminés par la surveillance du système pendant l'exécution du programme.Pour chaque cœur de CPU, le taux d'occupation est indiqué dans la figure 4 et le nombre de threads utilisés est indiqué dans la figure 5.

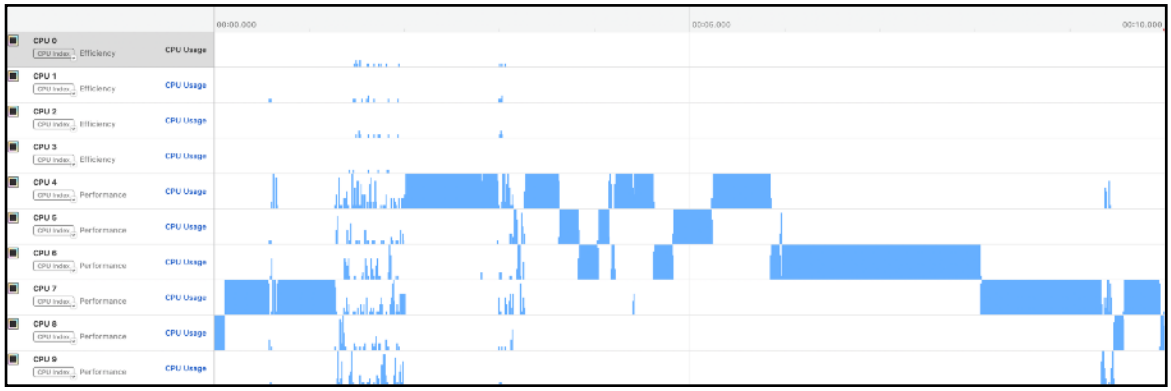

Figure 4 - Utilisation du CPU pendant l'exécution

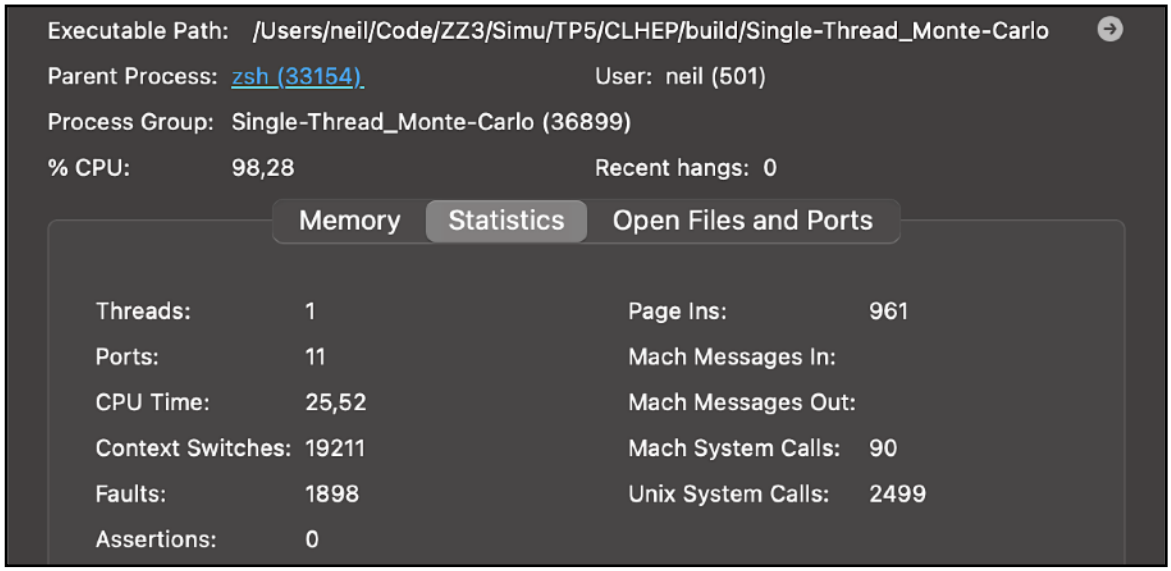

Figure 5 - Détails du programme

Après la validation, le programme a été exécuté cinq fois au total et la durée d'exécution de chaque programme est indiquée dans le tableau 1.

Tableau 1 - Exécution à un seul thread

	Résultat du calcul	Temps utilisé(s)
1	4.1888288984000415	444.659401
2	4.1888288984000415	444.41460599999999
3	4.1888288984000415	445.813219
4	4.1888288984000415	443.738271
5	4.1888288984000415	446.19775700000002
Avg	**4.1888288984000415**	**444.9646**

Opérations de Monte Carlo Multithreads

Pour les opérations multithread, les expériences utilisent directement le mécanisme multithread de C++ en plaçant chaque opération de Monte Carlo dans un thread séparé, c'est-à-dire que dix nouveaux threads sont ajoutés pour le calcul des résultats.

Dans ce cas, d'après la surveillance du système, le programme multithread occupe un total de 11 threads, et son occupation de l'unité centrale est illustrée à la figure 6.

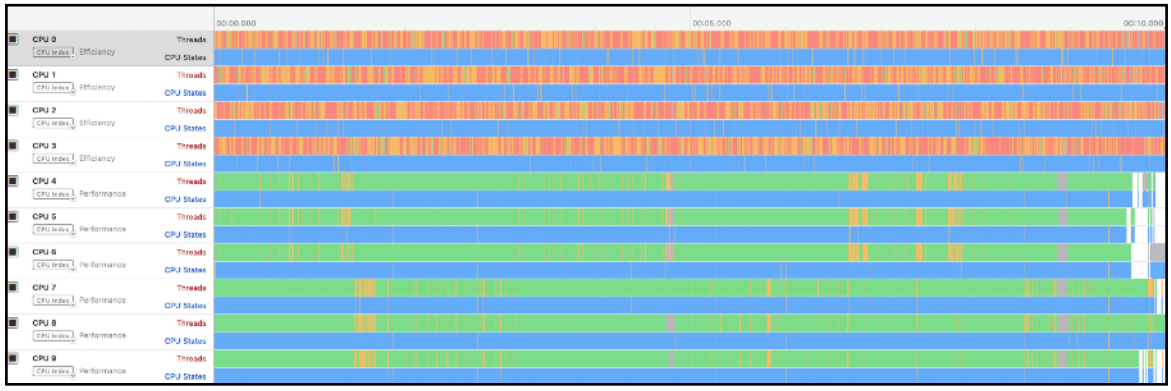

Figure 6 - Utilisation du CPU pendant l'exécution Multithreads

De même, le programme a été exécuté cinq fois au cours de l'expérience et sa durée d'exécution ainsi que les résultats sont indiqués dans le tableau 2.

Tableau 2 - Exécution à Multithreads

	Résultat du calcul	Temps utilisé(s)
1	4.1888410676000412	59.9718900000000002
2	4.1888410676000412	59.613900999999998
3	4.1888410676000412	58.900995000000002
4	4.1888410676000412	58.456543000000003
5	4.1888410676000412	59.054395999999997
Avg	4.1888410676000412	**59.1995**

Opération Monte Carlo OpenMP

En ce qui concerne l'utilisation d'OpenMP, 10 calculs Monte Carlo ont été directement confiés à OpenMP pour être gérés lors des expériences, c'est-à-dire qu'OpenMP a été utilisé pour gérer automatiquement le système. Dans ce cas, l'utilisation des ressources du système est illustrée à la figure 7.

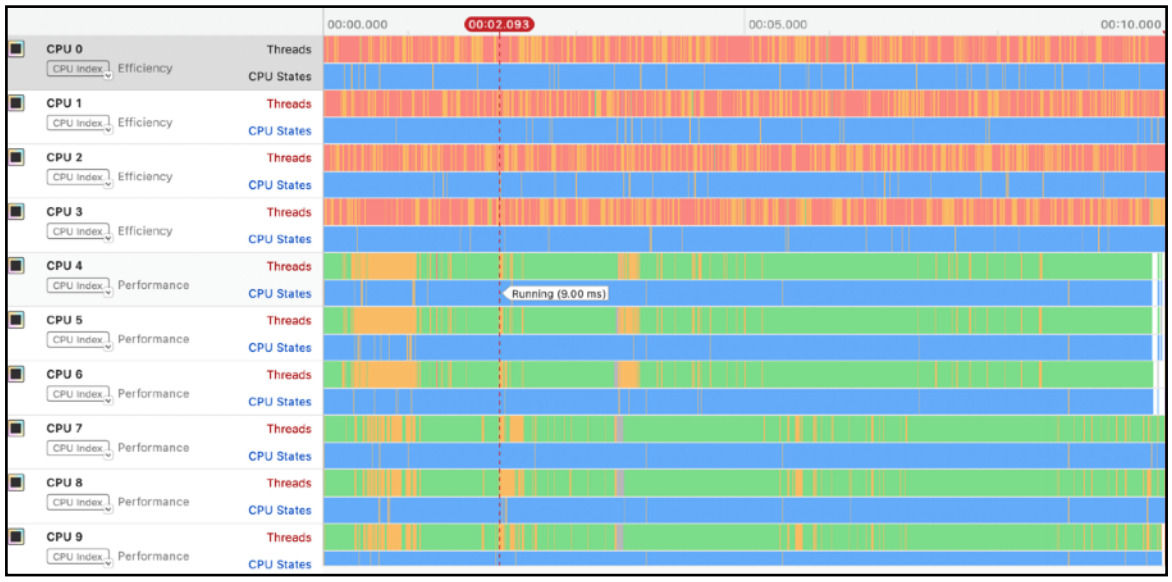

Figure 7 - Utilisation du CPU pendant l'exécution OpenMP

De même, le programme a été exécuté cinq fois au cours de l'expérience et sa durée d'exécution ainsi que les résultats sont indiqués dans le tableau 3.

Tableau 2 - Exécution à OpenMP

	Résultat du calcul	Temps utilisé(s)
1	4.1888288984000415	61.0807
2	4.1888288984000415	58.994428999999997
3	4.1888288984000415	60.303753999999998
4	4.1888288984000415	60.373646999999998
5	4.1888288984000415	59.392418999999997
Avg	4.1888288984000415	60,0290

Résultats et analyses

Cette section est consacrée à l'analyse et à la synthèse des résultats de l'expérience après l'obtention des résultats ci-dessus, ainsi qu'à la description du suivi expérimental.

1. Résultats du calcul de volume

Pour le volume de toutes les sphères, la formule est présentée dans l'équation 1.

$$V = \frac{4}{3}\pi r^3 \quad (1)$$

À l'aide de cette formule, nous pouvons calculer que le volume d'une sphère de rayon 1 est d'environ 4,1887902, c'est-à-dire que les trois méthodes d'arithmétique aboutissent à des résultats que l'on peut considérer comme approximativement corrects.

2. Monothread VS Multithreads

En ce qui concerne les performances du programme, l'algorithme monothread prend en moyenne 445 secondes dans l'environnement matériel et logiciel actuel, tandis que les algorithmes multithreads (OpenMP et multithreads) prennent environ 60 secondes. L'exemple d'accélération multithreads disponible selon la loi d'Amdahl [7] est présenté dans l'équation 2.

$$S = \frac{1}{f + \frac{1-f}{P}} \quad (2)$$

Dans cette équation, S est le taux d'accélération du programme, f est la proportion de la partie non parallélisable du programme et P est le nombre de threads utilisés. Dans le code expérimental, seul le temps d'exécution de

l'algorithme de Monte Carlo a été mesuré. Par conséquent, en théorie, la partie parallélisable du programme est de 100 %. Lorsque l'on utilise OpenMP pour créer 10 threads afin d'effectuer une tâche de calcul, la durée d'exécution d'un programme multithread devrait théoriquement être 10 fois plus rapide que celle d'un programme à un seul thread. Cependant, les mesures réelles montrent une accélération d'environ 7,4 fois seulement. Cela suggère que la performance est quelque peu limitée par le matériel, en particulier par la différence de performance entre le cœur de performance (P-core) et le cœur économe en énergie (E-core).

D'autres mesures montrent qu'au cours d'un programme multithread, les cœurs économes en énergie tournent à environ 2,4 GHz, tandis que les cœurs performants tournent à 3,5 GHz. L'appareil contient 6 cœurs performants et 4 cœurs économes en énergie. Les programmes monothématiques ont été exécutés sur les cœurs de performance (CPU4 - CPU9), en tirant parti de leur plus grande puissance de calcul. Théoriquement, hors impact du cœur du processeur, la vitesse peut être augmentée jusqu'à 8,5 fois.

3. Différentes implémentations multithreading

Lorsque l'on compare les threads de contrôle manuel C++ et OpenMP, il apparaît clairement que le temps de calcul utilisé par les deux est extrêmement similaire. La différence réside dans le fait que la gestion manuelle utilise une approche plus complexe et crée 11 threads, alors qu'OpenMP utilise automatiquement le maximum de 10 threads que le système peut fournir de manière plus simple.

4. Améliorations ultérieures possibles

En termes de calcul, cette expérience a testé le multithreading et OpenMP, mais pour un grand nombre de tâches simples, les avantages du GPU seraient plus évidents. En effet, les GPU sont conçus avec un grand nombre de cœurs de calcul (par exemple, les cœurs CUDA [8] ou les unités de calcul Metal) qui peuvent traiter des tâches hautement parallèles. Deuxièmement, les GPU utilisent des modèles de données multiples d'ordre unique (SIMD) [8] et une mémoire à large bande passante pour augmenter le débit tout en améliorant les vitesses d'accès.

En ce qui concerne l'environnement expérimental, l'environnement utilisé dans cette expérience contenait d'autres logiciels en cours d'exécution, ce qui a affecté la précision des résultats dans une certaine mesure. Dans la suite du processus, nous pouvons envisager d'utiliser, par exemple, Docker pour isoler l'environnement expérimental.

Bibliography

[1] N. Metropolis and S. Ulam, "The monte carlo method," *Journal of the American statistical association,* vol. 44, no. 247, pp. 335-341, 1949.

[2] M. H. Kalos and P. A. Whitlock, *Monte carlo methods*. John Wiley & Sons, 2009.

[3] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering,* vol. 5, no. 1, pp. 46-55, 1998.

[4] L. Lönnblad, "CLHEP—a project for designing a C++ class library for high energy physics," *Computer Physics Communications,* vol. 84, no. 1-3, 1994/11/01, doi: 10.1016/0010-4655(94)90217-8.

[5] "754-2019 - IEEE Standard for Floating-Point Arithmetic | IEEE Standard | IEEE Xplore," doi: 10.1109/IEEESTD.2019.8766229.

[6] GoldbergDavid, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR),* vol. 23, no. 1, 1991-03-01, doi: 10.1145/103162.103163.

[7] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483-485.

[8] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing | IEEE Conference Publication | IEEE Xplore," doi: 10.1109/ISBI.2008.4541126.