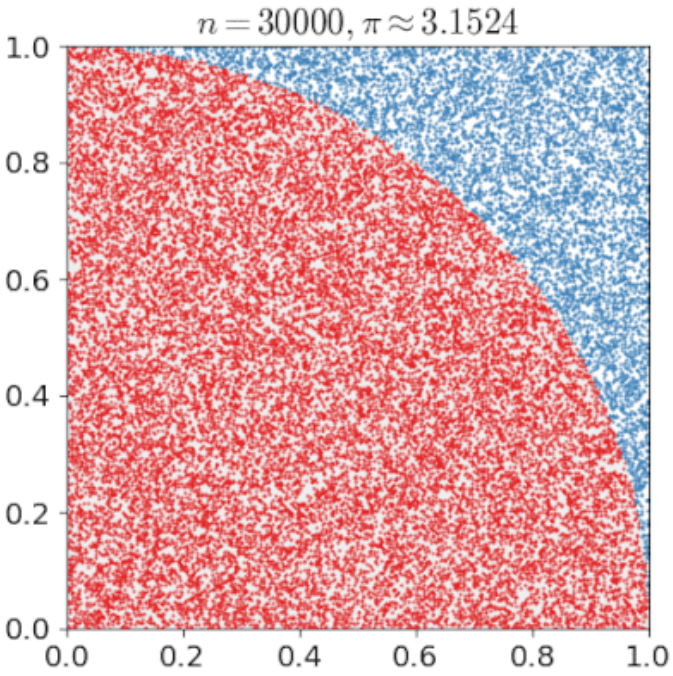
# Rapport TP3 de Simulation

**Monte Carlo Simulation & Confidence Intervals**

**ZZ2 2023-2024**




*Auteur :*
ROUBILLE Mathis

*Rendu à :*
HILL David

20 décembre 2024

# Table des matières

**1 Questions** **3**
1.1 Question 2&3 : . . . . . 3
1.2 Question 4&5 : . . . . . 4
1.3 Question 6 : . . . . . 7

**2 Conclusion** **13**

# Table des codes

1 Préparation des États Aléatoires . . . . . 3
2 Monte Carlo Volume . . . . . 4
3 Recherche Séquentielle de Séquence Cible . . . . . 8
4 Recherche Parallèle de Séquence Cible . . . . . 9

# Introduction

La simulation stochastique est un outil essentiel pour modéliser des phénomènes aléatoires dans des contextes variés, allant des sciences fondamentales à des applications pratiques. Ce travail pratique vise à approfondir la compréhension des défis liés à l'utilisation de l'aléatoire dans des calculs parallèles et séquentiels, en explorant les limitations, les difficultés et les avantages de ces deux approches.

Dans ce TP, deux exercices principaux sont abordés. Le premier met en œuvre des simulations utilisant la méthode de Monte Carlo pour illustrer la parallélisation et la gestion de flux pseudo-aléatoires. Le second exercice s'intéresse à la génération aléatoire de chaînes de nucléotides, avec l'objectif de reconstituer une séquence cible. Ce problème met en évidence les contraintes liées à l'aléatoire, telles que la difficulté d'obtenir un résultat spécifique, et illustre les gains possibles en performance grâce à une approche parallèle.

Un point central de ce travail est l'importance de la reproductibilité. Que ce soit pour garantir la validité des données scientifiques ou pour faciliter le débogage des programmes, la capacité à reproduire exactement une séquence aléatoire est un critère essentiel. Cette reproductibilité est assurée ici par la gestion précise des statuts des générateurs pseudo-aléatoires, notamment en utilisant le générateur *Mersenne Twister*.

Enfin, ce TP met l'accent sur la comparaison entre les approches séquentielles et parallèles dans des contextes impliquant de l'aléatoire, en évaluant les performances, les contraintes et les bénéfices de chaque méthode. En se confrontant à des outils professionnels comme la bibliothèque `CLHEP`, ce travail vise à fournir une vision complète des techniques modernes utilisées pour relever ces défis dans des environnements scientifiques et techniques.

# 1 Questions

## 1.1 Question 2&3 :

Principe de l'algorithme

Cet algorithme prégénère et sauvegarde les états du générateur de nombres pseudo-aléatoires Mersenne Twister (*CLHEP : :MTwistEngine*) dans des fichiers. Chaque état est sauvegardé après un grand nombre de générations aléatoires, garantissant ainsi une progression significative dans la séquence aléatoire. Voici le fonctionnement et l'intérêt de cette méthode :

**Fonctionnement**

1. Génération progressive : Le générateur effectue tirages_par_sauvegarde (2 milliards) de tirages aléatoires entre chaque sauvegarde.
2. Sauvegarde de l'état : Après chaque lot de tirages, l'état interne du générateur est sauvegardé dans un fichier dédié (status_X.txt).
3. Itération multiple : Cela est répété pour nombre_de_statuts (400) états différents.

**Intérêt de la prégénération d'états**

1. Indépendance des états : En prégénérant des états par progression dans la séquence, on garantit que chaque état sauvegardé est éloigné des précédents. Cela réduit les risques de corrélation entre les états utilisés dans des simulations parallèles.
2. Évite le chevauchement : Contrairement à l'utilisation de N graines différentes prises au hasard, où il existe un risque de chevauchement ou de faible diversité statistique entre les séquences, cette méthode produit des séquences non corrélées grâce à l'uniformité intrinsèque du Mersenne Twister.
3. Robustesse pour des simulations longues : Pour des simulations nécessitant un grand nombre de générations (par exemple, 1 milliard), la prégénération garantit que chaque état initialisé est adapté à une longue utilisation sans interférence entre threads.
4. Optimisation pour calculs parallèles : Lors de calculs parallèles, chaque thread peut être initialisé avec un état distinct prégénéré. Cela élimine la nécessité de synchroniser les générateurs ou de se préoccuper de leur indépendance pendant l'exécution.

**Pourquoi éviter les graines au hasard ?**

1. Risque de recouvrement : Deux graines "différentes" peuvent générer des séquences aléatoires qui se recoupent ou qui sont statistiquement similaires, surtout si la graine aléatoire est dérivée de l'horloge système.
2. Non-uniformité : Les graines peuvent introduire des biais ou des dépendances non désirées.
3. Moins prédictible : Les simulations nécessitent souvent des séquences reproductibles pour des comparaisons ou des débogages. Avec des graines prégénérées, il est possible de garantir cette reproductibilité.

En résumé, prégénérer les états permet d'assurer des simulations indépendantes, reproductibles et robustes, même dans des contextes de calcul intensif ou parallèle, où la qualité et l'indépendance des nombres aléatoires sont essentielles.

Préparation des États Aléatoires

```cpp
int main()
{
    CLHEP::MTwistEngine generator;

    const long long tirages_par_sauvegarde = 2000000000LL;
    const int nombre_de_statuts = 400;

    for (int i = 0; i < nombre_de_statuts; ++i)
    {
        for (long long j = 0; j < tirages_par_sauvegarde; ++j)
        {
            generator.flat();
        }

        std::string filename = "save/status_" + std::to_string(i) + ".txt";
        generator.saveStatus(filename.c_str());
        std::cout << "Statut sauvegard  dans " << filename << " apr s "
                  << (i + 1) * tirages_par_sauvegarde << " tirages.\n";
    }
```

## 1.2 Question 4&5 :

Principe de l'algorithme

Cet algorithme estime le volume d'une sphère inscrite dans un cube de côté 2, centré sur l'origine, via la méthode de Monte-Carlo :

1. Génération aléatoire : Tire un grand nombre de points uniformément dans le cube $[1, 1]^3$.
2. Test d'appartenance : Vérifie si chaque point satisfait $x^2 + y^2 + z^2 \leq 1$ (dans la sphère).
3. Calcul du volume : Estime le volume par $V = 8 * (points\_dans\_la\_sphre/points\_totaux)$

monte_carlo_volume

```cpp
/** ---------------------------------------------------------------------- *
 * monte_carlo_volume                                                      *
 *                                                                         *
 * @brief  Calcule une approximation du volume d'une sph re en utilisant  *
 *         la m thode de Monte-Carlo.                                      *
 *                                                                         *
 * @param  generator : moteur de g n ration al atoire                      *
 * @param  points : nombre de points    tirer                              *
 *                                                                         *
 * @return Volume approxim  de la sph re                                   *
 * ---------------------------------------------------------------------- **/
double monte_carlo_volume(CLHEP::MTwistEngine &generator, long long points)
{
    long long inside = 0;

    for (long long i = 0; i < points; ++i)
    {
        double x = generator.flat() * 2.0 - 1.0;
        double y = generator.flat() * 2.0 - 1.0;
        double z = generator.flat() * 2.0 - 1.0;

        if (x * x + y * y + z * z <= 1.0)
        {
            ++inside;
        }
    }

    return 8.0 * static_cast<double>(inside) / points;
}
```

Sortie Question 4 Séquentiel

Réplication 1 : Volume approximé = 4.18891 (temps = 13.2404 secondes)
Réplication 2 : Volume approximé = 4.18869 (temps = 13.0931 secondes)
Réplication 3 : Volume approximé = 4.18884 (temps = 13.0516 secondes)
Réplication 4 : Volume approximé = 4.18847 (temps = 13.0252 secondes)
Réplication 5 : Volume approximé = 4.18873 (temps = 13.051 secondes)
Réplication 6 : Volume approximé = 4.18875 (temps = 13.0162 secondes)
Réplication 7 : Volume approximé = 4.18889 (temps = 12.9817 secondes)
Réplication 8 : Volume approximé = 4.18889 (temps = 13.0281 secondes)
Réplication 9 : Volume approximé = 4.18871 (temps = 13.0269 secondes)
Réplication 10 : Volume approximé = 4.18899 (temps = 13.0413 secondes)

Temps total de calcul : 130.589 secondes.

Sortie Question 5 Parallèle

Réplication 1 : Volume approximé = 4.18891 (temps = 28.0169 secondes)
Réplication 8 : Volume approximé = 4.18889 (temps = 28.115 secondes)
Réplication 10 : Volume approximé = 4.18899 (temps = 28.1157 secondes)
Réplication 9 : Volume approximé = 4.18871 (temps = 28.1272 secondes)
Réplication 3 : Volume approximé = 4.18884 (temps = 28.2294 secondes)
Réplication 5 : Volume approximé = 4.18873 (temps = 28.2445 secondes)
Réplication 7 : Volume approximé = 4.18889 (temps = 28.2728 secondes)
Réplication 6 : Volume approximé = 4.18875 (temps = 28.3064 secondes)
Réplication 2 : Volume approximé = 4.18869 (temps = 15.3905 secondes)
Réplication 4 : Volume approximé = 4.18847 (temps = 15.294 secondes)

Temps total de calcul parallèle : 43.53 secondes.

Analyse des Résultats de Simulation

**Analyse des résultats**

1. Temps séquentiel : Le temps total pour 10 réplications sans threads est de 130 secondes, soit environ 13 secondes par réplication.
2. Temps parallèle : Avec 8 threads, le temps total est de 43 secondes, ce qui représente une accélération par un facteur de 2.86. Cependant, ce résultat est supérieur à l'estimation intuitive de 26 secondes (13s x 2 groupes de threads).

**Causes de l'écart observé**

1. Temps de création et de lancement des threads : Initialiser et démarrer les threads a un coût non négligeable. Ce coût devient proportionnellement moins significatif pour des tâches longues, mais ici, il impacte fortement car chaque réplication est relativement courte.
2. Non-optimalité des ressources : Avec 10 réplications réparties sur 8 threads, certains threads doivent attendre la fin d'autres pour s'exécuter. Utiliser un multiple de 8 (par exemple 16 ou 24) aurait permis de mieux exploiter les capacités du CPU.
3. Surcharge parallèle : La gestion parallèle (synchronisation, communication, allocation de mémoire, etc.) ajoute une surcharge, qui peut dépasser les bénéfices si les calculs eux-mêmes sont courts.

**Apprentissage**

1. Gain significatif malgré tout : Le facteur d'accélération de 2.86 est notable, montrant que le multithreading est efficace ici, mais qu'il n'atteint pas son plein potentiel.
2. Limites du multithreading : Le multithreading ne compense pas un code inefficace. Dans des cas comme le Minimax, une optimisation comme l'élagage alpha-bêta peut réduire exponentiellement le temps de calcul, bien plus qu'une simple parallélisation.
   Dans des algorithmes branchés (comme Minimax), le parallélisme est limité par la structure même de l'algorithme. Les optimisations algorithmiques sont souvent plus impactantes.

**Conclusion**

Le multithreading apporte ici une amélioration notable, mais elle est limitée par des facteurs structurels (temps de création des threads, charge non équilibrée). Optimiser le code séquentiel et choisir une stratégie adaptée pour tirer parti du parallélisme sont essentiels pour maximiser les performances.

## 1.3 Question 6 :

Principe de l'algorithme

**Recherche de la séquence cible :**
L'algorithme compare chaque séquence générée à la séquence cible. Une fois la séquence cible trouvée, il enregistre le nombre de tentatives nécessaires.

**Calcul des statistiques :**
Pour chaque version (séquentielle et parallèle), on calcule : Le temps total d'exécution.
Le nombre moyen d'essais pour trouver la séquence.
L'écart-type des essais.

**Exécution séquentielle :**
Toutes les réplications sont exécutées une par une. Un seul thread effectue tout le travail.

**Exécution parallèle :**
Les réplications sont divisées entre plusieurs threads (jusqu'à $NUM\_THREADS$).
Chaque thread utilise un état différent du générateur pseudo-aléatoire.
Les threads vérifient périodiquement (tous les $CHECK\_INTERVAL$) si une autre réplication a déjà trouvé la séquence, ce qui permet d'interrompre les threads inutiles pour réduire le temps de calcul.

Reproductibilité

**Réplicabilité dans l'exécution parallèle**
La réplicabilité est essentielle pour garantir que les résultats d'un programme sont reproductibles entre différentes exécutions. Si elle est facile à obtenir dans un code séquentiel en utilisant un état initial identique pour le générateur pseudo-aléatoire, elle devient plus complexe dans un contexte parallèle, notamment pour les raisons suivantes :

**Gestion des threads et synchronisation**
En multithreading, plusieurs threads s'exécutent simultanément. L'ordre exact dans lequel ils sont lancés et complétés peut varier d'une exécution à l'autre, car il dépend du système d'exploitation et des circonstances.
Lorsqu'un thread trouve la solution, il devient nécessaire de stopper les autres threads. Cela nécessite une variable partagée (comme *min_ tries*) pour indiquer le nombre minimal d'itérations nécessaires, permettant aux autres threads de vérifier s'ils doivent arrêter ou continuer.
Problème : L'écriture simultanée sur une variable partagée peut ralentir considérablement les threads, car l'accès concurrent doit être synchronisé pour éviter des conflits.
Solution :
Dans cet algorithme, seules les écritures nécessaires sont effectuées : le thread qui trouve une solution écrit son nombre d'itérations sur *min_ tries*. Les autres threads ne peuvent écrire que s'ils trouvent une meilleure solution (moins d'itérations), ce qui est rare.

**Ordre d'exécution des threads**
Le principal défi pour la réplicabilité est que l'ordre de lancement et de terminaison des threads peut changer entre différentes exécutions. Si deux threads terminent presque simultanément, l'ordre dans lequel ils écrivent sur min_tries pourrait varier, donnant des résultats différents. Problème spécifique :
Si le thread 4 et le thread 6 trouvent la solution presque au même moment, l'exécution qui laisse le thread 4 écrire en premier peut donner un résultat différent de celle où le thread 6 écrit en premier.
Solution :
L'algorithme retient le nombre minimal d'itérations nécessaires pour trouver la solution et stoppe uniquement les threads qui prennent plus d'itérations. À la fin, le résultat final est basé sur le thread ayant exécuté le moins d'itérations, ce qui garantit que, avec les mêmes états de départ, le thread ayant le moins d'itérations sera toujours le même. Cela assure une réplicabilité parfaite.

**Limitation du nombre de threads**
Attention pour préserver l'efficacité du code il est important de s'assurer que le nombre de threads ne dépasse pas le nombre de processeurs logiques disponibles (dans cet exemple, 8). Si on lance plus de threads que de processeurs logiques : Les premiers threads doivent attendre que des threads en cours se terminent, ce qui peut rallonger le temps d'exécution global. L'ordre de complétion devient encore moins prévisible, ce qui peut introduire de la variabilité dans les résultats.

**Réplicabilité garantie**
L'algorithme s'assure que le thread ayant effectué le moins d'itérations est toujours sélectionné, grâce à la gestion stricte de la variable *min_ tries*.
Avec les mêmes états initiaux pour les générateurs aléatoires, les résultats seront identiques, peu importe l'ordre de lancement ou d'exécution des threads.

simulate_sequential

```cpp
/** ------------------------------------------------------------------ *
 * simulate_sequential                                                *
 *                                                                    *
 * @brief  Simulation s quentielle de recherche de s quence cible.    *
 *                                                                    *
 * @param  tries_needed_seq : vecteur pour stocker les essais n cessaires *
 * @param  time_seq : r f rence pour stocker le temps d'ex cution     *
 * ------------------------------------------------------------------ **/
void simulate_sequential(std::vector<long long> &tries_needed_seq, double &time_seq)
{
    auto start_seq = std::chrono::high_resolution_clock::now();

    CLHEP::MTwistEngine generator;
    std::string status_file = "save/status_0.txt";
    generator.restoreStatus(status_file.c_str());

    for (int replication = 0; replication < NUM_REPLICATIONS; ++replication)
    {
        long long tries = 0;

        while (true)
        {
            std::string sequence;
            for (int i = 0; i < TARGET_LENGTH; ++i)
            {
                int index = generator.flat() * NUM_NUCLEOTIDES;
                char nucleotide = NUCLEOTIDES[index];
                sequence += nucleotide;
            }
            ++tries;

            if (sequence == TARGET_SEQUENCE)
            {
                tries_needed_seq[replication] = tries;
                std::cout << "S quentiel - R plication " << replication + 1
                          << ": S quence trouv e apr s " << tries << " essais.\n";
                break;
            }
        }
    }

    auto end_seq = std::chrono::high_resolution_clock::now();
    time_seq = std::chrono::duration<double>(end_seq - start_seq).count();
}
```

simulate_parallel

```
void simulate_parallel(std::vector<long long> &tries_needed_par,std::vector<double> &time_needed_par,double &time_par)
{
    auto start_par = std::chrono::high_resolution_clock::now();
    for (int replication = 0; replication < NUM_REPLICATIONS; ++replication)
    {
        std::vector<long long> tries_per_thread(NUM_THREADS, 0);
        long long min_tries = LLONG_MAX;
        auto start_thread = std::chrono::high_resolution_clock::now();

        #pragma omp parallel num_threads(NUM_THREADS) shared(min_tries)
        {
            int thread_id = omp_get_thread_num();
            CLHEP::MTwistEngine generator;
            int status_index = replication * NUM_THREADS + thread_id;
            status_index = status_index % NB_STATUT;
            std::string status_file = "save/status_" + std::to_string(status_index) + ".txt";
            generator.restoreStatus(status_file.c_str());

            long long local_tries = 0;
            while (true)
            {
                if (local_tries % CHECK_INTERVAL == 0)
                {
                    long long local_min_tries;
                    #pragma omp atomic read
                    local_min_tries = min_tries;
                    if (local_tries > local_min_tries)
                    {
                        break;
                    }
                }
                std::string sequence;
                for (int i = 0; i < TARGET_LENGTH; ++i)
                {
                    int index = generator.flat() * NUM_NUCLEOTIDES;
                    char nucleotide = NUCLEOTIDES[index];
                    sequence += nucleotide;
                }
                ++local_tries;

                if (sequence == TARGET_SEQUENCE)
                {
                    tries_per_thread[thread_id] = local_tries;
                    #pragma omp atomic write
                    min_tries = local_tries;
                    break;
                }
            }
        }
        auto end_thread = std::chrono::high_resolution_clock::now();
        min_tries = LLONG_MAX;
        double min_time = std::chrono::duration<double>(end_thread - start_thread).count();
        for (int i = 0; i < NUM_THREADS; ++i)
        {
            if (tries_per_thread[i] > 0 && tries_per_thread[i] < min_tries)
            {
                min_tries = tries_per_thread[i];
            }
        }
        tries_needed_par[replication] = min_tries;
        time_needed_par[replication] = min_time;
    }
    auto end_par = std::chrono::high_resolution_clock::now();
    time_par = std::chrono::duration<double>(end_par - start_par).count();
}
```

Sortie

Séquentiel - Réplication 1 : Séquence trouvée après 29961378 essais.
Séquentiel - Réplication 2 : Séquence trouvée après 305564475 essais.
Séquentiel - Réplication 3 : Séquence trouvée après 196146727 essais.
Séquentiel - Réplication 4 : Séquence trouvée après 196165112 essais.
Séquentiel - Réplication 5 : Séquence trouvée après 78502804 essais.
Séquentiel - Réplication 6 : Séquence trouvée après 89150323 essais.
Séquentiel - Réplication 7 : Séquence trouvée après 856342775 essais.
Séquentiel - Réplication 8 : Séquence trouvée après 5655219 essais.
Séquentiel - Réplication 9 : Séquence trouvée après 26183563 essais.
Séquentiel - Réplication 10 : Séquence trouvée après 707614173 essais.

Parallèle - Réplication 1, Thread 3 : Séquence trouvée après 22176224 essais.
Parallèle - Réplication 2, Thread 0 : Séquence trouvée après 3385740 essais.
Parallèle - Réplication 3, Thread 4 : Séquence trouvée après 47637942
Parallèle - Réplication 4, Thread 4 : Séquence trouvée après 26089 essais.
Parallèle - Réplication 5, Thread 7 : Séquence trouvée après 9413509 essais.
Parallèle - Réplication 6, Thread 7 : Séquence trouvée après 4667801 essais.
Parallèle - Réplication 7, Thread 0 : Séquence trouvée après 56829722 essais.
Parallèle - Réplication 8, Thread 7 : Séquence trouvée après 19026837 essais.
Parallèle - Réplication 9, Thread 0 : Séquence trouvée après 37895864 essais.
Parallèle - Réplication 10, Thread 3 : Séquence trouvée après 35421944 essais.

— Résultats Séquentiels —
Nombre moyen d'essais : 2.49129e+08
Écart-type : 2.98038e+08
Temps d'execution : 207.993 secondes

— Résultats Parallèles —
Nombre moyen d'essais : 2.36482e+07
Écart-type : 1.99252e+07
Temps d'exécution : 44.026 secondes

— Comparaison des Temps —
Temps séquentiel total : 207.993 secondes
Temps parallèle total : 44.026 secondes

Analyse des Résultats

**Résultats séquentiels :**

1. Nombre moyen d'essais : $2.49 * 10^8$
2. Temps d'exécution total : $\tilde{2}08$ secondes.

**Résultats parallèles :**

1. Nombre moyen d'essais : $2.36 * 10^7$
2. Temps d'exécution total : $\tilde{4}4$ secondes.
3. Facteur d'accélération : $207.993/44.026 \approx 4.72$

**Accélération théorique attendue :**
Avec 8 threads, on pourrait s'attendre à un facteur proche de 8 dans des conditions idéales.
Raisons pour lesquelles l'accélération est inférieure à 8 Coût de création et synchronisation des threads :

1. Le lancement et la gestion des threads ajoutent une surcharge fixe. Dans ce code, chaque thread doit fréquemment consulter une variable partagée (*min_tries*), ce qui nécessite une synchronisation (atomic), ralentissant les threads.
2. Facteur chance : La nature aléatoire de l'algorithme peut jouer un rôle important :
   Scénario favorable : Dans le mode parallèle, un thread peut trouver la solution très rapidement, ce qui entraîne un gain important. À l'inverse, dans le mode séquentiel, le même scénario chanceux sur une réplication réduirait le temps global.
   Scénario défavorable : Si aucun thread ne trouve rapidement la solution en mode parallèle ou si le séquentiel bénéficie de plusieurs répétitions "chanceuses" avec peu d'essais, cela déséquilibre la comparaison.
   Bien que le nombre de réplications tende à compenser ce facteur, il reste une source de variabilité entre les modes séquentiel et parallèle.

**Pourquoi une accélération de 4.72x est tout de même satisfaisante ?**

1. Complexité de la tâche : Le temps nécessaire pour qu'un thread vérifie s'il doit s'arrêter est proportionnel au nombre d'essais effectués. Si les threads travaillent sur des durées très différentes, le multithreading est moins efficace.
2. Réplicabilité assurée : L'algorithme garantit que le même thread termine avec le nombre minimal d'essais pour chaque réplication, même si cela ajoute une petite surcharge. Cette réplicabilité est prioritaire par rapport à une accélération maximale.
3. Proximité d'un facteur idéal avec optimisation : Avec un nombre de réplications plus important ou des calculs plus longs, la surcharge liée aux threads deviendrait négligeable, permettant d'approcher un facteur d'accélération plus proche de 8.

**Optimisations possibles**

1. Réduire la fréquence des accès à *min_tries* : Augmenter la valeur de *CHECK_INTERVAL* pour que les threads consultent moins fréquemment la variable partagée. Cela peut réduire la surcharge de synchronisation.
2. Augmenter la charge de travail : Si chaque réplication nécessite plus d'essais (par exemple, en augmentant la taille de la séquence cible), le coût fixe de gestion des threads deviendrait négligeable par rapport au temps total.

# 2 Conclusion

Ce travail a permis d'explorer les performances des approches séquentielles et parallèles dans des contextes de simulation stochastique, tout en mettant en avant les défis liés à la reproductibilité et à l'optimisation. Les résultats obtenus montrent que :

1. Le multithreading améliore significativement les performances : Dans le cas de la recherche de séquence cible et des simulations Monte Carlo, l'accélération obtenue, bien qu'inférieure au facteur théorique, reste substantielle.
2. La reproductibilité est un enjeu clé : L'utilisation d'états prégénérés pour le générateur pseudo-aléatoire Mersenne Twister garantit une indépendance et une reproductibilité des résultats, essentiels pour valider les simulations. Les solutions implémentées, comme la gestion stricte de la variable partagée *min_ tries*, assurent une reproductibilité parfaite même en environnement parallèle.
3. Le multithreading a des limitations : L'efficacité est réduite par le coût de création et de synchronisation des threads, les déséquilibres de charge entre threads, et la variabilité inhérente aux processus aléatoires. Ces facteurs limitent l'accélération théorique, particulièrement pour des tâches courtes ou faiblement parallélisables.
4. L'optimisation séquentielle reste essentielle : Avant de recourir au multithreading, un code séquentiel optimisé offre souvent des gains de performance supérieurs. Par exemple, dans des algorithmes comme Minimax, des optimisations algorithmiques (comme l'élagage alpha-bêta) peuvent réduire exponentiellement le temps de calcul.

**Perspectives et recommandations**

1. Optimiser la gestion parallèle : Réduire les accès aux variables partagées et mieux équilibrer la charge entre threads pourraient encore améliorer les performances.
2. Augmenter la charge de travail : Étendre la taille des simulations permettrait de minimiser l'impact des coûts fixes associés au multithreading.
3. Favoriser l'adaptation aux ressources matérielles : Limiter le nombre de threads au nombre de processeurs logiques garantit une exploitation optimale des capacités matérielles.

En conclusion, ce TP a démontré que le multithreading, bien que complexe à implémenter efficacement, peut considérablement améliorer les performances tout en assurant une reproductibilité fiable, un atout majeur pour des applications scientifiques et techniques.