Rapport TP5 IDM

Génération de flux parallèles de nombres pseudo-aléatoires
Bibliothèque CLHEP

Année : 2024-2025

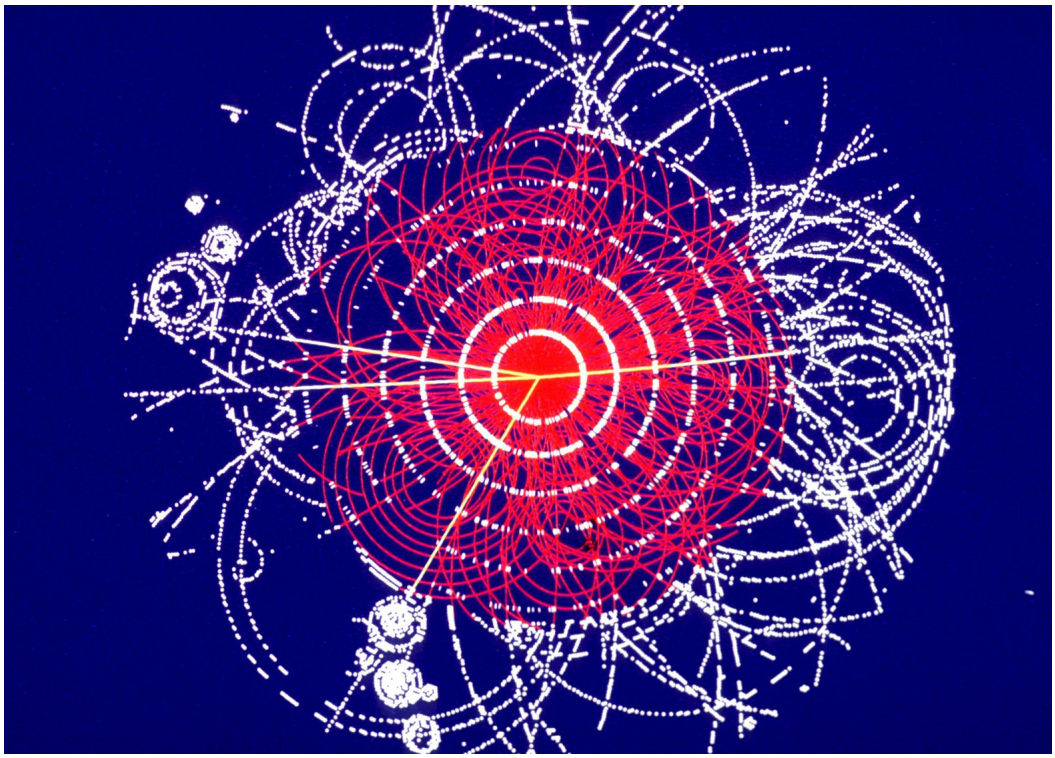

FIGURE 1 – Simulation de la désintégration d'un boson de Higgs

Table des matières

Table des figures

1 Introduction

Le TP est organisé de la manière suivante. Les résultats sont disponibles dans les fichiers intitulés "`results`" (`TP5-A` et `TP5-B`). La compilation s'effectue à l'aide du Makefile : pour la première partie du TP, utilisez la commande `make`, et pour la deuxième partie, utilisez la commande : `make tp5b`.

Pour exécuter la partie B du tp5, exécuter le script `runParallelSim.sh`.

```
TP5
├── Random
│   ├── CLHEP
│   └── Random
│       ├── autotools
│       ├── bin
│       ├── doc
│       ├── include
│       ├── lib
│       └── src
├── bonus
│   ├── bio-sequence
│   └── open-mp
├── results
├── scripts
│   └── results
├── status
└── utils
```

- **`Random`** : bibliothèque CLHEP
- **`bonus`** : exercices supplémentaires du TP
- **`results`** : résultats des simulations (TP5a et TP5b)
- **`scripts`** : scripts pour l'exécution des simulations
- **`utils`** : uilitaires pour les simulations et le traitement des données

Bibliothèque CLHEP (Class Library for High Energy Physics)

La bibliothèque CLHEP (Class Library for High Energy Physics) est une collection de bibliothèques écrites en C++ principalement utilisées dans le domaine de la physique des hautes énergies.

CLHEP fournit des outils et des classes utilitaires largement utilisés pour la simulation et l'analyse des données. Elle est intégrée dans de nombreux projets scientifiques, y compris des frameworks comme Geant4.

2 Installation de la bibliothèque

Q1.

2.1 Compilation séquentielle

Nous testons dans un premier temps la compilation séquentielle (sans parallélisation avec les jobs).

```
time make
```

Voici les résultats que nous obtenons respectivement pour les temps réels, utilisateurs et systèmes :

```
real    1m19,202s
user    1m11,357s
sys     0m7,600s
```

2.2 Compilation en parallèle

La compilation est réalisée sur ma machine personnelle, qui possèdent 12 coeurs (commande `nproc`) sous Linux (Ubuntu).

```
    nproc
    >> 12
```

Nous spécifions donc une compilation parallèle avec 12 cœurs logiques.

```
    time make -j12
```

Voici les résultats que nous obtenons respectivement pour les temps réels, utilisateurs et systèmes :

```
real    0m29,562s
user    4m31,260s
sys     0m22,291s
```

2.3 Observations

Les résultats en mode séquentiel et en parallèle, montrent clairement l'avantage de la compilation parallélisée. Nous constatons que la parallélisation optimise le temps réel de compilation, mais maximise le temps utilisateur (`user`).
En passant de **1m19s (séquentiel)** à **29s (parallèle)**, la compilation parallèle divise presque le temps réel par un facteur de **2.68**.
Le **temps utilisateur** passe de **1m11s (séquentiel)** à **4m31s (parallèle)**. Cela reflète une augmentation de la charge de travail répartie sur plusieurs cœurs.

Bien que le temps total `utilisateur + système` augmente (liée au traitement simultané sur plusieurs cœurs), le gain en temps réel justifie l'approche.

En calculant le ratio du temps réel en séquentiel sur le temps réel en parallèle, nous pouvons en déduire l'efficacité de la simulation lancée en parallèle :

$$\frac{time\_real\_seq}{time\_real\_parallel} = \frac{71,4}{29} = 2,46 > 1$$

L'exécution de la simulation en parallèle est donc 2 fois plus rapide que le lancement de la simulation

3 Tests des générateurs de la librairie

Une fois la librarie installée, nous procédons à l'exécution des tests (`testRandom.cc` et `testrand.cc`).

Le fichier `testRandom.cc` initialise plusieurs moteurs aléatoires, comme le générateur `HepJamesRandom`, `RandEngine` et nous pouvons choisir l'un de ces moteurs pour effectuer des tirages aléatoires sur diverses distributions (uniforme, exponentielle, gaussienne, etc.). Les résultats des tirages sont affichés à chaque étape.

```
g++ testRandom.cc -I../include ../lib/libCLHEP-Random-2.1.0.0.a -static
```

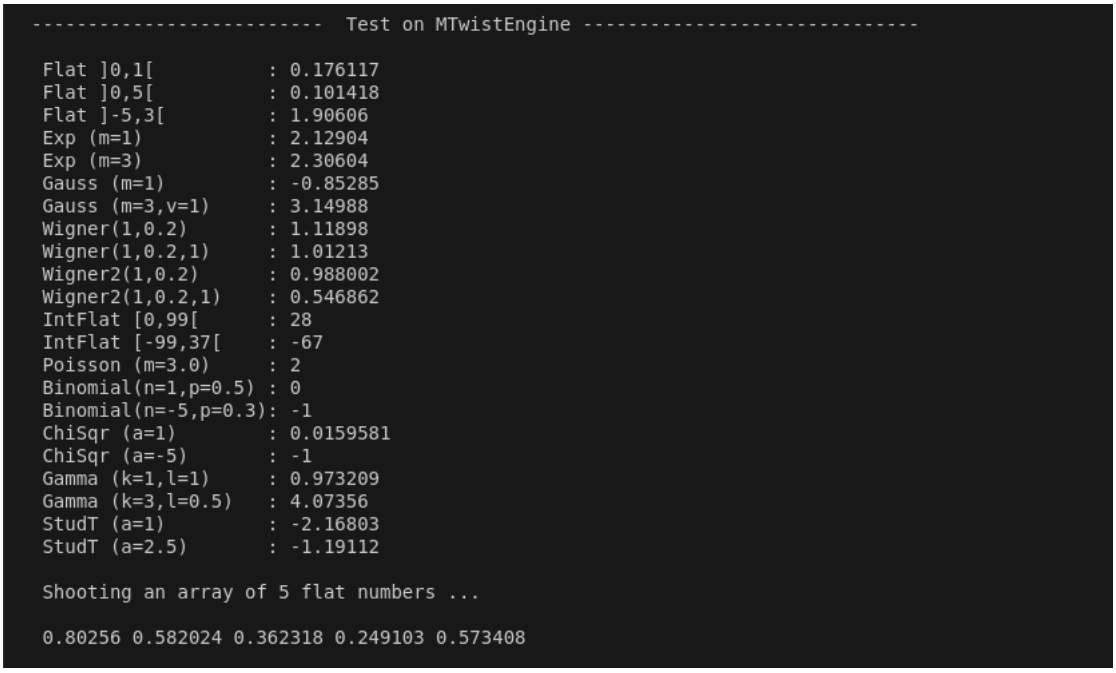

FIGURE 2 – Exemple de test (`testRandom`) sur le générateur Mersenne Twister

La majorité des valeurs sont dans les plages attendues pour chaque distribution statistique.

Nous testons également les générateurs avec le fichier donné en sujet :

```cpp
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <unistd.h>

#include "CLHEP/Random/MTwistEngine.h"

int main ()
{
   CLHEP::MTwistEngine * s = new CLHEP::MTwistEngine();

   int fs;
   double f;
   unsigned int nbr;

   fs = open("./rngbForMarsagliaTests",O_CREAT|O_TRUNC|O_WRONLY,S_IRUSR|S_IWUSR);

   for(int i = 1; i < 3000000; i++)
   {
     f = s->flat();
     nbr = (unsigned int) (f * UINT_MAX);

     std::cout << f << std::endl;

     write(fs,&nbr,sizeof(unsigned int));
   }

   close(fs);

   delete s;

   return 0;
}
```

Le programme de test génère 3 000 000 nombres pseudo-aléatoires flottants (en utilisant le moteur Mersenne Twister), les convertit en entiers non signés, les écrit dans un fichier binaire, et affiche les valeurs flottantes à la console.

Le fichier généré peut être utilisé pour des tests (par exemple, des tests de Marsaglia pour évaluer la qualité des générateurs aléatoires).

4 Utilisation de la librairie

Q2.

Le but ici est de créer une instance du générateur de nombres aléatoires **MTwistEngine (CLHEP)**.

```cpp
#include "Random/Random/MTwistEngine.h"
#include <iostream>

int statusMT() {
    CLHEP::MTwistEngine mtEngine;

    mtEngine.saveStatus();

    std::cout << "First 10 generated numbers:\n";
    for (int i = 0; i < 10; ++i) {
        std::cout << mtEngine.flat() << std::endl;
    }

    std::cout << "Generator state saved to 'MTwist.conf'.\n";
    mtEngine.restoreStatus();

    std::cout << "Generator restored.\n";
    std::cout << "\nRestored generator, first 10 generated numbers:\n";

    for (int i = 0; i < 10; ++i) {
        std::cout << mtEngine.flat() << std::endl;
    }

    return 0;
}

int main()
{
    statusMT();
}
```

Nous sauvegardons d'abord l'état du générateur avec `saveStatus()`. Ensuite, il génère et affiche les 10 premiers nombres aléatoires.

L'état du générateur est restauré avec `restoreStatus()`, et les 10 premiers nombres aléatoires sont de nouveau générés et affichés, reproduisant ainsi exactement la même séquence qu'au début.

4.1 Analyse de `saveStatus()` et `restoreStatus()`

La méthode `saveStatus()` enregistre l'état interne du générateur dans un fichier texte, afin de pouvoir le recharger plus tard avec `restoreStatus()`.

La méthode `restoreStatus()` recharge l'état du générateur depuis un fichier. Cela permet de reprendre la génération de nombres aléatoires à partir du même point que celui où elle avait été interrompue.

```
First 10 generated numbers:
0.176117
0.0202835
0.863258
0.118952
0.463625
0.0618998
0.823987
0.571751
0.0917234
0.777519
Generator state saved to 'MTwist.conf'.
Generator restored.

Restored generator, first 10 generated numbers:
0.176117
0.0202835
0.863258
0.118952
0.463625
0.0618998
0.823987
0.571751
0.0917234
0.777519
```

FIGURE 3 – Restauration de l'état du générateur Mersenne Twister entre deux générations successives d'une séquence de 10 nombres aléatoires

Q3.

```
/**
 * @brief generates and saves multiple random generator statuses.
 *
 * generates multiple random generator states, saving each state as a separate file
 * in the "status" directory. Each draw is tracked using a progress bar to visualize
    progress.
 *
 * @param numStatus number of generator states to create.
 * @param numDraws number of random draws to generate for each status.
 */
void generateAndSaveStatus(int numStatus, long numDraws) {

    printf("\n ==== Saving generator status ==== \n");
    printf("Num. status : %d, Num. draws : %ld \n", numStatus, numDraws);

    std::string directory = "status";

    std::string name;
    std::string filename;

    if (!std::filesystem::exists(directory)) {
        std::filesystem::create_directory(directory);
    }

    CLHEP::MTwistEngine mtEngine;

    for (int statusIndex = 0; statusIndex < numStatus; ++statusIndex) {
        for (long i = 0; i < numDraws; ++i) {
            mtEngine.flat();
            if ((i + 1) % (numDraws / 100) == 0) progress(i + 1, numDraws);
        }

        name = "status-" + std::to_string(statusIndex + 1) + ".conf";
        filename = directory + "/" + name;

```

```
35          mtEngine.saveStatus(filename.c_str());
36
37          std::cout << "\n" << name << " saved." << std::endl;
38      }
39  }
```

```
 ==== Saving generator status ====
Num. status : 10, Num. draws : 3000000000
[==================================================================] 100 %
status-1.conf saved.
[==================================================================] 100 %
status-2.conf saved.
[==================================================================] 100 %
status-3.conf saved.
[==================================================================] 100 %
status-4.conf saved.
[==================================================================] 100 %
status-5.conf saved.
[==================================================================] 100 %
status-6.conf saved.
[==================================================================] 100 %
status-7.conf saved.
[==================================================================] 100 %
status-8.conf saved.
[==================================================================] 100 %
status-9.conf saved.
[==================================================================] 100 %
status-10.conf saved.
```

FIGURE 4 – Visualisation de la progression et sauvegarde des états du générateur Mersenne Twister

5 Simulation par Monte Carlo

Q4. & Q5.

5.1 Approche séquentielle

Nous allons dans un premier temps lancer la simulation en séquentielle puis comparer le temps d'exécution et les résultats obtenus par `sequence splitting` (approche parallèle).

Voici les résultats que nous obtenons pour la simulation d'un calcul de volume d'une sphère par Monte Carlo pour 10×3 milliards de tirages.

```
 === Performing Monte Carlo Simulation ===

** Start task at: 2024-12-19 21:30:22

 Num. replications : 10 / Num. draws per replication : 3000000000

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 1: Estimated volume = 4.18873

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 2: Estimated volume = 4.18878

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 3: Estimated volume = 4.18882

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 4: Estimated volume = 4.18884

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 5: Estimated volume = 4.1889

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 6: Estimated volume = 4.18885

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 7: Estimated volume = 4.18882

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 8: Estimated volume = 4.18874

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 9: Estimated volume = 4.18879

 Estimating volume by Monte Carlo method...
[====================================================================] 100 %
 Replication 10: Estimated volume = 4.18878

 >> Average estimated volume after 10 replications: 4.1888
Time used : 534.667 seconds
```

FIGURE 5 – Visualisation de la progression et résultats de l'estimation du volume d'une sphère par la méthode de Monte Carlo

Le temps de la simulation par Monte Carlo en **séquentiel** est donc de **534.667 secondes** soit d'environ **8 minutes**.

5.2 Approche en parallélisation

5.2.1 Sequence splitting

L'approche par "sequence splitting" consiste à diviser la séquence aléatoire en plusieurs sous-séquences indépendantes. Pour cela, on utilise des états différents du générateur aléatoire pour chaque réplication.
Chaque tâche exécute le même programme, mais utilise un flux aléatoire différent initialisé par un fichier de statut unique (approche SPMD).

Voici la démarche que nous allons adopter. Tout d'abord nous allons générer les fichiers (status) d'état aléatoires (soit les fichiers `status-x.conf`).

Notre programme doit accepter le **fichier de statut** et le **nombre de points comme arguments**. Chaque réplication (chaque exécution) doit restaurer son propre état et effectuer le calcul de manière indépendante.

Le script suivant va nous permettra de lancer **10 réplications en parallèle selon un status pour le MT choisi** et passé en paramètre :

```
#!/bin/bash

num_points=3000000000
status_file_prefix="status-"

mkdir -p results

overall_start_time=$(date +%s%3N)

# Run the 10 replications in parallel
for i in {1..10}
do
    start_time=$(date "+%Y-%m-%d %H:%M:%S.%3N")

    echo "[Script PID: $$] [Replication $i] Starting replication at $start_time" >>
        replication_log.txt

    status_file="../status/${status_file_prefix}${i}.conf"

    if [ ! -f "$status_file" ]; then
        echo "[Script PID: $$] [Replication $i] Error: Status file $status_file does
            not exist!" >> replication_log.txt
        continue
    fi

    ./../tp5b $status_file $num_points > "results/output_replication_$i.txt" &

    background_pid=$!

    echo "[Script PID: $$] [Replication $i] Background job started with PID:
        $background_pid at $start_time" >> replication_log.txt
done
```

Nous pouvons améliorer le script pour obtenir plus d'informations sur l'exécution de la simulation (temps total mis, volume moyen estimé) :

```
wait

completion_time=$(date "+%Y-%m-%d %H:%M:%S.%3N")
overall_end_time=$(date +%s%3N)

total_time_ms=$((overall_end_time - overall_start_time))
total_time_seconds=$(echo "scale=3; $total_time_ms / 1000" | bc)

echo "[Script PID: $$]  *** All simulations completed at $completion_time ***" >> replication_log.txt
echo "[Script PID: $$]  *** Total time taken: $total_time_seconds seconds ***" >> replication_log.txt

# Average estimated volume
total_volume=0
count=0

for i in {1..10}
do
    output_file="results/output_replication_$i.txt"

    if [ -f "$output_file" ]; then
        estimated_volume=$(grep "Volume:" "$output_file" | awk -F ':' '{print $2}' | xargs)
        if [ -n "$estimated_volume" ]; then
            total_volume=$(echo "$total_volume + $estimated_volume" | bc)
            count=$((count + 1))
        fi
    fi
done

if [ $count -gt 0 ]; then
    average_volume=$(echo "scale=5; $total_volume / $count" | bc)
    echo "Average estimated volume after $count replications: $average_volume" >> replication_log.txt
    echo "Average estimated volume after $count replications: $average_volume"
else
    echo "No valid estimated volumes found in the results."
    echo "No valid estimated volumes found in the results." >> replication_log.txt
fi

echo "All simulations completed. Total time taken: $total_time_seconds seconds."
```

La progression des différentes simulations peut-être consultée dans le dossier `output-results-x.txt`. Les logs sont disponsibles dans le fichier `result`.

Voici les résultats que nous obtenons (logs) :

```
[Script PID: 66620] [Replication 1] Starting replication at 2024-12-19 21:56:46.325
[Script PID: 66620] [Replication 1] Background job started with PID: 66624 at
    2024-12-19 21:56:46.325

[Script PID: 66620] [Replication 2] Starting replication at 2024-12-19 21:56:46.330
[Script PID: 66620] [Replication 2] Background job started with PID: 66626 at
    2024-12-19 21:56:46.330

[Script PID: 66620] [Replication 3] Starting replication at 2024-12-19 21:56:46.335
[Script PID: 66620] [Replication 3] Background job started with PID: 66628 at
    2024-12-19 21:56:46.335

[Script PID: 66620] [Replication 4] Starting replication at 2024-12-19 21:56:46.341
[Script PID: 66620] [Replication 4] Background job started with PID: 66630 at
    2024-12-19 21:56:46.341

[Script PID: 66620] [Replication 5] Starting replication at 2024-12-19 21:56:46.346
[Script PID: 66620] [Replication 5] Background job started with PID: 66632 at
    2024-12-19 21:56:46.346

[Script PID: 66620] [Replication 6] Starting replication at 2024-12-19 21:56:46.352
[Script PID: 66620] [Replication 6] Background job started with PID: 66634 at
    2024-12-19 21:56:46.352

[Script PID: 66620] [Replication 7] Starting replication at 2024-12-19 21:56:46.358
[Script PID: 66620] [Replication 7] Background job started with PID: 66636 at
    2024-12-19 21:56:46.358

[Script PID: 66620] [Replication 8] Starting replication at 2024-12-19 21:56:46.363
[Script PID: 66620] [Replication 8] Background job started with PID: 66638 at
    2024-12-19 21:56:46.363

[Script PID: 66620] [Replication 9] Starting replication at 2024-12-19 21:56:46.368
[Script PID: 66620] [Replication 9] Background job started with PID: 66640 at
    2024-12-19 21:56:46.368

[Script PID: 66620] [Replication 10] Starting replication at 2024-12-19 21:56:46.374
[Script PID: 66620] [Replication 10] Background job started with PID: 66642 at
    2024-12-19 21:56:46.374


[Script PID: 66620]  *** All simulations completed at 2024-12-19 21:58:30.013 ***

[Script PID: 66620]  *** Total time taken: 103.694 seconds ***
Average estimated volume after 10 replications: 4.18880
```

Le temps total de la simulation de Monte Carlo en **parallèle** (après terminaison des 10 threads) est de **103.694 secondes** soit environ **1 minutes et 43 secondes**.

	Nombre de points	**Temps total (s)**	**Volume estimé (moyenne)**
Séquentielle	10×3 milliards	534.667 s (8 min 55 s)	4.18888
Parallèle (sequence splitting)	10×3 milliards	103.694 s (1 min 43 s)	4.18880

TABLE 1 – Comparaison des temps d'exécution et des volumes estimés par Monte Carlo

Nous pouvons calculer l'efficacité de l'approche parallèle en calculant le ratio suivant :

$$\frac{time\_seq}{time\_parallel} = \frac{534.667}{103.694} = 5 > 1$$

L'exécution de la simulation en parallèle est donc 5 fois plus rapide que la simulation séquentielle de l'estimation du volume par Monte Carlo et cela pour un résultat similaire (volume moyen d'environ $\frac{4}{3} \times \pi$).

6 Indicateurs statistiques

Nous pouvons également évaluer les résultats obtenus par les approches séquentielle et parallèle à l'aide d'indicateurs statistiques tels que la moyenne, la variance, l'écart-type et l'intervalle de confiance.

— **Variance (σ^2)** :

$$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)^2$$

où μ est la moyenne et x_i est le volume estimé de la i-ième réplication.

Une variance faible indiquerait que les différentes réplications donnent des résultats proches les uns des autres, ce qui est un signe de stabilité et de précision des estimations. Au contraire, une variance élevée pourrait indiquer la nécessité d'augmenter le nombre de points tirés ou le nombre de réplications.

— **Écart-type (σ)** :

$$\sigma = \sqrt{\sigma^2}$$

Comme l'écart-type a la même unité que la grandeur mesurée (le volume), il est utilisé pour interpréter la précision des résultats.

— **Intervalle de confiance à 95%** :

$$IC = \mu \pm t_{\alpha/2} \cdot \frac{\sigma}{\sqrt{n}}$$

où $t_{\alpha/2}$ est la valeur critique de la distribution de Student pour $n-1$ degrés de liberté, μ est la moyenne, σ est l'écart-type et n est le nombre de réplications.

L'intervalle de confiance à 95% permet de quantifier l'incertitude sur l'estimation du volume. Il garantit, avec une probabilité de 95%, que la "vraie" valeur du volume se trouve à l'intérieur de cet intervalle. Plus l'intervalle est étroit, plus la précision de l'estimation est élevée. La largeur de l'intervalle dépend de la variance, du nombre de réplications n et de la valeur critique $t_{\alpha/2}$ de la loi de Student.

```
    double averageVolume = totalVolume / numReplications;

    double sumSquaredDiffs = 0.0;
    for (double volume : volumes) {
        sumSquaredDiffs += std::pow(volume - averageVolume, 2);
    }
    double variance = sumSquaredDiffs / (numReplications - 1);
    double stdDeviation = std::sqrt(variance);

    double t_value = 2.262; // Student's t-value for a 95% confidence interval (n = 10 - 1 = 9 degrees of freedom)
    double marginOfError = t_value * (stdDeviation / std::sqrt(numReplications));
    double lowerBound = averageVolume - marginOfError;
    double upperBound = averageVolume + marginOfError;
```

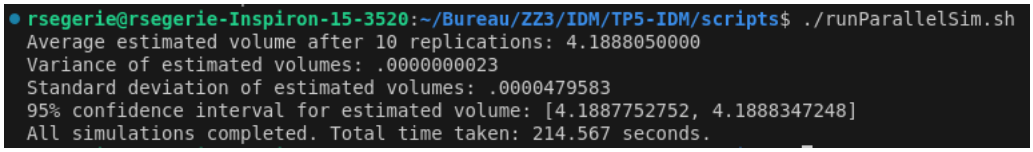

FIGURE 7 – Indicateurs statistiques pour la simulation en parallèle

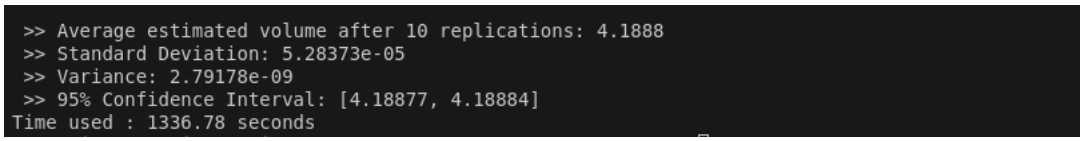

FIGURE 6 – Indicateurs statistiques pour la simulation en séquentielle

7 Différentes techniques et bibliothèques de parallélisation

7.1 OpenMP (Open Multi-Processing)

OpenMP est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. L'API prends en charge de nombreuses plateformes (incluant GNU/Linux, OS X et Windows) pour les langages de programmation tels que le **C**, **C++** et **Fortran**.

7.1.1 Simulation d'un neutron

Nous allons implémenter la simulation d'un neutron afin de manipuler la bibliothèque `OpenMP`. Pour modéliser le comportement d'un neutron dans un milieu d'eau, plusieurs paramètres clés sont définis :

1. le **libre parcours moyen** : déterminant la distance moyenne parcourue par un neutron avant qu'un événement (collision, absorption) ne se produise.

2. le **rayon de la goutte d'eau** : définissantla frontière au-delà de laquelle le neutron est considéré comme échappé.

3. le **nombre maximum de pas** : représentant la limite du nombre de déplacements d'un neutron avant d'arrêter la simulation.

```c
#define LAMBDA 1.0
#define RADIUS 10.0
#define MAX_STEPS 10000

double random_step() {
    return -LAMBDA * log((double)rand() / RAND_MAX);
}

void simulate_neutron(int dimension, int *escaped, int *absorbed) {
    double x = 0.0, y = 0.0, z = 0.0;
    int steps = 0;

    while (steps < MAX_STEPS) {
        steps++;

        double step_length = random_step();

        if (dimension == 1) {
            x += (rand() % 2 ? 1 : -1) * step_length;
        } else if (dimension == 2) {
            double angle = ((double)rand() / RAND_MAX) * 2 * M_PI;
            x += step_length * cos(angle);
            y += step_length * sin(angle);
        } else if (dimension == 3) {
            double theta = ((double)rand() / RAND_MAX) * 2 * M_PI;
            double phi = acos(1 - 2 * ((double)rand() / RAND_MAX));
            x += step_length * sin(phi) * cos(theta);
            y += step_length * sin(phi) * sin(theta);
            z += step_length * cos(phi);
        }

        double distance = sqrt(x * x + y * y + z * z);
        if (distance > RADIUS) {
            #pragma omp atomic
            (*escaped)++;
            return;
        }

        if (((double)rand() / RAND_MAX) < 0.1) {
            #pragma omp atomic
            (*absorbed)++;
            return;
        }
    }
}

int main(int argc, char *argv[]) {
    if ( argc != 3) {
        printf( "Usage: ./{executable_file} {total_neutrons} {dimension}...\n" );
        exit( 0 );
    }

    int total_neutrons = atoi(argv[1]);
    int dimension      = atoi(argv[2]);

    int escaped  = 0;
    int absorbed = 0;

    srand(time(NULL));

    #pragma omp parallel for
```

```
62      for (int i = 0; i < total_neutrons; i++) {
63          int local_escaped = 0;
64          int local_absorbed = 0;
65          simulate_neutron(dimension, &local_escaped, &local_absorbed);
66
67          #pragma omp atomic
68          escaped += local_escaped;
69
70          #pragma omp atomic
71          absorbed += local_absorbed;
72      }
73
74      printf("Total neutrons: \t%d\n", total_neutrons);
75      printf("* escaped : \t%d\n", escaped);
76      printf("* absorbed: \t%d\n", absorbed);
77
78      return 0;
79  }
```

7.2 pthread

La bibliothèque `pthread.h` fait partie de la norme POSIX et **permet de créer, gérer et synchroniser des threads dans des programmes C/C++.**

Cette bibliothèque est largement utilisée pour la programmation multithread sur les systèmes Unix/Linux et macOS.

8 Application de la génération aléatoire de séquences en bioinformatique avec Mersenne Twister (MT)

L'objectif ici est de **simuler des séquences de nucléotides (A, C, G, T)** et de **rechercher une séquence cible prédéfinie**.

Nous utiliserons le générateur Mersenne Twister pour réaliser cette simulation.

Dans un premier temps, nous exécuterons plusieurs simulations séquentielles afin de déterminer le nombre de tentatives nécessaires pour obtenir la séquence cible, représentée ici par l'oligopeptide `"gattaca"`. Par la suite, nous paralléliserons ces essais à l'aide de la technique de `sequence splitting` abordée précédemment.

Enfin, nous effectuerons une analyse statistique comprenant le calcul de l'**écart-type** et de l'**intervalle de confiance**, afin d'estimer la probabilité d'obtenir, par hasard, des séquences plus longues, comme un génome humain).

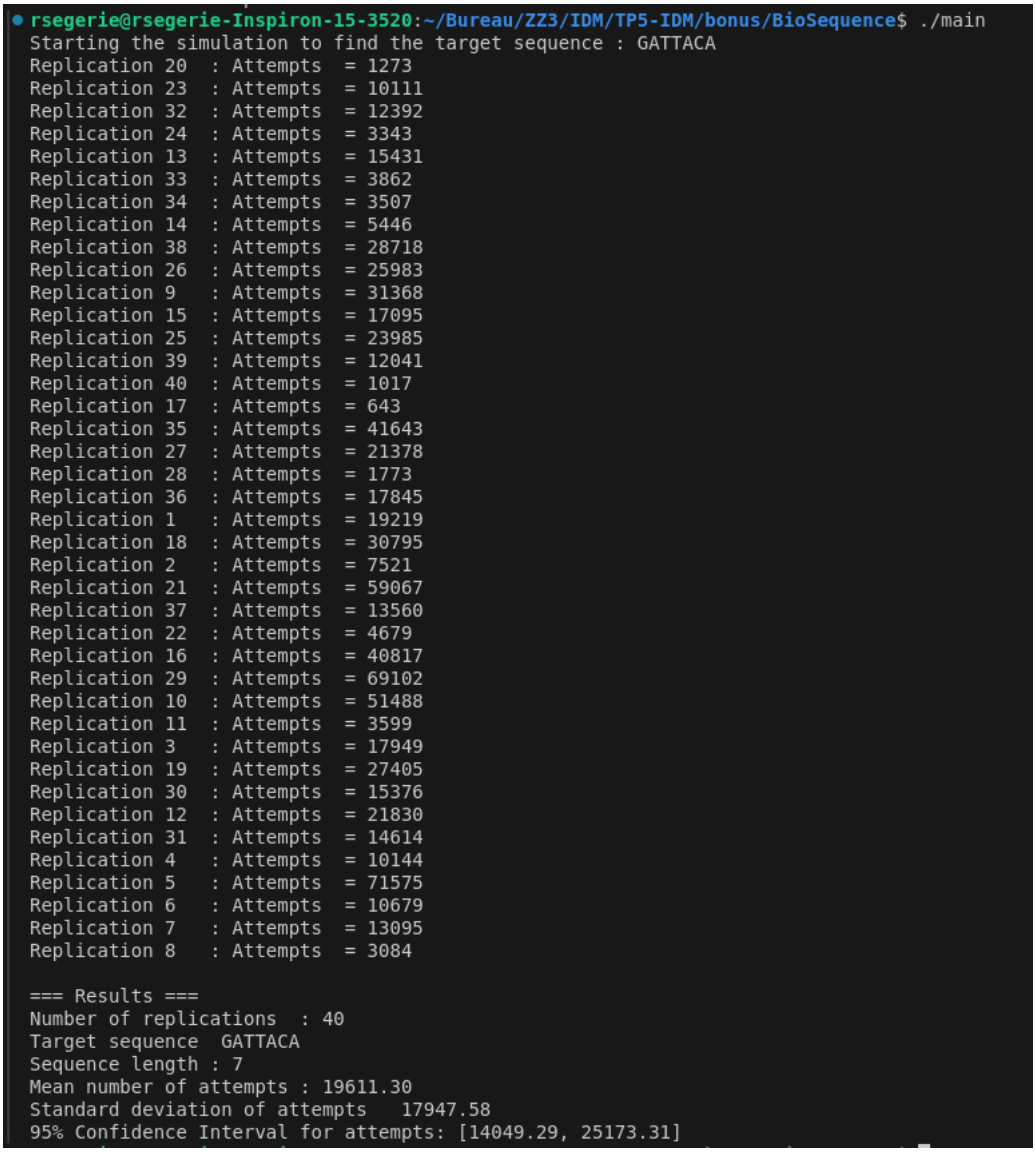

FIGURE 8 – Recherche d'une séquence cible (**"GATTACA"**) avec Mersenne Twister (approche parallèle)

La probabilité de tirer **au hasard** la bonne lettre à une position donnée est :

$$p_1 = \frac{1}{4}$$

Pour générer la séquence correcte de **3 milliards de bases**, chaque position doit être correcte et indépendante des autres. La probabilité totale P de générer **par hasard** cette chaîne exacte est donc le produit des probabilités de chaque position :

$$P = \left(\frac{1}{4}\right)^N$$

où $N = 3 \times 10^9$, correspondant à 3 milliards de nucléotides.

En réécrivant, nous obtenons :

$$P = 4^{-N} = 4^{-3\times10^9}$$

On peut également écrire 4 sous la forme 2^2, ce qui donne :

$$P = \left(2^2\right)^{-3\times10^9} = 2^{-2\cdot3\times10^9} = 2^{-6\times10^9}$$

```
** Target sequence: AAATTTGCGTTCGATTAG
sub-sequence: A
* attempts: 1
* probability (theorical): 0.2500000000
================================
sub-sequence: AA
* attempts: 21
* probability (theorical): 0.0625000000
================================
sub-sequence: AAA
* attempts: 11
* probability (theorical): 0.0156250000
================================
sub-sequence: AAAT
* attempts: 971
* probability (theorical): 0.0039062500
================================
sub-sequence: AAATT
* attempts: 2507
* probability (theorical): 0.0009765625
================================
sub-sequence: AAATTT
* attempts: 2525
* probability (theorical): 0.0002441406
================================
sub-sequence: AAATTTG
* attempts: 1451
* probability (theorical): 0.0000610352
================================
sub-sequence: AAATTTGC
* attempts: 28257
* probability (theorical): 0.0000152588
================================
sub-sequence: AAATTTGCG
* attempts: 797676
* probability (theorical): 0.0000038147
================================
sub-sequence: AAATTTGCGT
* attempts: 361018
* probability (theorical): 0.0000009537
================================
sub-sequence: AAATTTGCGTT
* attempts: 12687274
* probability (theorical): 0.0000002384
================================
sub-sequence: AAATTTGCGTTC
* attempts: 10513771
* probability (theorical): 0.0000000596
================================
sub-sequence: AAATTTGCGTTCG
* attempts: 95543582
* probability (theorical): 0.0000000149
================================
sub-sequence: AAATTTGCGTTCGA
* attempts: 55044844
* probability (theorical): 0.0000000037
================================
sub-sequence: AAATTTGCGTTCGAT
* attempts: 423629810
* probability (theorical): 0.0000000009
================================
```

Nous avons généré successivement des sous-séquences de la chaîne cible, et avons mesuré le nombre d'essais nécessaires pour obtenir chaque sous-séquence.

Les résultats montrent que, à mesure que la sous-séquence devient plus longue, le nombre d'essais nécessaires pour la générer augmente de manière exponentielle, ce qui est en accord avec la probabilité théorique. Par exemple, pour la sous-séquence `"A"`, qui représente une seule position, il suffit d'un seul essai pour l'obtenir. Cependant, pour des sous-séquences plus longues, comme `"AAATTTGCGTTCGAT"`, le nombre d'essais nécessaires est extrêmement élevé (423,629,810 essais).