



UNIVERSITE RENNES 1

---

# Rapport de fin de projet CPU/GPU

---

**Petit** Matthieu

**Caromel** Ulysse

**Morvan** Djemsay

**Encadrant** - MARIKO DUNSEATH

**Année universitaire 2023-2024**

# Table des matières

<b>1</b>	<b>Partie théorique</b>	<b>2</b>
1.1	Méthode de quadrature de Simpson . . . . .	2
1.2	Méthode de Gauss 2D . . . . .	2
1.3	Méthode de Runge-Kutta . . . . .	3
1.4	Méthode d'intégration de Monte-Carlo . . . . .	3
<b>2</b>	<b>Outils utilisés</b>	<b>4</b>
2.1	Open MP (CPU) . . . . .	4
2.2	MPI (CPU) . . . . .	5
2.3	CUDA (GPU) . . . . .	5
<b>3</b>	<b>Implémentation et résultats</b>	<b>7</b>
3.1	Méthode de Simpson . . . . .	7
3.1.1	Open MP . . . . .	7
3.1.2	MPI . . . . .	9
3.1.3	Cuda . . . . .	9
3.2	Gauss . . . . .	9
3.2.1	Open MP . . . . .	9
3.2.2	MPI . . . . .	9
3.2.3	Cuda . . . . .	9
3.3	Runge Kutta . . . . .	9
3.3.1	Open MP . . . . .	9
3.3.2	MPI . . . . .	9
3.3.3	Cuda . . . . .	10
3.4	Monte Carlo . . . . .	10
3.4.1	Open MP . . . . .	10
3.4.2	MPI . . . . .	10
3.4.3	Cuda . . . . .	10
3.5	Analyse des résultats . . . . .	10
<b>4</b>	<b>Quelques explications du code</b>	<b>11</b>
4.1	Exécution . . . . .	11
4.2	Graph Python . . . . .	11
4.3	Remarques . . . . .	11

# 1 Partie théorique

## 1.1 Méthode de quadrature de Simpson

La méthode de quadrature de Simpson est une technique numérique utilisée pour estimer l'intégrale numériquement. Elle est basée sur l'approximation d'une fonction par un polynôme quadratique entre chaque paire de points adjacents.

Supposons que nous ayons une fonction  $f(x)$  que nous voulons intégrer sur l'intervalle  $[a, b]$ . La méthode de Simpson divise cet intervalle en sous-intervalles de largeur égale  $h = \frac{b-a}{n}$ , où  $n$  est un nombre pair.

L'approximation de l'intégrale sur chaque sous-intervalle  $[x_i, x_{i+2}]$  est donnée par :

$$\int_{x_i}^{x_{i+2}} f(x) dx \approx \frac{h}{3} [f(x_i) + 4f(x_{i+1}) + f(x_{i+2})]$$

La somme de ces approximations sur tous les sous-intervalles donne l'estimation finale de l'intégrale :

$$\int_a^b f(x) dx \approx \frac{h}{3} [f(a) + 4f(x_1) + 2f(x_2) + \dots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(b)]$$

La méthode de Simpson est souvent plus précise que les méthodes de quadrature plus simples comme la méthode des rectangles, car elle prend en compte les variations de la fonction sur chaque sous-intervalle.

## 1.2 Méthode de Gauss 2D

La méthode de quadrature Gaussienne en deux dimensions (Gauss 2D) est utilisée pour estimer numériquement l'intégrale d'une fonction  $f(x, y)$  sur une région bidimensionnelle définie par  $a \leq x \leq b$  et  $c \leq y \leq d$ .

Supposons que nous ayons une fonction à intégrer sur cette région. La méthode de Gauss 2D utilise un ensemble de points et de poids associés pour approximer l'intégrale. La formule générale pour cette approximation est donnée par :

$$\iint_R f(x, y) dx dy \approx \sum_{i=1}^n \sum_{j=1}^m w_{ij} \cdot f(x_i, y_j)$$

où  $n$  et  $m$  sont le nombre de points dans les directions  $x$  et  $y$ , respectivement. Les points  $x_i$  et  $y_j$  sont les emplacements des points de quadrature, et  $w_{ij}$  sont les poids associés à ces points.

Une formule spécifique de quadrature Gaussienne 2D pour un quadrilatère est donnée par :

$$\iint_R f(x, y) dx dy \approx \sum_{i=1}^n \sum_{j=1}^m w_{ij} \cdot f\left(\frac{1}{2}(1 + \xi_i)x + \frac{1}{2}(1 - \xi_i)y, \frac{1}{2}(1 + \eta_j)x + \frac{1}{2}(1 - \eta_j)y\right)$$

où  $\xi_i$  et  $\eta_j$  sont les points de quadrature et  $w_{ij}$  sont les poids associés.

La méthode de quadrature Gaussienne 2D offre une précision supérieure à la quadrature de Gauss unidimensionnelle, et elle est souvent utilisée pour résoudre numériquement des intégrales sur des domaines bidimensionnels complexes.

### 1.3 Méthode de Runge-Kutta

La méthode de Runge-Kutta est une technique numérique utilisée pour résoudre des équations différentielles ordinaires (EDO). La forme la plus courante est la méthode de Runge-Kutta d'ordre 4 (RK4), qui est souvent utilisée en raison de son équilibre entre précision et complexité.

Supposons que nous ayons une EDO du premier ordre sous la forme :

$$\frac{dy}{dt} = f(t, y)$$

La méthode RK4 consiste en les étapes suivantes, où  $h$  est la taille du pas de discrétisation :

$$\begin{aligned}k_1 &= h \cdot f(t_n, y_n) \\k_2 &= h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\k_3 &= h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \\k_4 &= h \cdot f(t_n + h, y_n + k_3)\end{aligned}$$

La mise à jour de la solution à chaque pas est alors donnée par :

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

où  $y_n$  est la valeur de la solution à l'instant  $t_n$ .

Cette méthode offre une meilleure précision par rapport aux méthodes de pas fixe plus simples, mais elle nécessite également plus de calculs. Cependant, elle est largement utilisée en pratique pour sa robustesse et sa polyvalence.

### 1.4 Méthode d'intégration de Monte-Carlo

La méthode d'intégration de Monte Carlo est une technique utilisée en mathématiques pour estimer des valeurs numériques complexes, notamment les intégrales de fonctions dans des domaines à plusieurs dimensions. Elle repose sur des principes probabilistes et tire son nom du célèbre casino de Monte Carlo à Monaco, associant le hasard à des calculs numériques.

L'idée principale de cette méthode consiste à estimer une valeur en utilisant des échantillons aléatoires dans un domaine donné. Prenons l'exemple d'une intégrale définie sur un domaine  $D$  dans le plan. Plutôt que d'utiliser des méthodes traditionnelles d'intégration, on peut estimer la valeur de cette intégrale en générant aléatoirement des points dans  $D$ .

Imaginons que nous voulions calculer l'intégrale  $I = \int \int_D f(x, y) dx dy$ . La méthode de Monte Carlo fonctionne approximativement comme suit :

1. Génération de points aléatoires : On génère un grand nombre de points aléatoires  $(x_i, y_i)$  dans le domaine  $D$  selon une distribution choisie (souvent uniforme ou gaussienne).
2. Évaluation de la fonction : On évalue la fonction  $f(x, y)$  pour chaque point  $(x_i, y_i)$ .
3. Calcul de la moyenne pondérée : On calcule la moyenne des valeurs de  $f(x, y)$  évaluées pour les points générés et on multiplie cette moyenne par la mesure de l'ensemble  $D$ . Cette mesure est souvent estimée par le rapport entre le nombre de points dans  $D$  et le nombre total de points générés.

La formule d'estimation de Monte Carlo pour l'intégrale devient alors :

$$I \approx A \cdot \frac{1}{N} \sum_{i=1}^N f(x_i, y_i)$$

Où  $A$  est la mesure de  $D$ ,  $N$  est le nombre de points générés et  $(x_i, y_i)$  sont les coordonnées des points aléatoires.

L'avantage clé de cette méthode est sa flexibilité et sa capacité à traiter des problèmes complexes dans des espaces à dimensions élevées. Cependant, sa précision dépend fortement du nombre de points aléatoires générés : plus le nombre de points est élevé, plus l'estimation de l'intégrale sera précise, mais cela peut nécessiter beaucoup de ressources computationnelles.

La méthode de Monte Carlo offre une approche probabiliste puissante pour estimer numériquement des quantités complexes en utilisant des échantillons aléatoires. Son utilisation s'étend dans de nombreux domaines, de la physique à la finance, en raison de sa flexibilité et de sa capacité à traiter des problèmes difficiles à résoudre analytiquement.

## 2 Outils utilisés

### 2.1 Open MP (CPU)

OpenMP (Open Multi-Processing) est une API (Interface de Programmation Applicative) qui facilite la programmation parallèle en C, C++, et Fortran. Son objectif principal est de permettre aux développeurs d'exploiter les architectures parallèles des processeurs multi-cœurs de manière simple et portable.

Voici quelques points clés à propos d'OpenMP :

- **Parallelisme explicite** : OpenMP permet aux programmeurs d'exprimer le parallélisme dans leur code de manière explicite à l'aide de directives de compilation. Ces directives sont des annotations spéciales ajoutées au code source.
- **Directives pragmatiques** : Les directives OpenMP sont écrites sous forme de commentaires pragmatiques qui sont ignorés par les compilateurs qui ne supportent pas OpenMP. Cela permet au code source d'être compilé et exécuté sur des machines qui ne prennent pas en charge OpenMP sans erreur.
- **Tâches parallèles et boucles** : OpenMP permet la parallélisation de boucles et de sections de code avec des directives telles que `#pragma omp parallel for` pour paralléliser une boucle, ou `#pragma omp parallel` pour diviser le code en sections parallèles.
- **Gestion automatique de l'ordonnancement** : OpenMP offre une gestion automatique de l'ordonnancement des tâches parallèles, simplifiant ainsi la tâche du programmeur en ce qui concerne la gestion des threads et l'ordonnancement des tâches.
- **Support multi-plateforme** : OpenMP est supporté par de nombreux compilateurs sur différentes plates-formes, ce qui rend le code portable entre différentes architectures.

En résumé, OpenMP est une solution efficace pour ajouter du parallélisme aux applications existantes sans avoir à réécrire complètement le code. Cela facilite la création de programmes performants sur des architectures multi-cœurs.

## 2.2 MPI (CPU)

MPI (Message Passing Interface) est une norme pour la programmation parallèle utilisée pour développer des applications sur des architectures distribuées ou parallèles. Contrairement à OpenMP qui se concentre sur les architectures multi-cœurs partagées, MPI est conçu pour gérer la communication entre différents processus s'exécutant sur des nœuds distincts d'un cluster.

Voici quelques points clés à propos de MPI :

- **Modèle de programmation distribuée** : MPI s'appuie sur un modèle de programmation distribuée où chaque nœud d'un système peut avoir sa propre mémoire et exécute son propre processus. Les processus communiquent entre eux à l'aide de messages.
- **Passage de messages** : La communication entre les processus est réalisée via le passage explicite de messages. Les processus s'envoient des messages pour échanger des données ou coordonner leurs activités.
- **Abstraction des communications** : MPI fournit une abstraction des communications en permettant aux programmeurs d'utiliser des opérations de communication comme `MPI_Send` et `MPI_Recv` pour envoyer et recevoir des messages.
- **Point à point et collectif** : MPI prend en charge à la fois les communications point à point (entre deux processus) et les communications collectives (impliquant plusieurs processus). Les opérations collectives incluent des fonctionnalités telles que la diffusion, la réduction, la barrière, etc.
- **Indépendance d'architecture** : Comme OpenMP, MPI est conçu pour être indépendant de l'architecture matérielle, ce qui signifie qu'un code MPI peut être exécuté sur divers types de clusters sans nécessiter de modifications significatives.

En résumé, MPI est une norme de programmation parallèle qui permet de développer des applications distribuées en utilisant un modèle de passage de messages. Il est largement utilisé dans le domaine de la simulation, de l'analyse de données massives et d'autres domaines nécessitant une puissance de calcul parallèle sur des clusters de machines.

## 2.3 CUDA (GPU)

CUDA (Compute Unified Device Architecture) est une architecture de calcul parallèle développée par NVIDIA. Elle permet d'utiliser les GPU (Graphical Processing Unit) pour accélérer des tâches de calcul intensif. CUDA est particulièrement utilisé pour le calcul parallèle sur les cartes graphiques NVIDIA.

Voici quelques points clés à propos de CUDA :

- **Modèle de programmation parallèle sur GPU** : CUDA permet aux programmeurs d'utiliser la puissance de calcul massivement parallèle des GPU. Il s'appuie sur un modèle de programmation parallèle où des threads s'exécutent simultanément sur les cœurs du GPU.
- **Kernels CUDA** : Le code CUDA s'exécute sur le GPU sous la forme de kernels. Un kernel est une fonction qui est exécutée par un grand nombre de threads sur le GPU.
- **Hétérogénéité** : CUDA permet l'exécution de code sur le CPU (hôte) et le GPU (dispositif) simultanément. Le CPU gère les tâches de coordination et de gestion générale, tandis que le GPU effectue des calculs massivement parallèles.
- **Hierarchie des threads et des blocs** : Les threads CUDA sont organisés en blocs, et les blocs peuvent être organisés en grilles. Cette hiérarchie permet une gestion fine du parallélisme.

- **Gestion explicite de la mémoire :** Les programmeurs doivent gérer explicitement le transfert des données entre le CPU et le GPU, ainsi que la gestion de la mémoire sur le GPU (mémoire globale, mémoire partagée, etc.).

En résumé, CUDA permet d'exploiter la puissance de calcul des GPU pour accélérer des tâches parallèles. Il offre un modèle de programmation hétérogène avec une gestion explicite de la mémoire et des directives spéciales pour définir des kernels qui s'exécutent sur le GPU. Cette approche est particulièrement efficace pour des tâches intensives en calcul, comme le rendu graphique, la simulation physique, et l'apprentissage profond.

### 3 Implémentation et résultats

On présente ici les résultats pour chaque méthode et chaque bibliothèque de parallélisation.

Pour chaque méthode, on présente une étude de l'erreur en fonction du nombre de subdivision ainsi qu'une étude du temps en fonction du nombre de subdivision.

Les outils OpenMP et MPI ont été utilisés avec un maximum de 8 processeurs, tandis que pour CUDA, la configuration utilisé est la suivante :

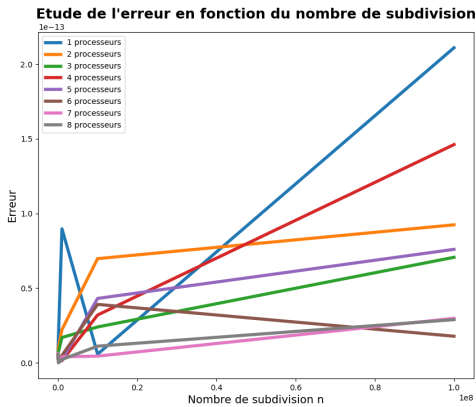
#### A COMPLETER

On présente ensuite une étude sur le temps d'exécution en fonction du nombre de subdivision, ainsi qu'une étude sur l'erreur en fonction du nombre de subdivision et cela pour chaque méthode (Simpson, Runge Kutta, Gauss 2D, Monte Carlo) et pour chaque parallélisation (Cuda, OpenMP, MPI).

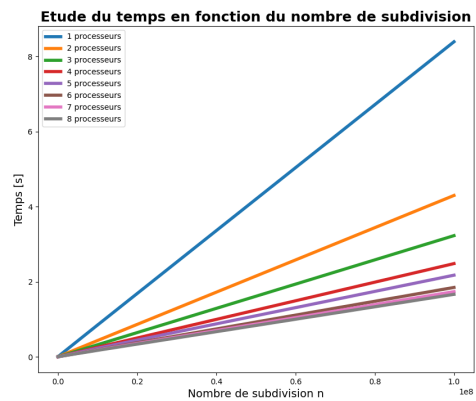
Nous affichons dans un premier temps tout les graphiques obtnus, une analyse sera donnée par la suite en partie (3.5).

#### 3.1 Méthode de Simpson

##### 3.1.1 Open MP



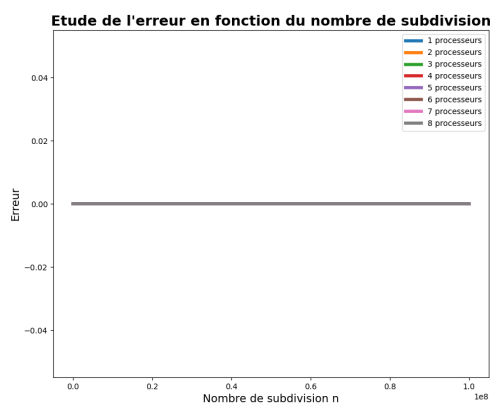
(a) Erreur



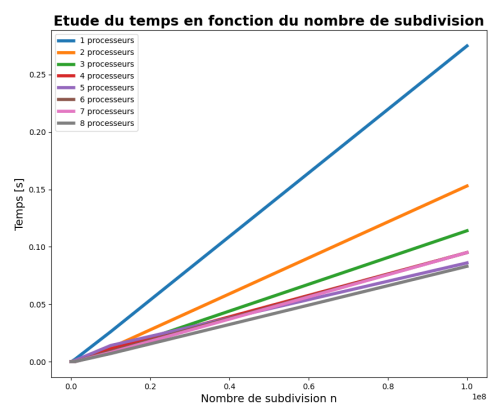
(b) Temps

FIGURE 1 – Méthode de Simpson : Open MP



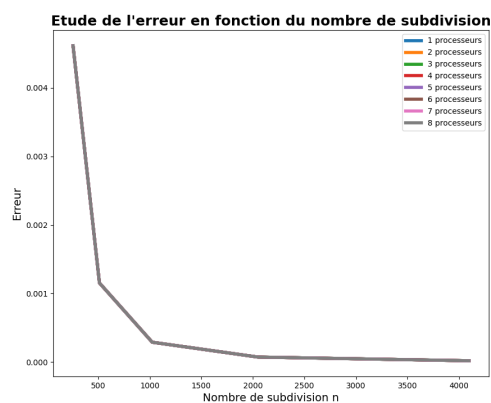


(a) Erreur

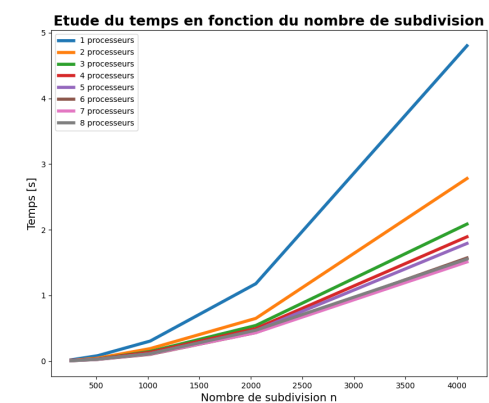


(b) Temps

FIGURE 2 – Méthode de Simpson : MPI

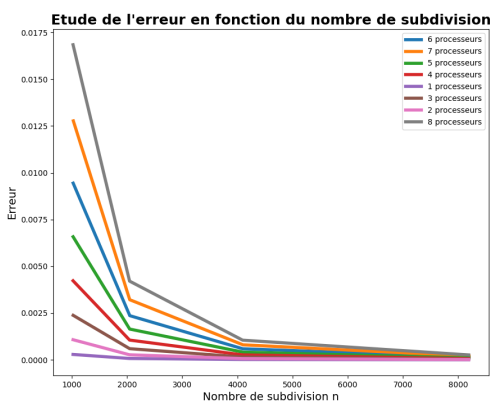


(a) Erreur

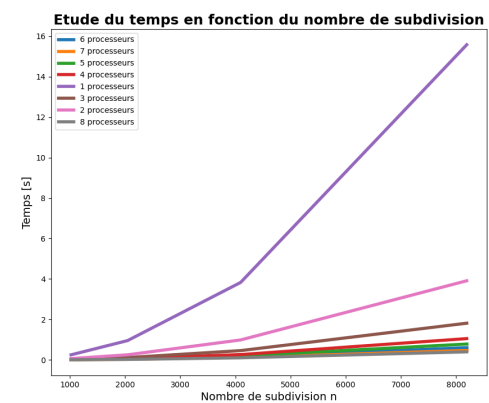


(b) Temps

FIGURE 3 – Méthode de Gaus 2D : Open MP



(a) Erreur



(b) Temps

FIGURE 4 – Méthode de Gauss 2D : MPI

### 3.1.2 MPI

### 3.1.3 Cuda

## 3.2 Gauss

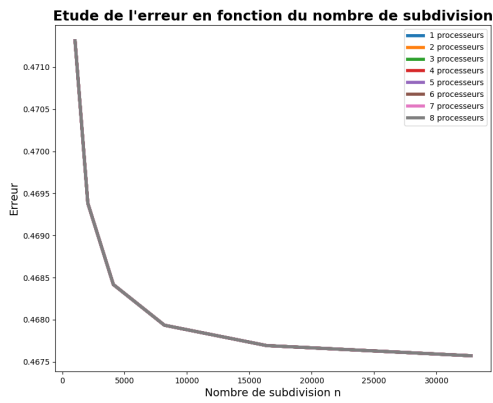
### 3.2.1 Open MP

### 3.2.2 MPI

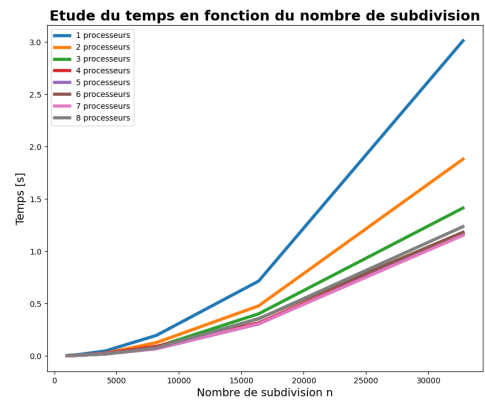
### 3.2.3 Cuda

## 3.3 Runge Kutta

### 3.3.1 Open MP



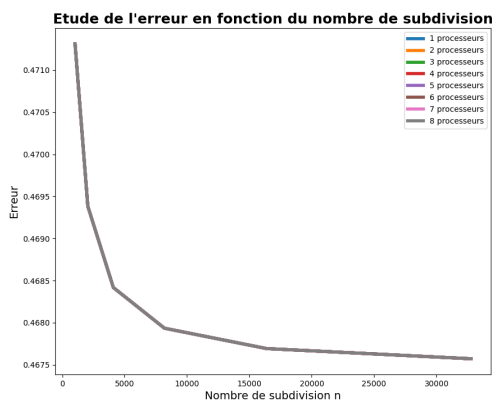
(a) Erreur



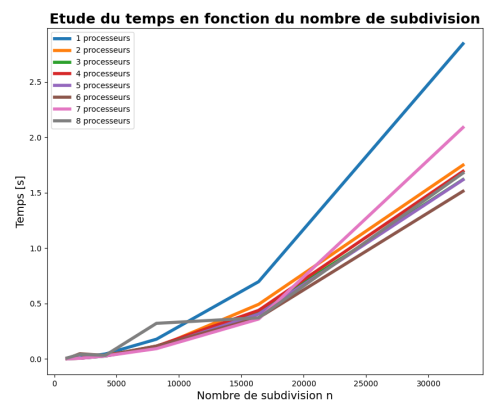
(b) Temps

FIGURE 5 – Méthode de Runge Kutta : Open MP

### 3.3.2 MPI



(a) Erreur



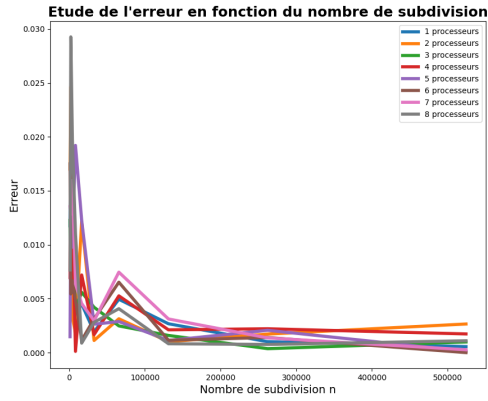
(b) Temps

FIGURE 6 – Méthode de Runge Kutta : MPI

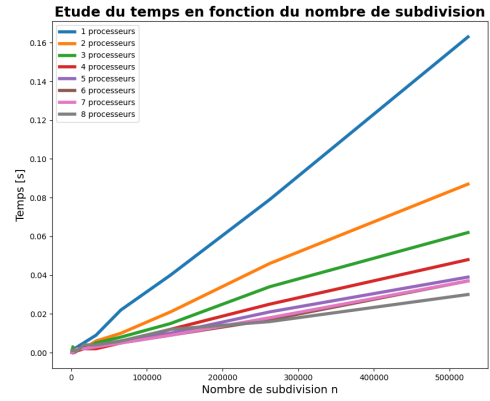
### 3.3.3 Cuda

## 3.4 Monte Carlo

### 3.4.1 Open MP



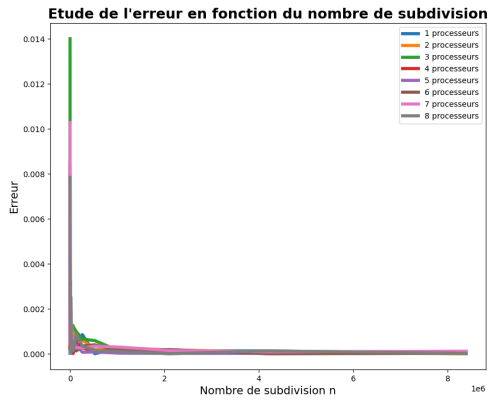
(a) Erreur



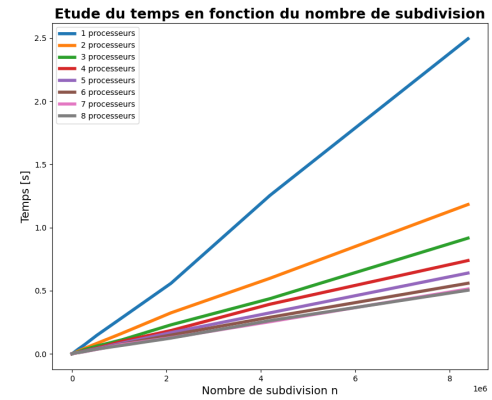
(b) Temps

FIGURE 7 – Méthode de Monte Carlo : Open MP

### 3.4.2 MPI



(a) Erreur



(b) Temps

FIGURE 8 – Méthode de Monte Carlo : MPI

### 3.4.3 Cuda

A COMPLETER

## 3.5 Analyse des résultats

A COMPLETER

## 4 Quelques explications du code

### 4.1 Exécution

Entrez dans le fichier `parametres_Machine` le nombre de processeur max que vous souhaitez utiliser.

Des scripts bash sont disponibles dans le dossier 'Bashes', ils permettent de lancer les codes pour chaque méthode et chaque parallélisation :

- Simpson :
- Gauss2D :
- Runge Kutta
- MonteCarlo

Pour les codes en CUDA, ils se trouvent dans le dossier Cuda.

Des scripts PowerShell sont également disponibles dans le dossier `PowerShell` afin de pouvoir exécuter les codes sous Windows.

### 4.2 Graph Python

Le fichier `graph.py` dans le sous répertoire `CodesPython` permet d'afficher les données dans le répertoire `Results`. Il prend en entrée le nom préfix des données que vous voulez afficher. Par exemple pour afficher les données pour la méthode de Simpson avec Open MP, placez vous dans le répertoire `CodesPython` et tapez :

```
python3 graph.py "simp_0p_MP"
```

### 4.3 Remarques

Le code pour la méthode de Gauss 2D utilise la librairie Eigen, il sera nécessaire de linker la librairie et donc de devoir modifier légèrement les fichiers `gauss2DMPI.cpp` et `gauss2DOpenMP.cpp`.