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

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 :

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

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

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

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 ξ_i et η_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 :

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

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 de Monte Carlo

1.5 Méthode de Monte Carlo pour l'intégration numérique

La méthode de Monte Carlo pour l'intégration numérique est une approche probabiliste qui repose sur l'utilisation d'échantillons aléatoires pour estimer une intégrale. Contrairement à la méthode de quadrature de Simpson qui divise l'intervalle en sous-intervalles égaux, la méthode de Monte Carlo génère des points aléatoires dans l'intervalle d'intérêt.

Supposons que nous ayons une fonction continue $f(x)$ définie sur l'intervalle $[a, b]$ que nous voulons intégrer. L'idée fondamentale de la méthode de Monte Carlo est d'estimer l'intégrale en utilisant la moyenne pondérée des valeurs de la fonction échantillonnées aléatoirement sur l'intervalle.

La formule de base de la méthode de Monte Carlo pour l'intégration numérique est donnée par :

$$\int_a^b f(x)\,dx \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i),$$

où N est le nombre d'échantillons aléatoires, et x_i sont des points générés aléatoirement dans l'intervalle $[a, b]$. Ces points sont obtenus à partir d'une distribution uniforme.

Pour appliquer la méthode de Monte Carlo à une intégrale spécifique, il est possible de suivre ces étapes :

1. Choisissez le nombre d'échantillons N.
2. Générez N échantillons aléatoires x_i dans l'intervalle $[a, b]$.
3. Calculez l'estimation de l'intégrale en utilisant la formule (1).

À la différence de la méthode de quadrature de Simpson qui divise l'intervalle en sous-intervalles et utilise une approximation polynomiale quadratique sur chaque sous-intervalle, la méthode de Monte Carlo n'impose pas de structure particulière à la division de l'intervalle. Elle s'adapte naturellement aux fonctions avec des variations complexes et peut être plus robuste dans certains cas.

Cependant, la précision de la méthode de Monte Carlo dépend fortement du nombre d'échantillons générés. Un grand nombre d'échantillons est généralement nécessaire pour obtenir des résultats précis, mais la méthode peut être efficace pour des fonctions avec des caractéristiques difficiles à modéliser de manière analytique.

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 se` 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.

— **Hiérarchie 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 bibiliothèque de parallélisation.

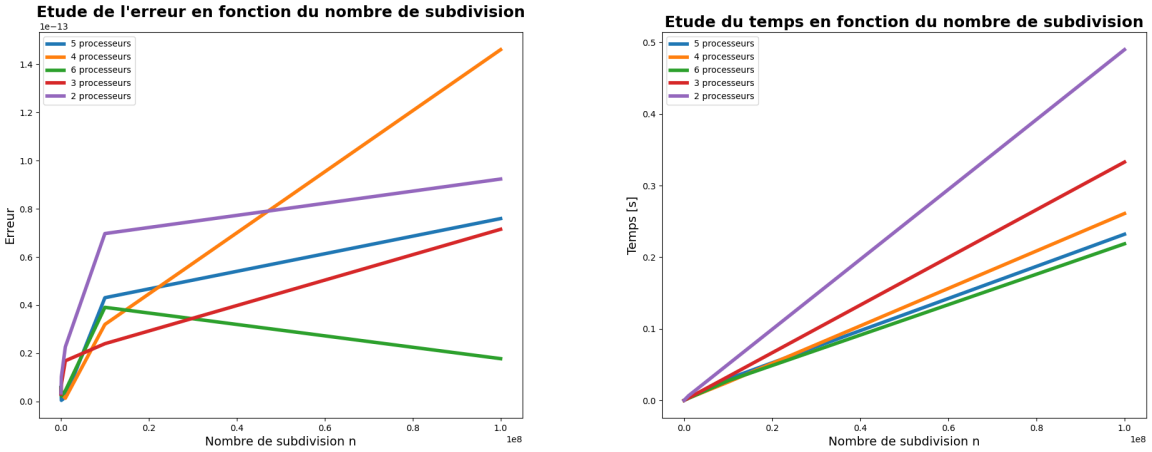
Pour chaque méthode, on présente une étude de l'erreur en fonction du nombre de subdivision ainsi qu'une étude du temps en fonction du nombre de subdivision.

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

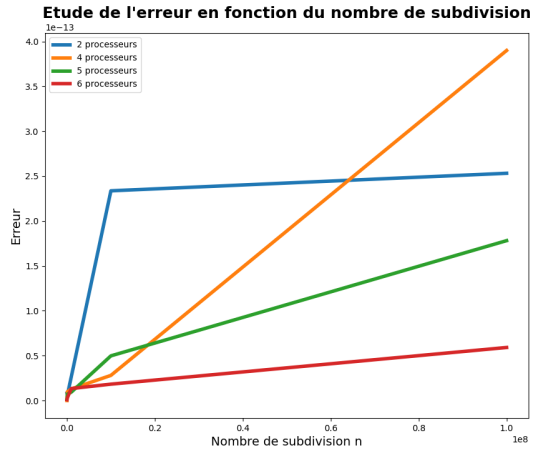
3.1 Méthode de Simpson

3.1.1 Open MP

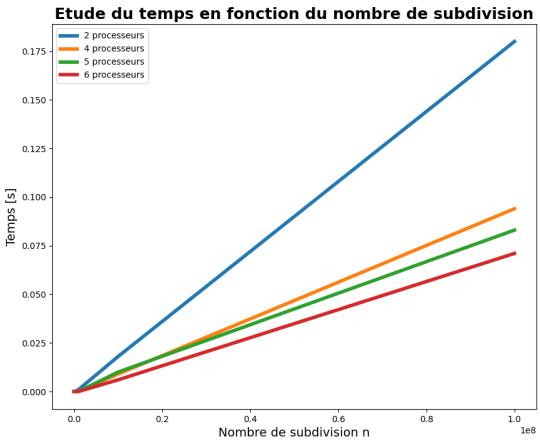


(a) Erreur (b) Temps

FIGURE 1 – Méthode de simpson : Open MP

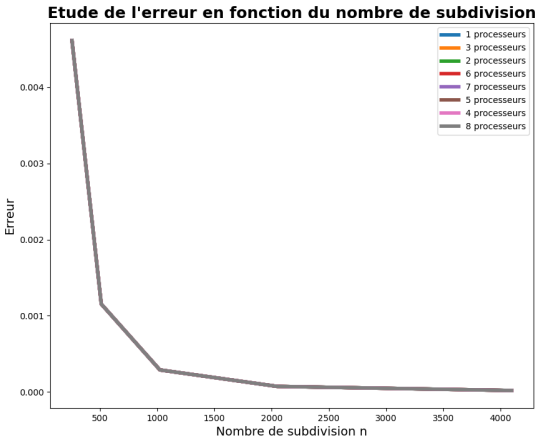


(a) Erreur

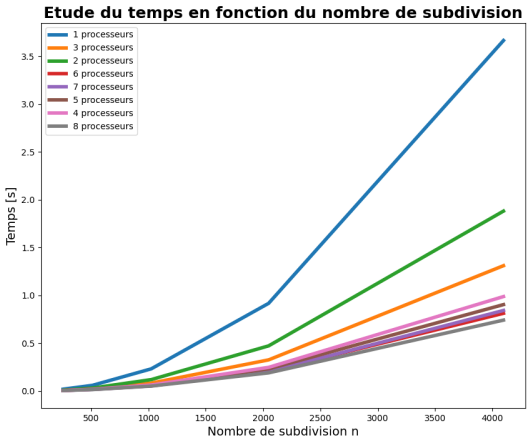


(b) Temps

FIGURE 2 – Méthode de simpson : MPI

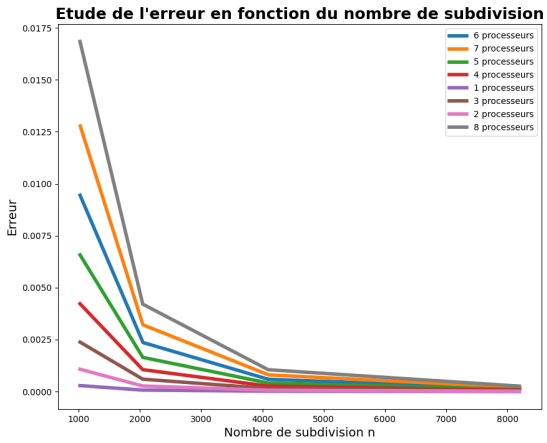


(a) Erreur

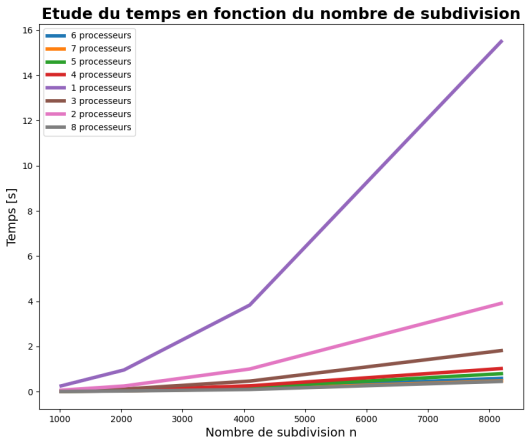


(b) Temps

FIGURE 3 – Méthode de Gaus 2D : Open MP

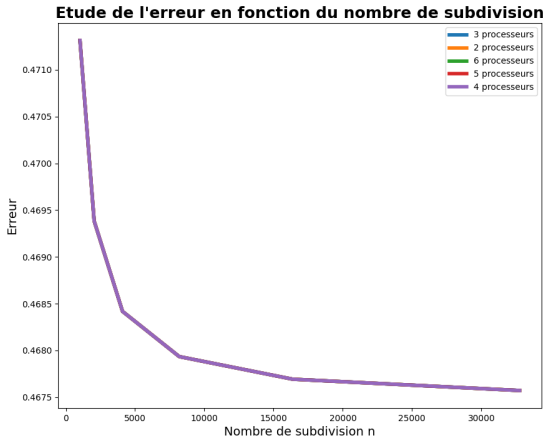


(a) Erreur

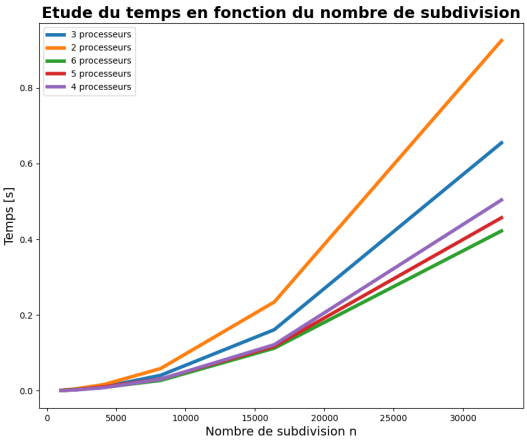


(b) Temps

FIGURE 4 – Méthode de Gauss 2D : MPI



(a) Erreur



(b) Temps

FIGURE 5 – Méthode de Runge Kutta : Open MP

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

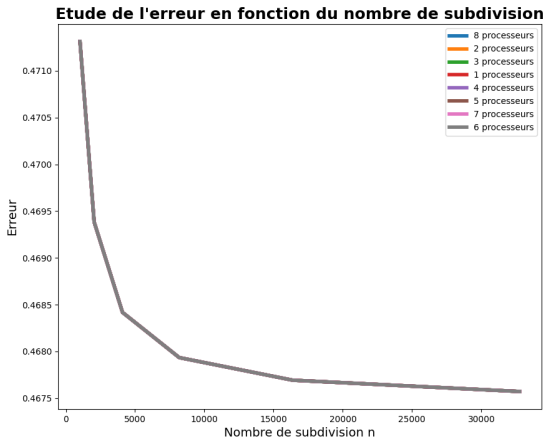
3.3.2 MPI

3.3.3 Cuda

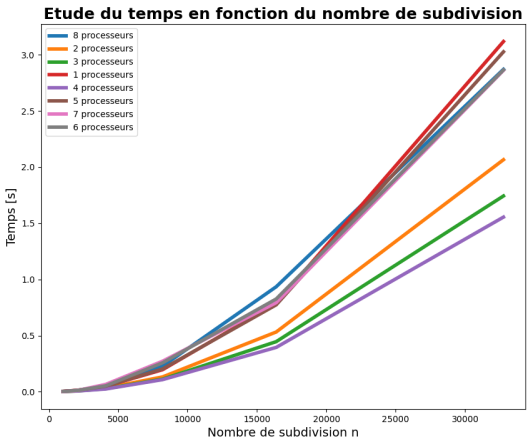
3.4 Monte Carlo

3.4.1 Open MP

3.4.2 MPI

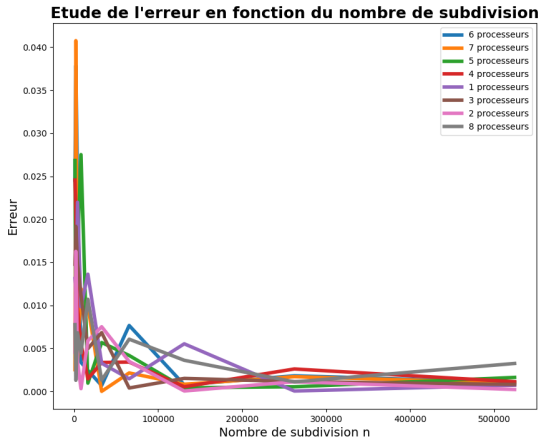


(a) Erreur

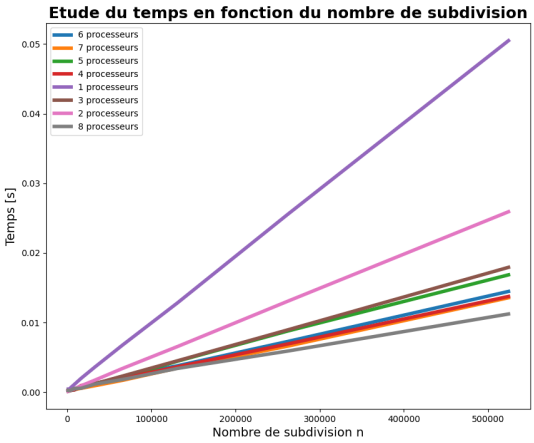


(b) Temps

FIGURE 6 – Méthode de Runge Kutta : MPI

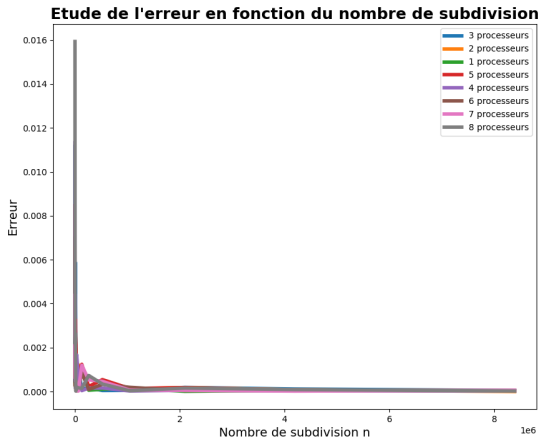


(a) Erreur

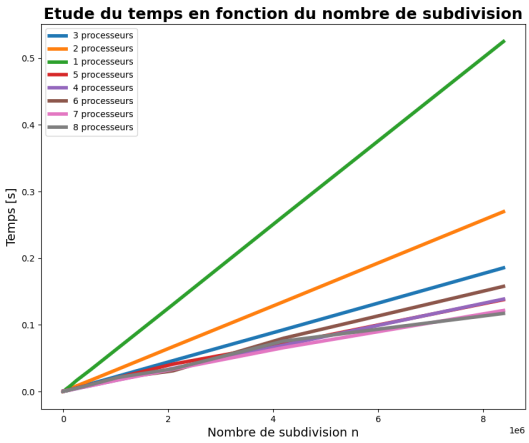


(b) Temps

FIGURE 7 – Méthode de Monte Carlo : Open MP



(a) Erreur



(b) Temps

FIGURE 8 – Méthode de Monte Carlo : MPI