# Rapport OS202 TD2

Gianluca Baghino Gómez

Février 2024

## 1 Exercice sur l'interblocage

### 1.1 Scénario où il y a interblocage

Interblocage, un défi fréquent en programmation parallèle, survient lorsque plusieurs processus se retrouvent dans une impasse, chacun attendant indéfiniment que l'autre libère une ressource critique. En MPI, cette situation peut se produire lorsque deux processus ou plus se bloquent mutuellement en attendant des événements qui dépendent les uns des autres.

Considérons le code MPI suivant, où deux processus (0 et 1) échangent des messages:

```
if (rank == 0) {
    // Le processus 0 attend de recevoir
    // un message du processus 1 avec l'étiquette 101.
    MPI_Recv(rcvbuf, count, MPI_DOUBLE, 1, 101, commGlob, &status);

    // Le processus 0 envoie un message
    // au processus 1 avec l'étiquette 102.
    MPI_Send(sndbuf, count, MPI_DOUBLE, 1, 102, commGlob);
} else if (rank == 1) {
    // Le processus 1 attend de recevoir
    // un message du processus 0 avec l'étiquette 102.
    MPI_Recv(rcvbuf, count, MPI_DOUBLE, 0, 102, commGlob, &status);

    // Le processus 1 envoie un message
    // au processus 0 avec l'étiquette 101.
    MPI_Send(sndbuf, count, MPI_DOUBLE, 0, 101, commGlob);
}
```

Dans ce scénario, les processus s'engagent dans un échange de messages où chaque processus attend un message de l'autre avant de pouvoir avancer. Plus précisément:

- Le processus 0 attend de recevoir un message du processus 1 avec l'étiquette 101, puis envoie un message au processus 1 avec l'étiquette 102.
- Le processus 1 attend de recevoir un message du processus 0 avec l'étiquette 102, puis envoie un message au processus 0 avec l'étiquette 101.

Cela crée une situation où les deux processus se bloquent mutuellement: le processus 0 attend un message du processus 1 qui attend lui-même un message du processus 0. De même, le processus 1 attend un message du processus 0 qui attend un message du processus 1. Ainsi, aucun des processus ne peut avancer, entraînant un interblocage.

Pour résoudre ce problème, il est impératif d'assurer que chaque opération d'envoi soit appariée avec une opération de réception correspondante, avec les étiquettes correctes et les expéditeurs appropriés. Cela garantira que les processus peuvent progresser sans se bloquer mutuellement dans des attentes interminables.

## 1.2 Scénario où il n'y a pas d' interblocage

Considérons maintenant un scénario sans interblocage :

```
MPI_Request req;
MPI_Status status;

if (rank == 0) {
    // Processus 0 initialise une réception non bloquante
    // des données du processus 1 avec l'étiquette 101.
    MPI_Irecv(rcvbuf, count, MPI_DOUBLE, 1, 101, commGlob, &req);

    // Processus 0 envoie des données
    // au processus 1 avec l'étiquette 102.
    MPI_Send(sndbuf, count, MPI_DOUBLE, 1, 102, commGlob);

    // Processus 0 attend que la réception non bloquante se termine.
    MPI_Wait(&req, &status);
} else if (rank == 1) {
    // Processus 1 initialise une réception non bloquante
    // des données du processus 0 avec l'étiquette 102.
    MPI_Irecv(rcvbuf, count, MPI_DOUBLE, 0, 102, commGlob, &req);

    // Processus 1 envoie des données
    // au processus 0 avec l'étiquette 101.
    MPI_Send(sndbuf, count, MPI_DOUBLE, 0, 101, commGlob);

    // Processus 1 attend que la réception non bloquante se termine.
    MPI_Wait(&req, &status);
}
```

Dans ce code, une communication bidirectionnelle est établie entre deux processus, l'un de rang 0 et l'autre de rang 1, en utilisant des opérations d'envoi et de réception MPI non bloquantes. Chaque processus attend la fin des opérations de réception non bloquantes avant de poursuivre.

## 1.3 Probabilité d'Interblocage

Dans le premier ensemble de codes, l'absence d'utilisation de fonctions non bloquantes peut entraîner un blocage si un processus atteint une opération d'envoi avant que l'autre ne puisse recevoir. Le même risque existe dans le deuxième ensemble de codes, même si des fonctions non bloquantes sont utilisées (MPI_Irecv et MPI_Wait).

La probabilité de blocage dépend de plusieurs facteurs, dont la synchronisation correcte des opérations d'envoi et de réception entre les processus, l'équilibre des communications et la gestion appropriée des opérations non bloquantes. La probabilité de blocage peut varier en fonction de la complexité du code et de la logique de communication entre les processus.

## 2 Analyse de l'efficacité de la parallélisation et de l'évolution des performances

Alice a partiellement parallélisé un code sur une machine à mémoire distribuée. Pour un ensemble de données spécifique, elle observe que la partie parallèle représente 90% du temps d'exécution du programme, tandis que la partie séquentielle constitue les 10% restants.

**En utilisant la loi d'Amdahl, pouvez-vous prédire l'accélération maximale qu'Alice peut obtenir avec son code (en considérant $n \gg 1$)?**

Commençons par appliquer la loi d'Amdahl pour analyser la situation. Étant donné que $n$ est considérablement supérieur à un, on peut utiliser l'expression finale de la loi :

$$S(n) = \frac{ts}{F \cdot ts + \frac{(1-f) \cdot ts}{n}} = \frac{n}{1 + (n-1)f} \rightarrow \lim_{n \to \infty} \rightarrow \frac{1}{f}$$

En utilisant les données fournies, où l'exécution parallèle représente 90% du temps d'exécution du programme et la partie séquentielle 10% (représentée par $f$), on calcule l'accélération théorique du programme parallélisé ($S(n)$):

$$S(n) = \frac{1}{0.1} = 10$$

**Pour cet ensemble de données spécifique, quel nombre de nœuds de calcul semble raisonnable pour éviter un gaspillage excessif des ressources CPU ?**

Dans ce scénario, déterminer un nombre raisonnable de nœuds de calcul dépend de plusieurs facteurs, notamment la proportion de code parallélisable, qui s'élève à 90%. Alors que la fraction séquentielle n'est que de 10%, elle étend inévitablement le temps d'exécution total d'au moins 10% au-delà du temps de parallélisation. Par conséquent, la réduction du temps d'exécution dépend principalement de l'efficacité de la parallélisation.

Le choix du nombre optimal de nœuds de calcul implique également de tenir compte des ressources financières disponibles pour le projet. Cependant, il est essentiel de noter qu'il existe une limite théorique au-delà de laquelle l'ajout de processeurs supplémentaires ne produira pas d'améliorations significatives du temps d'exécution.

Ce phénomène de rendements décroissants peut être dû à des limitations de communication, des dépendances de données ou d'autres contraintes inhérentes au problème. Par conséquent, déterminer le nombre optimal de nœuds de calcul nécessite une analyse approfondie de la nature du programme, des caractéristiques de l'ensemble de données, des coûts associés et de la capacité de parallélisation effective.

**Alice constate une accélération maximale de quatre en augmentant le nombre de nœuds de calcul pour son ensemble de données spécifique. Si le volume de données est doublé et en supposant une complexité algorithmique parallèle**

**linéaire, quelle accélération maximale peut-elle espérer en utilisant la loi de Gustafson ?**

Tout d'abord, rappelons la loi de Gustafson, exprimée comme suit :

$$S(n) = \frac{ts + n \cdot tp}{ts + tp}$$

Avant de dupliquer l'ensemble de données, on peut reformuler la loi de la manière suivante :

$$4 = \frac{0.1 + n \cdot (0.9)}{1}$$

En résolvant pour $n$ afin de déterminer le nombre de nœuds, on obtient :

$$n = \frac{4 - 0.1}{0.9} = \frac{13}{3}$$

En tenant compte de ces valeurs et des déclarations suivantes :

- Le temps d'exécution du code séquentiel ($ts$) reste inchangé quelle que soit la taille des données en entrée.
- Le temps d'exécution du code parallèle ($tp$) dépend linéairement du volume de données en entrée et a désormais une valeur de $2 \times ts$.

On applique la loi de Gustafson pour déterminer la nouvelle vitesse de traitement du programme. En simplifiant l'équation :

$$S(n) = \frac{ts + 2 \cdot n \cdot tp}{ts + 2 \cdot tp} = \frac{0.1 + 2 \cdot n \cdot (0.9)}{0.1 + 2 \cdot (0.9)} = 4.16$$

Alice peut donc anticiper une accélération de 4,16 avec son ensemble de données augmenté.

# 3 Analyse du Calcul Parallèle de l'Ensemble de Mandelbrot

**À partir du code séquentiel mandelbrot.cpp, faire une partition équitable par ligne de l'image à calculer pour distribuer le calcul sur les nbp tâches exécutées par l'utilisateur puis rassembler l'image sur le processus zéro pour la sauvegarder. Calculer le temps d'exécution pour différents nombre de tâches et calculer le speedup. Comment interpréter les résultats obtenus ?**

Le fichier 'mandelbrot.py' contient le code permettant de calculer l'ensemble de Mandelbrot de manière parallèle en utilisant le module `multiprocessing` de Python. Voici une analyse détaillée de la procédure, des résultats obtenus et des conclusions tirées :

- Importation des Bibliothèques

  Le code commence par importer les bibliothèques nécessaires telles que `numpy`, `matplotlib.pyplot`, `multiprocessing`, `PIL`, et `time`. Ces bibliothèques sont essentielles pour effectuer des calculs numériques, créer des graphiques et gérer les processus parallèles.

- Définition de la Classe MandelbrotSet

  La classe `MandelbrotSet` est définie pour calculer l'ensemble de Mandelbrot. Elle contient des méthodes pour déterminer la convergence d'un point complexe sur l'ensemble de Mandelbrot et pour compter les itérations nécessaires pour qu'un point diverge. Cette classe fournit la base pour le calcul parallèle de l'ensemble de Mandelbrot.

- Calcul Parallèle de l'Ensemble de Mandelbrot

  La fonction `calculate_row` est définie pour calculer une ligne de l'ensemble de Mandelbrot en parallèle. Cette fonction est utilisée pour répartir les calculs sur plusieurs processus, ce qui permet d'accélérer le calcul global de l'ensemble de Mandelbrot.

- Calcul du Temps d'Exécution et du Speedup

  Dans le bloc principal du code, une boucle itère sur différents nombres de processus. À chaque itération, le code utilise `multiprocessing.Pool` pour paralléliser le calcul des lignes de l'ensemble de Mandelbrot. Le temps d'exécution est mesuré pour chaque configuration de processus, et le speedup est calculé pour évaluer l'efficacité de la parallélisation.

```
# Importation des bibliothèques nécessaires
import numpy as np
# Pour les opérations sur les tableaux
import matplotlib.pyplot as plt
# Pour la création de graphiques
import multiprocessing
# Pour le traitement multi-processus
from PIL import Image
# Pour la manipulation des images
from time import time
# Pour mesurer le temps d'exécution

# Définition de la classe MandelbrotSet
# pour calculer l'ensemble de Mandelbrot
class MandelbrotSet:
    def __init__(self, max_iterations=50, escape_radius=2.0):
        # Initialisation des paramètres
        # de l'ensemble de Mandelbrot
        self.max_iterations = max_iterations
        # Nombre maximal d'itérations
        self.escape_radius = escape_radius
        # Rayon d'échappement

    # Méthode pour calculer la convergence
```

```
    # d'un point complexe sur l'ensemble de Mandelbrot
    def convergence(self, c: complex, smooth=False, clamp=True) -> float:
        # Calcul de la convergence du point c
        value = self.count_iterations(c, smooth) / self.max_iterations
        # Assurer que la valeur de convergence
        # reste entre 0 et 1 si clamp est True
        return max(0.0, min(value, 1.0)) if clamp else value

    # Méthode pour compter les itérations
    # nécessaires pour qu'un point complexe diverge
    def count_iterations(self, c: complex, smooth=False) -> int or float:
        # Initialisation des variables
        z = 0
        # Boucle pour effectuer les itérations
        for iter in range(self.max_iterations):
            z = z*z + c
            # Vérifier si le point a divergé
            if abs(z) > self.escape_radius:
                # Retourner le nombre d'itérations ou
                # une valeur lissée si smooth est True
                if smooth:
                    return iter + 1 - np.log(np.log(abs(z))) / np.log(2)
                return iter
        return self.max_iterations

# Fonction pour calculer une ligne
# de l'ensemble de Mandelbrot
def calculate_row(y):
    global mandelbrot_set, width, scaleX, scaleY
    row = np.empty(width)
    # Boucle pour calculer chaque pixel de la ligne y
    for x in range(width):
        # Calcul du point complexe correspondant au pixel (x, y)
        c = complex(-2. + scaleX * x, -1.125 + scaleY * y)
        # Calcul de la convergence du
        # point c et stockage dans le tableau row
        row[x] = mandelbrot_set.convergence(c, smooth=True)
    return row

# Point d'entrée du programme
if __name__ == '__main__':
    # Initialisation des paramètres pour
    # le calcul de l'ensemble de Mandelbrot
    mandelbrot_set = MandelbrotSet(max_iterations=50, escape_radius=10)
    width, height = 1024, 1024
    scaleX = 3. / width
    scaleY = 2.25 / height

    # Liste pour stocker les temps d'exécution
    # pour différents nombres de processus
    times = []
    # Boucle pour tester différents nombres de processus
    for num_processes in range(1, multiprocessing.cpu_count() + 1):
        # Mesurer le temps de début d'exécution
        deb = time()
        # Créer un pool de processus avec
        # le nombre spécifié de processus
        with multiprocessing.Pool(processes=num_processes) as pool:
            # Mapper la fonction calculate_row
            # sur les lignes de l'image
            results = pool.map(calculate_row, range(height))
```

```
        # Convertir les résultats en un tableau numpy
        convergence = np.array(results)
        # Mesurer le temps de fin d'exécution
        fin = time()
        # Calculer le temps d'exécution
        execution_time = fin - deb
        # Ajouter le temps d'exécution à la liste des temps
        times.append(execution_time)
        # Afficher le temps d'exécution
        # pour le nombre de processus actuel
        print(f"Temps du calcul de l'ensemble de Mandelbrot avec
                {num_processes} processus : {execution_time}")

    # Calculer le temps d'exécution pour un seul processus
    single_process_time = times[0]
    # Calculer le speedup pour
    # chaque configuration de processus
    speedups = [single_process_time / time for time in times]

    # Afficher le tableau des temps
    # d'exécution et des speedups
    print("Nombre Processus - Temps du Calcul [s] - Speedup [S(n)]")
    for num_processes, execution_time, speedup
        in zip(range(1, multiprocessing.cpu_count() + 1)
                , times, speedups):
        print(f"{num_processes} - {execution_time:.4f} - {speedup:.4f}")

    # Tracer le graphique du speedup
    # en fonction du nombre de processus
    plt.subplot(2, 1, 1)
    plt.plot(range(1, multiprocessing.cpu_count() + 1), speedups, marker='o')
    plt.xlabel('Nombre de Processus')
    plt.ylabel('Speedup [S(n)]')
    plt.title('Speedup en fonction du Nombre de Processus')
    plt.grid(True)

    # Tracer le graphique du temps de calcul
    # en fonction du nombre de processus
    plt.subplot(2, 1, 2)
    plt.plot(range(1, multiprocessing.cpu_count() + 1)
            , times, marker='o', color='r')
    plt.xlabel('Nombre de Processus')
    plt.ylabel('Temps de Calcul [s]')
    plt.title('Temps de Calcul en fonction du Nombre de Processus')
    plt.grid(True)

    # Ajuster les paramètres de disposition
    plt.tight_layout()
    # Afficher les graphiques
    plt.show()

    # Rassembler l'image et la sauvegarder
    image = np.vstack(results)
    image = (image / np.max(image)) * 255
    Image.fromarray(image.astype(np.uint8)).save('mandelbrot.png')
```

### Analyse des Résultats

| Nombre de Processus | Temps du Calcul [s] | Speedup [S(n)] |
|---|---|---|
| 1 | 4.0120 | 1.0000 |
| 2 | 2.0862 | 1.9231 |
| 3 | 1.4720 | 2.7256 |
| 4 | 1.0975 | 3.6556 |
| 5 | 0.9123 | 4.3979 |
| 6 | 0.7878 | 5.0928 |
| 7 | 0.7152 | 5.6100 |
| 8 | 0.6553 | 6.1221 |
| 9 | 0.6887 | 5.8252 |
| 10 | 0.6556 | 6.1195 |
| 11 | 0.6504 | 6.1686 |
| 12 | 0.6630 | 6.0512 |
| 13 | 0.6636 | 6.0455 |
| 14 | 0.6808 | 5.8928 |
| 15 | 0.6765 | 5.9310 |
| 16 | 0.6404 | 6.2647 |

Table 1: Résultats du calcul

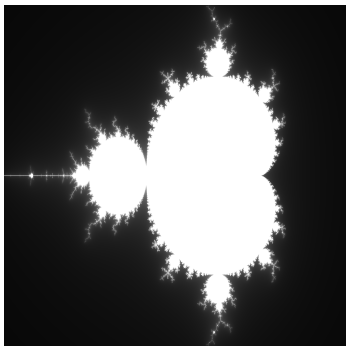

Figure 1

Les résultats obtenus montrent une diminution significative du temps de calcul avec l'augmentation du nombre de processus. Le speedup, qui représente l'accélération du calcul par rapport à une exécution séquentielle, est également élevé. Cependant, le graphique du speedup en fonction du nombre de processus montre une croissance initiale rapide suivie d'un plateau, indiquant que l'efficacité parallèle atteint un plafond à un certain nombre de processus.

**Conclusions**

Le code parallèle a réussi à accélérer le calcul de l'ensemble de Mandelbrot en exploitant les ressources de la machine. Les résultats obtenus démontrent l'efficacité du parallélisme dans la résolution de problèmes computationnels intensifs. Cependant, il est important de noter que l'efficacité parallèle peut être limitée par des facteurs tels que les coûts de communication et la répartition des tâches. Une analyse approfondie des performances parallèles est essentielle pour optimiser l'utilisation des ressources disponibles et maximiser les gains d'efficacité.

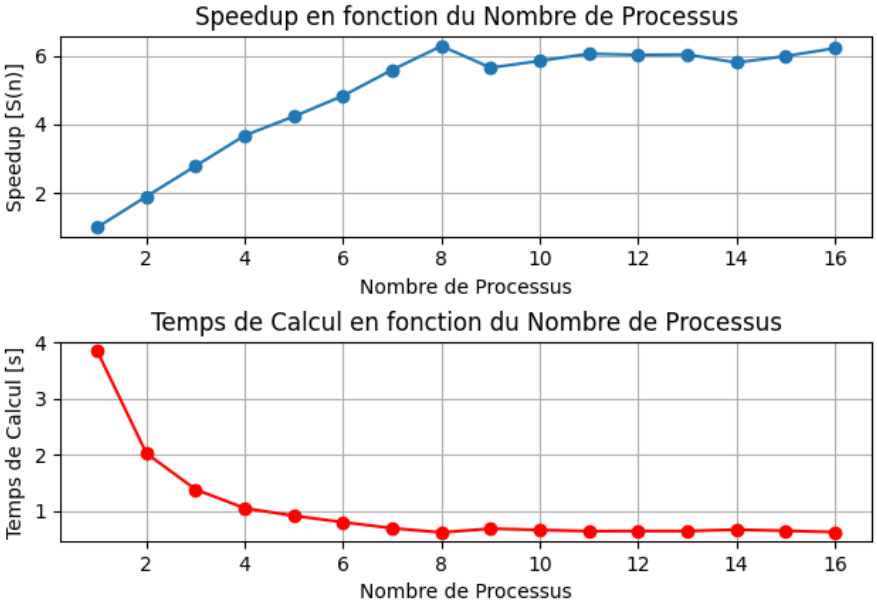


Figure 2

**Mettre en œuvre une stratégie maître–esclave pour distribuer les différentes lignes de l'image à calculer. Calculer le speedup avec cette nouvelle approche. Qu'en conclure?**

```
# Importation des bibliothèques nécessaires
import numpy as np
# Pour les opérations sur les tableaux
import matplotlib.pyplot as plt
# Pour la création de graphiques
import multiprocessing
# Pour le traitement multi-processus
from PIL import Image
# Pour la manipulation des images
from time import time
# Pour mesurer le temps d'exécution
from multiprocessing import Pool, Manager, cpu_count

# Définition de la classe MandelbrotSet
# pour calculer l'ensemble de Mandelbrot
class MandelbrotSet:
    def __init__(self, max_iterations=50, escape_radius=2.0):
        # Initialisation des paramètres
        # de l'ensemble de Mandelbrot
        self.max_iterations = max_iterations
        # Nombre maximal d'itérations
        self.escape_radius = escape_radius
        # Rayon d'échappement

    # Méthode pour calculer la convergence
    # d'un point complexe sur l'ensemble de Mandelbrot
    def convergence(self, c: complex, smooth=False, clamp=True) -> float:
        # Calcul de la convergence du point c
        value = self.count_iterations(c, smooth) / self.max_iterations
        # Assurer que la valeur de convergence
```

```
        # reste entre 0 et 1 si clamp est True
        return max(0.0, min(value, 1.0)) if clamp else value

    # Méthode pour compter les itérations
    # nécessaires pour qu'un point complexe diverge
    def count_iterations(self, c: complex, smooth=False) -> int or float:
        # Initialisation des variables
        z = 0
        # Boucle pour effectuer les itérations
        for iter in range(self.max_iterations):
            z = z*z + c
            # Vérifier si le point a divergé
            if abs(z) > self.escape_radius:
                # Retourner le nombre d'itérations ou
                # une valeur lissée si smooth est True
                if smooth:
                    return iter + 1 - np.log(np.log(abs(z))) / np.log(2)
                return iter
        return self.max_iterations

# Fonction pour calculer une ligne
# de l'ensemble de Mandelbrot
def calculate_row(y):
    global mandelbrot_set, width, scaleX, scaleY
    row = np.empty(width)
    # Boucle pour calculer chaque pixel de la ligne y
    for x in range(width):
        # Calcul du point complexe
        # correspondant au pixel (x, y)
        c = complex(-2. + scaleX * x, -1.125 + scaleY * y)
        # Calcul de la convergence du
        # point c et stockage dans le tableau row
        row[x] = mandelbrot_set.convergence(c, smooth=True)
    return row

# Fonction pour le travail des processus esclaves
def worker_process(args):
    queue, y = args
    row = calculate_row(y)
    queue.put((y, row))

# Point d'entrée du programme
if __name__ == '__main__':
    # Initialisation des paramètres pour
    # le calcul de l'ensemble de Mandelbrot
    mandelbrot_set = MandelbrotSet(max_iterations=50, escape_radius=10)
    width, height = 1024, 1024
    scaleX = 3. / width
    scaleY = 2.25 / height

    # Liste pour stocker les temps d'exécution
    # pour différents nombres de processus
    times = []
    # Boucle pour tester différents nombres de processus
    for num_processes in range(1, multiprocessing.cpu_count() + 1):
        # Mesurer le temps de début d'exécution
        deb = time()
        # Créer une file de messages pour la communication
        # entre le maître et les esclaves
        with Manager() as manager:
            queue = manager.Queue()
```

```
            # Créer un pool de processus avec le
            # nombre spécifié de processus
            with Pool(processes=num_processes) as pool:
                # Mapper la fonction worker_process
                # sur les lignes de l'image
                pool.map(worker_process, [(queue, y) for y in range(height)])
                # Collecter les résultats des processus esclaves
                results = [queue.get() for _ in range(height)]
        # Convertir les résultats en un tableau numpy
        results.sort()  # Assurer que les résultats sont triés par y
        convergence = np.array([row for y, row in results])
        # Mesurer le temps de fin d'exécution
        fin = time()
        # Calculer le temps d'exécution
        execution_time = fin - deb
        # Ajouter le temps d'exécution à la liste des temps
        times.append(execution_time)
        # Afficher le temps d'exécution pour le nombre de processus actuel
        print(f"Temps du calcul de l'ensemble de Mandelbrot avec
                {num_processes} processus : {execution_time}")

    # Calculer le temps d'exécution pour un seul processus
    single_process_time = times[0]
    # Calculer le speedup pour chaque configuration de processus
    speedups = [single_process_time / time for time in times]

    # Afficher le tableau des temps d'exécution et des speedups
    print("Nombre Processus - Temps du Calcul [s] - Speedup [S(n)]")
    for num_processes, execution_time, speedup
        in zip(range(1, multiprocessing.cpu_count() + 1)
               , times, speedups):
        print(f"{num_processes} - {execution_time:.4f} - {speedup:.4f}")

    # Tracer le graphique du speedup
    # en fonction du nombre de processus
    plt.subplot(2, 1, 1)
    plt.plot(range(1, multiprocessing.cpu_count() + 1), speedups, marker='o')
    plt.xlabel('Nombre de Processus')
    plt.ylabel('Speedup [S(n)]')
    plt.title('Speedup en fonction du Nombre de Processus')
    plt.grid(True)

    # Tracer le graphique du temps de calcul
    # en fonction du nombre de processus
    plt.subplot(2, 1, 2)
    plt.plot(range(1, multiprocessing.cpu_count() + 1)
             , times, marker='o', color='r')
    plt.xlabel('Nombre de Processus')
    plt.ylabel('Temps de Calcul [s]')
    plt.title('Temps de Calcul en fonction du Nombre de Processus')
    plt.grid(True)

    # Ajuster les paramètres de disposition
    plt.tight_layout()
    # Afficher les graphiques
    plt.show()

    # Rassembler l'image et la sauvegarder
    image = np.vstack(convergence)
    image = (image / np.max(image)) * 255
    Image.fromarray(image.astype(np.uint8)).save('mandelbrot_master_slave.png')
```

Dans le code, une stratégie maître-esclave a été adoptée pour répartir la charge de travail du calcul de l'ensemble de Mandelbrot sur plusieurs processus. Cette stratégie vise à améliorer l'efficacité computationnelle en exploitant les capacités de traitement parallèle. Les principales modifications apportées au code comprennent l'introduction d'une fonction `worker_process` et l'utilisation d'une `Manager.Queue` pour la communication inter-processus.

Tout d'abord, la fonction `worker_process` a été incorporée pour représenter les tâches effectuées par chaque processus esclave. Cette fonction accepte une file d'attente et un indice de ligne (`y`) en tant qu'arguments, permettant aux processus esclaves de placer leurs résultats dans la file d'attente pour une récupération ultérieure par le processus maître. Le bloc principal d'exécution du code a été ajusté pour accommoder la stratégie maître-esclave. Dans ce bloc, le code itère sur différents nombres de processus, allant de 1 au nombre de cœurs CPU disponibles.

Un pool de processus est créé en utilisant `multiprocessing.Pool`, chaque processus étant assigné pour exécuter la fonction `worker_process`. Grâce à `Pool.map()`, le processus maître répartit efficacement la charge de travail en associant la fonction `worker_process` à chaque indice de ligne (`y`) dans la plage de hauteur de l'image. Une fois que les processus esclaves ont terminé leurs calculs, le processus maître récupère les résultats de la file d'attente. Il attend patiemment que chaque processus esclave contribue ses résultats avant de les assembler dans l'image de Mandelbrot finale.

Après avoir collecté les résultats, le code procède au calcul du speedup pour chaque configuration de processus et le trace par rapport au nombre de processus utilisés. Cette évaluation nous permet d'évaluer les gains d'efficacité obtenus grâce au traitement parallèle.

En résumé, l'adoption de la stratégie maître-esclave optimise la distribution des tâches computationnelles, en exploitant les capacités de traitement parallèle pour améliorer les performances globales et réduire le temps de calcul. Cette approche facilite l'utilisation efficace des ressources disponibles et permet de mettre à l'échelle efficacement avec l'augmentation des demandes computationnelles.

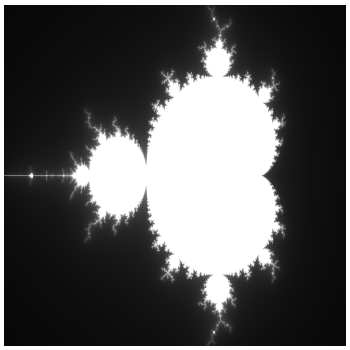

Figure 3

| Nombre de Processus | Temps du Calcul [s] | Speedup [S(n)] |
|---|---|---|
| 1 | 3.9912 | 1.0000 |
| 2 | 2.2245 | 1.7942 |
| 3 | 1.5723 | 2.5385 |
| 4 | 1.1826 | 3.3749 |
| 5 | 1.0371 | 3.8485 |
| 6 | 0.9310 | 4.2869 |
| 7 | 0.9804 | 4.0711 |
| 8 | 0.8543 | 4.6718 |
| 9 | 0.8236 | 4.8462 |
| 10 | 0.8164 | 4.8888 |
| 11 | 0.8176 | 4.8819 |
| 12 | 0.7918 | 5.0408 |
| 13 | 0.7624 | 5.2351 |
| 14 | 0.7344 | 5.4346 |
| 15 | 0.7495 | 5.3250 |
| 16 | 0.7243 | 5.5102 |

Table 2: Résultats du calcul

**Analyse des Résultats**

Premièrement, nous observons une diminution significative du temps de calcul lorsque le nombre de processus est augmenté. Cette réduction du temps de calcul est clairement illustrée par les valeurs du tableau, où le temps de calcul diminue presque de moitié lorsqu'on passe d'un seul processus à deux, et continue à diminuer progressivement à mesure que le nombre de processus augmente.

En outre, le speedup ($S(n)$), qui mesure l'accélération du calcul par rapport à l'exécution séquentielle, montre une tendance positive avec l'augmentation du nombre de processus.

Le speedup est calculé en divisant le temps d'exécution d'une exécution séquentielle par le temps d'exécution de l'exécution parallèle avec $n$ processus. Plus le speedup est élevé, plus l'efficacité de la parallélisation est grande.

Dans ce cas, le speedup augmente progressivement jusqu'à atteindre une valeur maximale avec un nombre optimal de processus, après quoi il commence à diminuer légèrement. Cela suggère qu'il existe une efficacité maximale de parallélisation pour cette tâche spécifique, et ajouter plus de processus au-delà de ce point n'apporte pas nécessairement de bénéfices significatifs en termes de réduction du temps de calcul.

**Conclusions**

Les résultats démontrent clairement les avantages de l'utilisation du traitement parallèle pour accélérer le calcul de l'ensemble de Mandelbrot. La stratégie de parallélisation adoptée permet une répartition efficace de la charge de travail, réduisant ainsi le temps de calcul global. Cependant, il est essentiel de choisir judicieusement le nombre de processus pour obtenir un équilibre optimal entre l'accélération du calcul et les coûts associés à la gestion des processus parallèles.

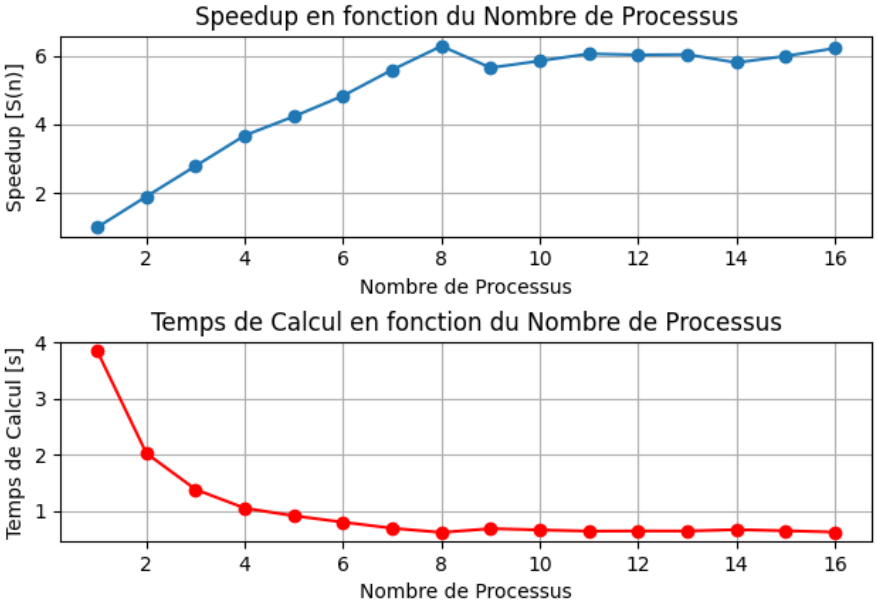


Figure 4

# 4 Produit matrice–vecteur

## 4.1 Produit parallèle matrice – vecteur par colonne

Dans cette approche, on a parallélisé le calcul du produit matrice-vecteur en découpant la matrice par colonnes. Voici les étapes réalisées :

1. **Initialisation de la matrice et du vecteur :** On a défini une matrice carrée $A$ de dimension $N$ et un vecteur $u$ de même dimension, où les éléments de la matrice sont définis par $A_{ij} = (i + j) \mod N$.

2. **Calcul du nombre de colonnes par tâche :** On a calculé le nombre de colonnes par tâche en divisant la dimension de la matrice par le nombre de tâches parallèles.

3. **Création des arguments pour chaque tâche :** On a créé des arguments pour chaque tâche, où chaque argument spécifie une tranche de colonnes sur laquelle la tâche doit effectuer le calcul.

4. **Calcul parallèle du produit matrice-vecteur par colonne :** On a utilisé la bibliothèque multiprocessing pour créer un pool de processus. Chaque processus est chargé de calculer le produit matrice-vecteur pour les colonnes qui lui ont été assignées.

5. **Assemblage des résultats partiels :** Une fois les calculs terminés, on a récupéré les résultats partiels de chaque tâche et les avons additionnés pour obtenir le vecteur résultant complet.

```python
import numpy as np
import multiprocessing

# Dimension du problème
dim = 120
# Nombre de tâches parallèles
nbp = 4

# Initialisation de la matrice A
A = np.array([[(i+j) % dim+1. for i in range(dim)] for j in range(dim)])

# Initialisation du vecteur u
u = np.array([i+1. for i in range(dim)])

# Fonction pour calculer le
# produit matrice-vecteur par colonne
def compute_column_product(start_col, end_col):
    # Initialisation du vecteur résultat partiel
    partial_result = np.zeros(dim)
    # Calcul du produit matrice-vecteur
    # pour chaque colonne assignée
    for col in range(start_col, end_col):
        partial_result += A[:, col] * u[col]
    return partial_result

if __name__ == '__main__':
    # Calcul du nombre de colonnes par tâche
    cols_per_task = dim // nbp
    # Création des arguments pour chaque tâche
    task_args = [(i * cols_per_task, (i+1) * cols_per_task)
                 for i in range(nbp)]

    # Création d'un pool de processus
    with multiprocessing.Pool(processes=nbp) as pool:
        # Calcul parallèle du
        # produit matrice-vecteur par colonne
        results = pool.starmap(compute_column_product, task_args)

    # Assemblage des résultats partiels
    # pour obtenir le vecteur résultant complet
    v = np.sum(results, axis=0)
    print("Vecteur résultant du produit matrice-vecteur par colonne :\n", v)

'''
Vecteur résultant du produit matrice-vecteur par colonne :
 [583220. 576080. 569060. 562160. 555380. 548720. 542180. 535760. 529460.
 523280. 517220. 511280. 505460. 499760. 494180. 488720. 483380. 478160.
 473060. 468080. 463220. 458480. 453860. 449360. 444980. 440720. 436580.
 432560. 428660. 424880. 421220. 417680. 414260. 410960. 407780. 404720.
 401780. 398960. 396260. 393680. 391220. 388880. 386660. 384560. 382580.
 380720. 378980. 377360. 375860. 374480. 373220. 372080. 371060. 370160.
 369380. 368720. 368180. 367760. 367460. 367280. 367220. 367280. 367460.
 367760. 368180. 368720. 369380. 370160. 371060. 372080. 373220. 374480.
 375860. 377360. 378980. 380720. 382580. 384560. 386660. 388880. 391220.
 393680. 396260. 398960. 401780. 404720. 407780. 410960. 414260. 417680.
 421220. 424880. 428660. 432560. 436580. 440720. 444980. 449360. 453860.
 458480. 463220. 468080. 473060. 478160. 483380. 488720. 494180. 499760.
 505460. 511280. 517220. 523280. 529460. 535760. 542180. 548720. 555380.
 562160. 569060. 576080.]
```

```
'''
```

### 4.2 Produit parallèle matrice – vecteur par ligne

Dans cette approche, on a parallélisé le calcul du produit matrice-vecteur en découpant la matrice par lignes. Voici les étapes réalisées :

1. **Initialisation de la matrice et du vecteur :** On a défini une matrice carrée $A$ de dimension $N$ et un vecteur $u$ de même dimension, où les éléments de la matrice sont définis par $A_{ij} = (i + j) \mod N$.

2. **Calcul du nombre de lignes par tâche :** On a calculé le nombre de lignes par tâche en divisant la dimension de la matrice par le nombre de tâches parallèles.

3. **Création des arguments pour chaque tâche :** On a créé des arguments pour chaque tâche, où chaque argument spécifie une tranche de lignes sur laquelle la tâche doit effectuer le calcul.

4. **Calcul parallèle du produit matrice-vecteur par ligne :** On a utilisé la bibliothèque multiprocessing pour créer un pool de processus. Chaque processus est chargé de calculer le produit matrice-vecteur pour les lignes qui lui ont été assignées.

5. **Assemblage des résultats partiels :** On a les calculs terminés, nous avons récupéré les résultats partiels de chaque tâche et les avons additionnés pour obtenir le vecteur résultant complet.

```
import numpy as np
import multiprocessing

# Dimension du problème
dim = 120
# Nombre de tâches parallèles
nbp = 4

# Initialisation de la matrice A
A = np.array([[(i+j) % dim+1. for i in range(dim)] for j in range(dim)])

# Initialisation du vecteur u
u = np.array([i+1. for i in range(dim)])

# Fonction pour calculer le
# produit matrice-vecteur par ligne
def compute_row_product(start_row, end_row):
    # Initialisation du vecteur résultat partiel
    partial_result = np.zeros(dim)
    # Calcul du produit matrice-vecteur
    # pour chaque ligne assignée
    for row in range(start_row, end_row):
        partial_result[row] = np.dot(A[row, :], u)
    return partial_result

if __name__ == '__main__':
```

```
    # Calcul du nombre de lignes par tâche
    rows_per_task = dim // nbp
    # Création des arguments pour chaque tâche
    task_args = [(i * rows_per_task, (i+1) * rows_per_task)
                 for i in range(nbp)]

    # Création d'un pool de processus
    with multiprocessing.Pool(processes=nbp) as pool:
        # Calcul parallèle du
        # produit matrice-vecteur par ligne
        results = pool.starmap(compute_row_product, task_args)

    # Assemblage des résultats partiels
    # pour obtenir le vecteur résultant complet
    v = np.sum(results, axis=0)
    print("Vecteur résultant du produit matrice-vecteur par ligne :\n", v)

'''
Vecteur résultant du produit matrice-vecteur par ligne :
 [583220. 576080. 569060. 562160. 555380. 548720. 542180. 535760. 529460.
 523280. 517220. 511280. 505460. 499760. 494180. 488720. 483380. 478160.
 473060. 468080. 463220. 458480. 453860. 449360. 444980. 440720. 436580.
 432560. 428660. 424880. 421220. 417680. 414260. 410960. 407780. 404720.
 401780. 398960. 396260. 393680. 391220. 388880. 386660. 384560. 382580.
 380720. 378980. 377360. 375860. 374480. 373220. 372080. 371060. 370160.
 369380. 368720. 368180. 367760. 367460. 367280. 367220. 367280. 367460.
 367760. 368180. 368720. 369380. 370160. 371060. 372080. 373220. 374480.
 375860. 377360. 378980. 380720. 382580. 384560. 386660. 388880. 391220.
 393680. 396260. 398960. 401780. 404720. 407780. 410960. 414260. 417680.
 421220. 424880. 428660. 432560. 436580. 440720. 444980. 449360. 453860.
 458480. 463220. 468080. 473060. 478160. 483380. 488720. 494180. 499760.
 505460. 511280. 517220. 523280. 529460. 535760. 542180. 548720. 555380.
 562160. 569060. 576080.]

'''
```

### Conclusions de produit matrice–vecteur

Les deux vecteurs résultants du produit matrice-vecteur par colonne et par ligne sont identiques. Cela signifie que les deux méthodes de calcul, par colonne et par ligne, produisent les mêmes résultats pour ce problème spécifique. Cela peut être dû à la nature de la matrice A et du vecteur u, ainsi qu'à la façon dont ils sont utilisés dans les calculs. Dans les deux approches, chaque élément du vecteur résultant est calculé en utilisant une combinaison linéaire des éléments correspondants de la matrice et du vecteur. Les opérations effectuées sont les mêmes dans les deux cas, mais l'ordonnancement des calculs diffère légèrement.

En général, la méthode de produit matrice-vecteur par ligne est souvent préférée dans les environnements parallèles, car elle exploite mieux la localité des données et la mise en cache, ce qui peut conduire à de meilleures performances. Cependant, dans ce cas spécifique, les deux méthodes donnent des résultats équivalents. Il est important de noter que la performance peut varier en fonction de la taille de la matrice, de la complexité des calculs et de la configuration matérielle du système sur lequel le code est exécuté. Il est donc recommandé de tester différentes approches et de mesurer les performances pour déterminer celle qui convient le mieux à chaque cas d'utilisation.