Systàmes parallêles et distribués
OS202 - TD2

Gallego, Natalia
AST
ENSTA Paris
natalia.gallego@ensta-paris.fr

I. Question 1.1

Reprendre l'exercice sur l'interblocage donné dans le cours et décrivez deux scénarios :

A. Scénario où il y a interblocage :

L'interblocage est une situation dans laquelle de nombreux processus s'attendent indéfiniment pour terminer leurs messages. Dans MPI, un blocage est une situation dans laquelle deux processus ou plus ne peuvent pas continuer à s'exécuter car chacun attend que l'autre libère une ressource.

```
if (rank==0){
    MPI_Recv(rcvbuf, count, MPI_DOUBLE, 1,
             101, commGlob, &status);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 1,
             102, commGlob);
}else if (rank==1){
    MPI_Recv(rcvbuf, count, MPI_DOUBLE, 0,
             102, commGlob, &status);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 0,
             101, commGlob);
}
```

Dans ce cas, le processus 1 attend de recevoir un message du processus 0 et le processus 0 attend de recevoir un message du processus 1. Alternativement, le processus 0 envoie un message au processus 1, et le processus 1 attend un message du processus 0 mais avec une mauvaise étiquette. Nous devons veiller à ce que chaque envoi ait un reçu correspondant avec la bonne étiquette et le bon expéditeur.

B. Scénario où il n'y a pas d'interblocage :

```
MPI_Request req;
MPI_Status status;

if (rank==0){
    MPI_Irecv(rcvbuf, count, MPI_DOUBLE, 1,
              101, commGlob, &req);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 1,
              102, commGlob);
    MPI_Wait(&req, &status);
}else if (rank==1){
    MPI_Irecv(rcvbuf, count, MPI_DOUBLE, 0,
              102, commGlob, &req);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 0,
             101, commGlob);
    MPI_Wait(&req, &status);
}
```

Ce code établit une communication bidirectionnelle entre deux processus de rang 0 et 1 en utilisant des opérations d'envoi et de réception MPI non bloquantes. Le processus de rang 0 envoie des données au processus de rang 1, et vice versa. Chaque processus attend ensuite la fin des opérations de réception non bloquantes avant de continuer.

C. Quelle est à votre avis la probabilité d'avoir un interblocage ? :

Dans le premier code, nous n'utilisons pas de fonctions non bloquantes, donc si le processus de rang 0 atteint la ligne **MPI˙Send** avant que le processus de rang 1 n'atteigne la ligne **MPI˙Recv**, il pourrait y avoir un blocage. Il en va de même si le processus de rang 1 atteint la ligne **MPI˙Send** avant que le processus de rang 0 n'atteigne la ligne **MPI˙Recv**.

Dans le deuxième code, nous utilisons des fonctions non bloquantes (**MPI˙Irecv** et **MPI˙Wait**). Cependant, lorsque vous utilisez **MPI˙Wait**, vous attendez la fin de l'opération de réception avant de continuer. Dans ce cas, la probabilité de blocage est similaire à celle du premier ensemble de codes, dans le sens où vous attendez la fin de la communication avant de continuer.

La probabilité de blocage dépend de plusieurs facteurs, tels que la bonne synchronisation des opérations d'envoi et de réception entre les processus, l'équilibre des communications et la gestion correcte des opérations non bloquantes.

La probabilité de blocage peut dépendre de la complexité du code et de la logique de communication entre les processus.

II. Question 1.2

A. En utilisant la loi d'Amdhal, pouvez-vous prédire l'accélération maximale que pourra obtenir Alice avec son code ? :

Pour commencer, nous utiliserons la loi d'Amdahl pour résoudre l'exercice. Puisqu'on nous dit que la valeur de n est bien supérieure à un, nous pouvons en utiliser la dernière expression.

$$S(n) = \frac{t_s}{F.t_s + \frac{(1-f)t_s}{n}} = \frac{n}{1+(n-1)f} -> \lim_{n\to\infty} -> \frac{1}{f} \quad (1)$$

En utilisant les données données dans l'exercice, où l'on nous dit que la partie qui s'exécute en parallèle représente 90%

du temps d'exécution du programme en temps de traitement, et donc la partie séquentielle est égale à 10 % (représentant **f** dans l'équation), on calcule le accélération théorique d'un programme parallélisé **(S(n))**.

$$S(n) = \frac{1}{0.1} = 10 \tag{2}$$

B. À votre avis, pour ce jeu de donné spécifique, quel nombre de noeuds de calcul semble-t-il raisonnable de prendre pour ne pas trop gaspiller de ressources CPU ? :

À mon avis, pour cet ensemble de données spécifique, la détermination d'un nombre raisonnable de nœuds de calcul dépend de plusieurs facteurs, dont la proportion parallélisable du programme, qui est de 90%.

La partie séquentielle représentant 10% du temps d'exécution, il est inévitable que le temps d'exécution total soit au moins 10% plus long que le temps de parallélisation. Ainsi, la réduction du temps d'exécution dépendra principalement de l'efficacité de la parallélisation.

Le choix du nombre de nœuds de calcul doit également tenir compte des ressources financières disponibles pour mettre en œuvre le projet. Cependant, il est essentiel de noter qu'il existe une limite théorique à laquelle l'ajout de processeurs supplémentaires n'entraînera pas une amélioration significative du temps d'exécution de la partie parallèle.

En d'autres termes, il existe un point de rendement décroissant auquel l'ajout de ressources supplémentaires n'apporte pas d'avantages proportionnels en termes d'exécution. Cela peut être dû à des limitations de communication, à des dépendances de données ou à d'autres limitations inhérentes à la nature du 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.

C. Une accélération maximale de quatre est obtenue en augmentant le nombre de nœuds de calcul pour l'ensemble de données spécifique. En doublant la quantité de donnée à traiter, et en supposant la complexité de l'algorithme parallèle linéaire, quelle accélération maximale peut espérer Alice en utilisant la loi de Gustafson ? :

La première chose est de reconnaître la loi de Gustafson, donnée par l'expression suivante. Pour cela on considérera que $t_s + t_p = 1$ (une unité de temps).

$$S(n) = \frac{t_s + n.t_p}{t_s + t_p} \tag{3}$$

En utilisant les données fournies par l'exercice, avant de dupliquer les données, nous pouvons réécrire la loi pour qu'elle ressemble à l'expression suivante:

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

Enfin, en résolvant n, pour déterminer le nombre de nœuds, obtenant ainsi une valeur de:

$$n = \frac{4 - 0.1}{0.9} = \frac{13}{3} \tag{5}$$

En utilisant maintenant ces données et les déclarations suivantes:

- Le temps d'exécution de la partie séquentielle du code (t_s) est indépendant du volume des données d'entrée. Cela continuera donc avec une valeur de 0.1.
- Le temps d'exécution de la partie parallèle du code (t_p) dépend linéairement du volume de données d'entrée. Il aura donc désormais une valeur de $2 * t_s$.

Nous utilisons la loi de Gustafson pour trouver la nouvelle vitesse de traitement du programme. En le laissant comme ceci :

$$S(n) = \frac{t_s + 2nt_p}{t_s + 2t_p} = \frac{0.1 + 2n(0.9)}{0.1 + 2(0.9)} = 4.16 \tag{6}$$

Alice peut donc s'attendre à une accélération de **4.16** avec sa nouvelle base de données.

III. Question 1.3

A. Faire une partition équitable par ligne de l'image à calculer pour distribuer le calcul sur les nbp tâches exécutées. Calculer le temps d'exécution pour différents nombre de tâches et calculer le speedup :

Pour accomplir la tâche demandée, le code python (**mandelbrot.py**) a été complété.

Ce code calcule la convergence d'un point complexe sur l'ensemble de Mandelbrot. Le code utilise ensuite le multi-traitement pour paralléliser le calcul des ensembles de Mandelbrot avec différents nombres de processus.

```
    def convergence(self, c: complex,
                    smooth=False, clamp=True) ->
    float:
        value = self.count_iter(c, smooth)/self.
    max_iterations
        return max(0.0, min(value, 1.0)) if clamp
    else value

    def count_iter(self, c: complex,
                         smooth=False) -> int |
    float:
        z:    complex
        iter: int
        if c.real*c.real+c.imag*c.imag < 0.0625:
            return self.max_iterations
        if (c.real+1)*(c.real+1)+c.imag*c.imag <
    0.0625:
            return self.max_iterations
```

```
        if (c.real > -0.75) and (c.real < 0.5):
            ct = c.real-0.25 + 1.j * c.imag
            ctnrm2 = abs(ct)
            if ctnrm2 < 0.5*(1-ct.real/max(ctnrm2,
    1.E-14)):
                return self.max_iterations
        z = 0
        for iter in range(self.max_iterations):
            z = z*z + c
            if abs(z) > self.escape_radius:
                if smooth:
                    return iter + 1 - log(log(abs(z)
    ))/log(2)
                return iter
        return self.max_iterations

if __name__ == '__main__':
    times = []
    for num_processes in range(1, multiprocessing.
    cpu_count() + 1):
        deb = time()
        with multiprocessing.Pool(processes=
    num_processes) as pool:
            results = pool.map(calculate_row, range(
    height))
```

Remarque : Pour revoir le code complet, rendez-vous sur le dépôt GitHub dans les annexes.

Méthode de convergence : Cette méthode calcule la convergence d'un point complexe c sur l'ensemble de Mandelbrot. Appelez la méthode **countIter** pour obtenir le nombre d'itérations requis pour que c échappe à l'ensemble de Mandelbrot. Normalise la valeur obtenue en la divisant par le nombre maximum d'itérations.

Méthode countIter : Cette méthode effectue des itérations sur l'ensemble de Mandelbrot pour un point complexe **c**. Il contient des conditions d'échappement rapide pour les points connus appartenant à des ensembles de Mandelbrot particuliers. Itère jusqu'à ce que le point s'échappe ou que le nombre maximum d'itérations soit atteint. Renvoie le nombre d'itérations ou une valeur lissée basée sur le logarithme.

Bloc main : Il utilise **multiprocessing.Pool** pour paralléliser le calcul des lignes de l'ensemble de Mandelbrot avec différents nombres de processus. Mesurez le temps d'exécution pour chaque configuration. Enfin, calculez l'accélération.

En implémentant le code précédent nous avons pu connaître ses temps d'exécution en fonction du nombre de processeurs qui ont été utilisés pour réaliser la tâche, et donc sa parallélisation. De plus, le SpeedUp pouvait également être vu. Les résultats sont présentés dans le tableau suivant et le comportement de SpeedUp dans la figure.

N Processus	T. du Calcul (s)	Speeduo (S(n))
1	3.773	1.0
2	2.595	1.454
3	2.339	1.613
4	2.422	1.558
5	2.878	1.311
6	2.650	1.034
7	2.475	1.086
8	3.645	1.035

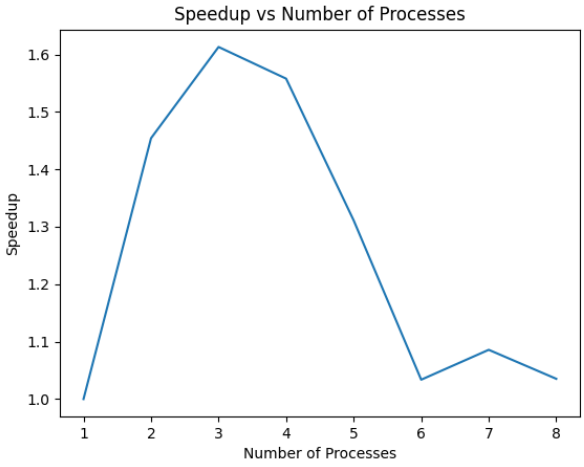

Fig. 1. Speedup vs Number of Processes

Enfin, l'image obtenue à la fin du processus est présentée, elle représente l'image de l'ensemble de Mandelbrot obtenu en utilisant le nombre maximum de processeurs spécifié dans la machine (jusqu'à 8 processeurs). Il ne s'agit pas seulement de l'itération avec 8 processeurs, mais plutôt de la combinaison des résultats de différentes itérations réalisées avec différents nombres de processeurs, de 1 à 8.

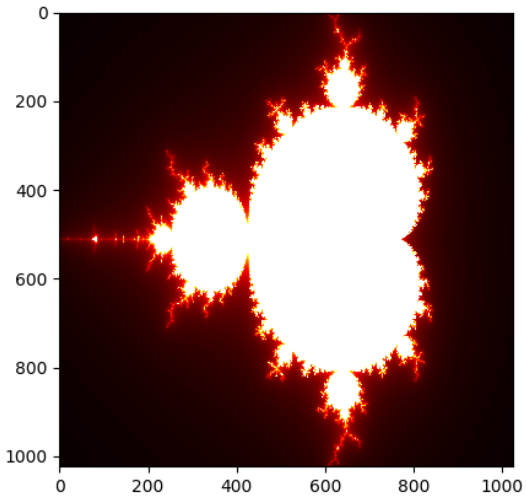

Fig. 2. Speedup vs Number of Processes

Avec 2 processus, l'accélération est d'environ 1,454, ce qui signifie que l'utilisation de deux processus accélère le calcul d'environ 1,454 fois par rapport à une exécution séquentielle. L'accélération atteint son point maximum avec 3 processus. Cependant, l'accélération diminue légèrement avec l'ajout de processus supplémentaires, ce qui indique que l'efficacité parallèle peut diminuer au-delà d'un certain point, car les coûts de communication et la répartition des tâches peuvent affecter les performances globales. Dans certains cas, l'ajout de processus ne conduit pas toujours à une accélération optimale et des ajustements peuvent être nécessaires pour optimiser les performances parallèles.

B. Mettre en oeuvre une stratégie maître–esclave pour distribuer les différentes lignes de l'image à calculer. Calculer le speedup avec cette nouvelle approche.

Le code suivant implémente un calcul parallèle de l'ensemble de Mandelbrot à l'aide de processus esclaves. Chaque processus esclave calcule en parallèle une ligne de l'image de Mandelbrot, et les résultats sont collectés et triés pour former la matrice de convergence finale. La bibliothèque **multiprocessing** et une file d'attente partagée sont utilisées pour coordonner la communication entre les processus esclaves et le processus maître.

```
def worker_process(args):
    queue, y = args
    row = calculate_row(y)
    queue.put((y, row))

if __name__ == '__main__':
    times = []
    for num_processes in range(1, cpu_count() + 1):
        deb = time()
        with Manager() as manager:
            queue = manager.Queue()
            with Pool(processes=num_processes) as
    pool:
                pool.map(worker_process,
                         [(queue, y) for y in range(
    height)])
            results = [queue.get() for _ in range(
    height)]
            results.sort()  # ensure the results are
     sorted by y
            convergence = np.array([row for y, row
    in results])
```

Remarque : Pour revoir le code complet, rendez-vous sur le dépôt GitHub dans les annexes.

Fonction worker˙process : cette fonction est la tâche principale que chaque processus de travail effectue en parallèle. Prend un tuple args comme argument et calcule la ligne correspondant à la valeur y à l'aide de la fonction **calculateRow(y)**.

Bloc principal (main) : Manager() est utilisé pour créer un objet gestionnaire qui fournit une file d'attente partagée (Queue) entre les processus. Ensuite, la file d'attente est créée à l'aide de **manager.Queue()**. **Pool(processes=numProcesses)** est utilisé pour créer un pool de processus parallèles. Chaque processus de travail exécutera la fonction **workerProcess** avec son tuple d'arguments respectif. Une fois tous les processus de travail terminés, les résultats de la file d'attente partagée sont collectés. Les résultats sont triés pour garantir qu'ils sont triés par la coordonnée y.

En implémentant le code précédent nous avons pu connaître ses temps d'exécution en fonction du nombre de processeurs qui ont été utilisés pour réaliser la tâche, et donc sa parallélisation. De plus, le SpeedUp pouvait également être vu. Les résultats sont présentés dans le tableau suivant et le comportement de SpeedUp dans la figure.

N Processus	T. du Calcul (s)	Speeduo (S(n))
1	5.382	1.0
2	3.984	1.431
3	3.496	1.589
4	3.545	1.468
5	3.775	1.470
6	4.347	1.338
7	4.211	1.218
8	4.486	1.072

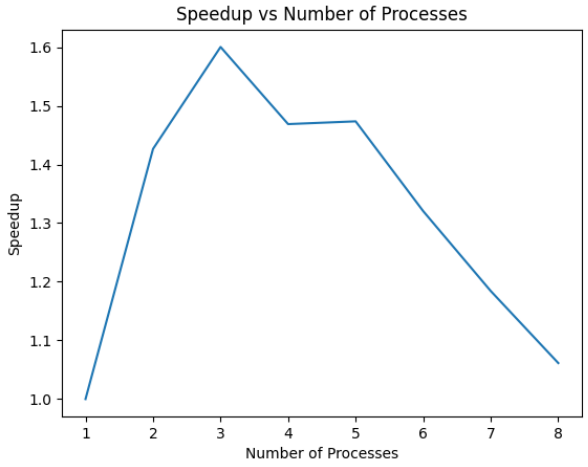

Fig. 3. Speedup vs Number of Processes

Les résultats montrent le temps de calcul et l'accélération pour différents nombres de processus dans un calcul parallèle. À mesure que le nombre de processus augmente, le temps de calcul diminue, indiquant une amélioration des performances. Cependant, l'accélération ne suit pas une tendance linéaire, ce qui montre que l'efficacité du parallélisme peut être affectée à mesure que le nombre de processus augmente. Ce phénomène, connu sous le nom de loi d'Amdahl, suggère qu'il existe une partie séquentielle dans le calcul qui limite l'efficacité du parallélisme. Dans ce cas, les avantages du parallélisme diminuent à mesure que davantage de processus sont ajoutés.

IV. Question 1.4

*A. Paralléliser le code séquentiel **matvec.cpp** en veillant à ce que chaque tâche n'assemble que la partie de la matrice utile à **sa somme** partielle du produit matrice-vecteur.*

Le code suivant partitionne la matrice par blocs de colonnes, effectue la multiplication en parallèle et assemble correctement le résultat final pour répondre aux exigences spécifiées.

```
dim = 120
A = np.array([[(i + j) % dim + 1. for i in
               range(dim)] for j in range(dim)])
u = np.array([i + 1. for i in range(dim)])
num_tasks = cpu_count()
Nloc = dim // num_tasks

def matvec_subset(cols):
    return A[:, cols].dot(u[cols])

def main():
    with Pool(num_tasks) as p:
        subsets = [range(i * Nloc, (i + 1) * Nloc)
                   for i in range(num_tasks)]
        results = p.map(matvec_subset, subsets)
    v = np.sum(results, axis=0)
```

Remarque : Pour revoir le code complet, rendez-vous sur le dépôt GitHub dans les annexes.

Partitionnement des blocs de colonnes: calcule le nombre de tâches en fonction du nombre de cœurs de processeur. De plus, il calcule la taille des sous-matrices locales (**Nloc**) pour répartir équitablement le travail entre les tâches.

Fonction matvecSubset : La fonction prend un ensemble de colonnes et effectue la multiplication de la sous-matrice de A par les colonnes correspondantes de u. Il garantit que chaque tâche ne traite que la partie de la matrice nécessaire à sa somme partielle du produit **matrice vector**.

Parallélisation avec pool et affectation de tâches: utilisez un groupe de processus (**pool**) pour répartir le travail entre les tâches. La fonction **matvecSubset** est appliquée en parallèle à chaque ensemble de colonnes à l'aide de **p.map()**. Les résultats sont stockés dans la liste des résultats.

*B. Paralléliser le code séquentiel **matvec.cpp** en veillant à ce que chaque tâche n'assemble que la partie de la matrice utile à **son produit** partielle du produit matrice-vecteur.*

Cette fois, puisque les lignes sont utilisées à la place des colonnes, outre le fait que le produit sera utilisé à la place de la somme, les modifications suivantes sont apportées par rapport au code original.

```
def matvec_subset(rows):
    return A[rows, :].dot(u)

v = np.concatenate(results)
```

Remarque : Pour revoir le code complet, rendez-vous sur le dépôt GitHub dans les annexes.

V. Annexes

github.com.