TD2 - OS202

PROGRAMMATION PARALLÈLE

André COSTA WERNECK
Leonardo GRECO PICOLI

2.1 Question du cours

Tout d'abord, il faut clarifier quand se passe un interblocage. Une situation d'interblocage arrive lorsque chaque processus d'un ensemble de processus est en attente d'un événement qui ne peut être causé que par un autre processus du même ensemble. Comme tous les processus de cet ensemble sont en attente, aucun ne pourra s'exécuter et générer les événements nécessaires à leurs activations. Ils attendront tous indéfiniment et on dit, donc, que les threads ou processus sont bloqués.

Dans le cadre de l'exercice donné dans le cours, on a 3 processus ou threads, le processus 1 (rank0), le 2 (rank1) et le 3 (rank2). Le code de l'exercice est le suivant :

```
MPI_Comm_rank(comm, &myRank ) ;
if (myRank == 0 ) {
    MPI_Ssend( sendbuf1, count, MPI_INT, 2, tag, comm);
    MPI_Recv( recvbuf1, count, MPI_INT, 2, tag, comm, &status);
} else if ( myRank == 1 ) {
    MPI_Ssend( sendbuf2, count, MPI_INT, 2, tag, comm);
  else if ( myRank == 2 ) {
    MPI_Recv( recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status );
    MPI_Ssend( sendbuf2, count, MPI_INT, 0, tag, comm);
    MPI_Recv( recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status );
}
```

Question: Un premier scénario où il n'y a pas d'interblocage

Donc, un scénario où il n'y a pas d'interblocage, ce lorsque le message du processus 0 arrive au processus 2 avant que celui du processus 1. Si cela se produit, le processus 2 recevra le message du processus 0, lui enverra un autre message et attendra que le message du processus 1 arrive, ce qui fera que le programme fonctionnera correctement, puisque le processus 1 avait déjà envoyé un message au processus 2, mais qu'il n'était pas encore arrivé.

Question: Un premier scénario où il y a interblocage

En revanche, si le processus 1 arrive avant le processus 0, le processus 2 recevra le message, mais enverra un message au processus 0. Le processus 0, à son tour, est toujours en train d'envoyer un message et ne peut pas recevoir de message du processus 2. Cela crée un interblocage, puisque les deux processus envoient des messages, mais ne s'attendent pas à recevoir quelque chose en retour, et sont donc bloqués.

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

Si on considère juste les 2 cas en dessus, la probabilité est donc de 50% d'avoir un interblocage.

2.2 Question du cours nº 2

Question: Alice a parallélisé en partie un code sur machine à mémoire distribuée. Pour un jeu de données spécifiques, elle remarque que la partie qu'elle exécute en parallèle représente en temps de traitement 90% du temps d'exécution du programme en séquentiel. En utilisant la loi d'Amdhal, pouvez-vous prédire l'accélération maximale que pourra obtenir Alice avec son code (en considérant n ≫ 1) ?

En utilisant la loi d'Amdhal vu en cours, on a:

- Let t_s be the time necessary to run the code in sequential
- Let f be the fraction of t_s, relative to the part of the code which can't be parallelized

So, the best expected speedup is :

$$S(n) = \frac{t_s}{f.t_s + \frac{(1-f)t_s}{n}} = \frac{n}{1+(n-1)f} \xrightarrow[n\to\infty]{} \frac{1}{f}$$

This law is useful to find a reasonable number of computing cores to use for an application.

Dans ce cas, f c'est la partie qui ne peut pas être parallélisé, donc

$$f = 100\% - 90\%$$

$$f = 10\%$$

et quand $n >> 1$ on a:

$$S(n) \to 1/f$$

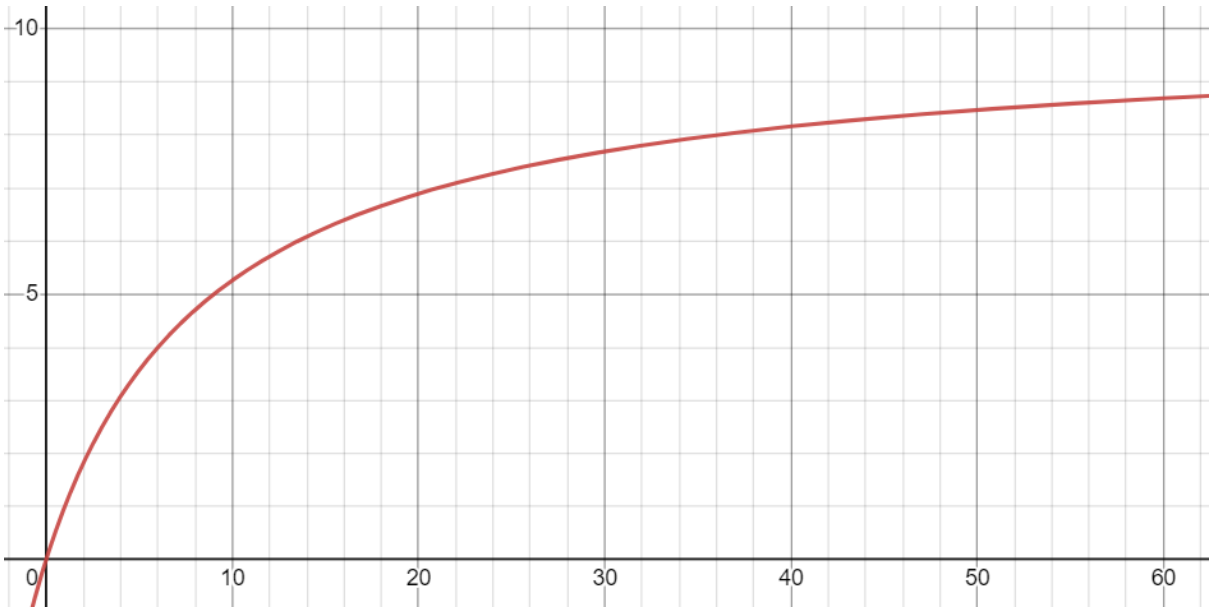
$$S(n) = 1/0.1 = 10$$

Donc l'accélération maximale que pourra obtenir Alice est égale à 10.

Question: À votre avis, pour ce jeu de données spécifique, quel nombre de nœuds de calcul semble-t-il raisonnable de prendre pour ne pas trop gaspiller de ressources CPU ?

En traçant un graphique de l'accélération maximale en fonction du nombre de noeuds, on obtient le graphique ci-dessous:

$$y = \frac{1}{0.1 + \frac{0.9}{x}}$$



On peut voir qu'après un nombre de nœuds égal à 10, l'inclinaison de la courbe devient de plus en plus petite. En plus c'est rare d'avoir un ordinateur avec plus que 16 nœuds de CPU disponibles. Dans ce cas là, un choix de 9 nœuds suffit bien les besoins. 9 nœuds donnerait une accélération de 5, la moitié de l'accélération maximale estimée.

Question: En effectuant son calcul sur son calculateur, Alice s'aperçoit qu'elle obtient une accélération maximale de quatre en augmentant le nombre de nœuds de calcul pour son jeu spécifique de données.
En doublant la quantité de données à traiter, et en supposant la complexité de l'algorithme parallèle linéaire, quelle accélération maximale peut espérer Alice en utilisant la loi de Gustafson ?

Si Alice a obtenu une accélération maximale de 4 après avoir fait les calculs, ça veut dire que:

$$y = \frac{1}{0.1 + \frac{0.9}{x}}$$

En mettant $y = 4$ dans l'équation, on obtient $x = 6\,n\text{œ}uds$.
Ça veut dire que Alice a 6 nœuds disponibles dans son ordinateur.

D'après la loi de Gustafson, on a:
$S = n - (n - 1) \cdot (1 - f)$

où:

$f = 0.9,\ n = 6\,noeuds$

Donc, l'accélération maximale en utilisant la Loi de Gustafson est égal à:

$$S = 6 - (6 - 1) \cdot (1 - 0.9) = 5.5$$

2.3 Ensemble de Mandelbrot

Pour paralléliser l'exécution de l'ensemble de Mandelbrot, la boucle for pour le calcul de la convergence a été divisée en parties également proportionnelles au nombre de threads/processus, comme dans l'extrait de code ci-dessous :

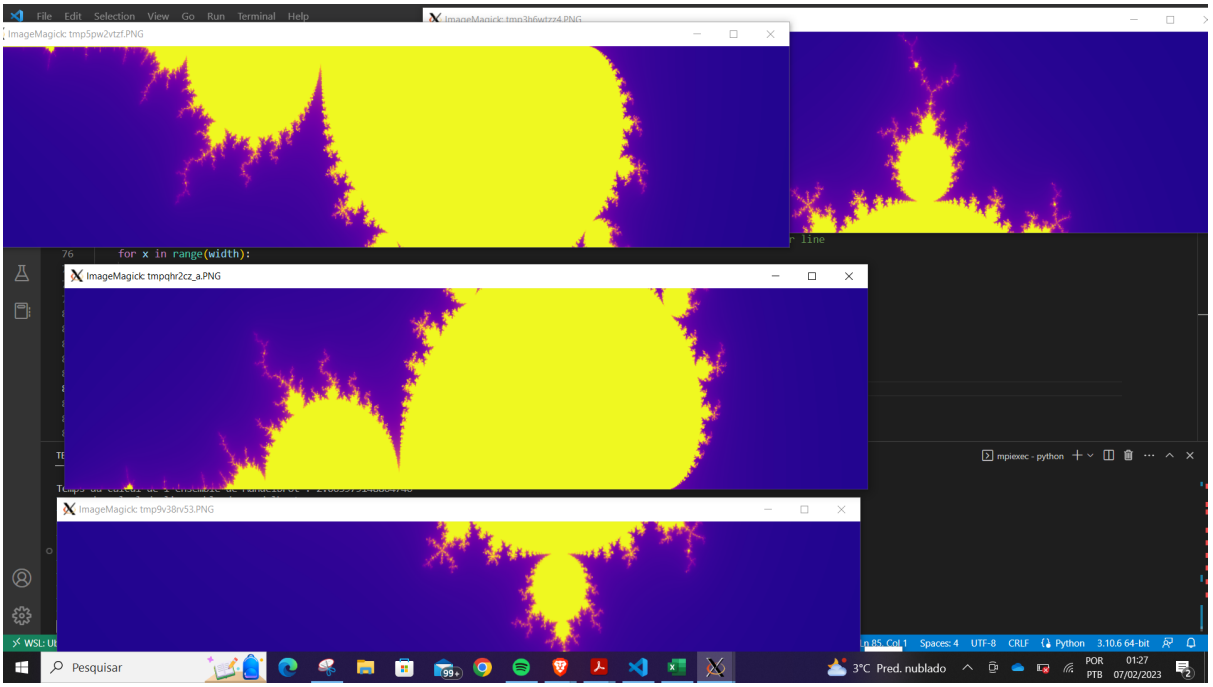
```
scat_data = globCom.scatter(scat_data,root=0)

# Calcul de l'ensemble de mandelbrot :
deb = time()
for y in range(rank*int(height/nbp),(rank+1)*int(height/nbp)): #first itering at the columns -> so, itering line per line
    for x in range(width):
        c = complex(-2. + scaleX*x, -1.125 + scaleY * y)
        scat_data[x,y - (rank*int(height/nbp))] = mandelbrot_set.convergence(c,smooth=True)

fin = time()
print(f"Temps du calcul de l'ensemble de Mandelbrot : {fin-deb}")

image = Image.fromarray(np.uint8(matplotlib.cm.plasma(scat_data.T)*255))
image.show()
```

En outre, pour mieux visualiser ce qui se passe, les images de chaque processus ont été imprimées.



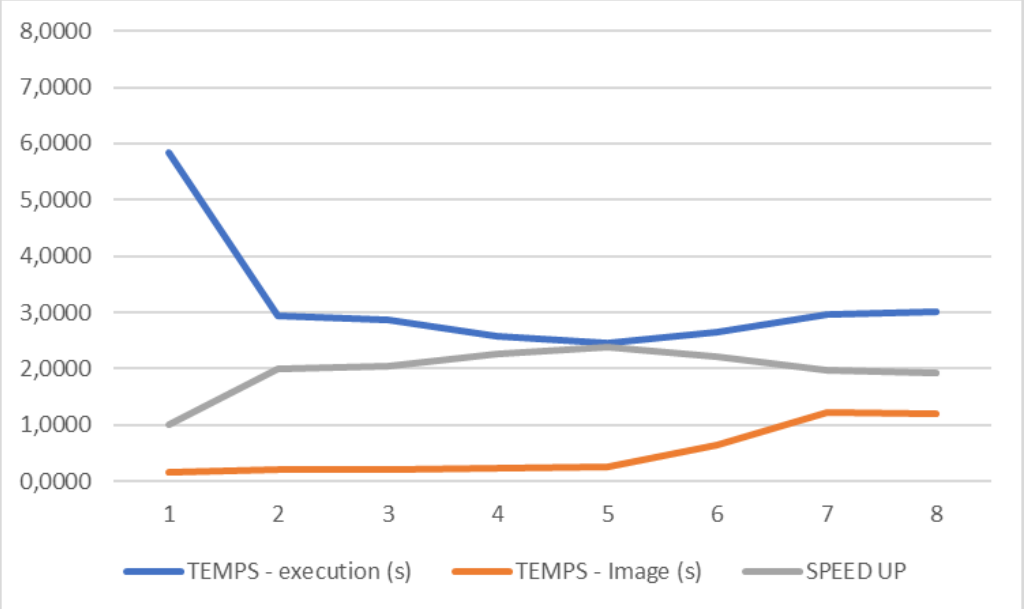
Ensuite, chaque partie calculée dans chaque thread a été jointe, grâce à la fonction gather de la MPI. De cette façon, l'image finale a été générée avec succès.

Pour le calcul du Speed Up j'ai utilisé la formule en dessus:

$$S(n) = \frac{t_s}{t_p(n)}$$

Les résultats sont les suivantes:

n	TEMPS - execution (s)	TEMPS - Image (s)	SPEED UP
1	5,8498	0,1715	1
2	2,9343	0,2101	1,9936
3	2,8698	0,2193	2,0384
4	2,5768	0,2429	2,2702
5	2,4591	0,2497	2,3788
6	2,6449	0,6433	2,2117
8	2,9629	1,2328	1,9743
16	3,0195	1,1948	1,9373



D'après le tableau et le graphique, les effets de la parallélisation deviennent très évidents. Jusqu'à n=4, on observe une augmentation de la vitesse et une diminution considérable du temps d'exécution. Après cela, le temps présente une légère augmentation, ce qui s'explique par le nombre de couleurs existantes dans la machine dans laquelle le code a été exécuté, une fois qu'elle n'a plus que 4 couleurs et, après n=4, la distribution des tâches n'est plus optimale, exactement pour la raison expliquée précédemment.

Le complet code est dans le github (fichier mandelbrot.py).

2.4 Produit matrice–vecteur

Dans le github (fichier matvec.cpp)