Travaux dirigés n°2

Nicolas RINCON VIJA

nicolas.rincon@ensta.fr

École Nationale Supérieure de Techniques Avancées - ENSTA
Systèmes parallèles et distribués - OS02
Palaiseau, France

Année académique 2024-2025

Table des matières

1 Parallélisation ensemble de Mandelbrot **3**
1.1 Travail demandé 3
1.1.1 Question 1 : 3
1.1.2 Question 2 : 4
1.1.3 Question 3 : 4

2 Produit matrice-vecteur **5**
2.1 Produit paralléle matrice-vecteur par colonne 5
2.1.1 Développement 5
2.2 Produit paralléle matrice-vecteur par ligne 6
2.2.1 Développement 6

Remarque préalable

Ce travail a été réalisé sur la base des résultats obtenus à partir des exécutions disponibles sur le dépôt GitHub suivant : https://github.com/Nrinconv/OS02_ENSTA.

1 Parallélisation ensemble de Mandelbrot

L'ensemble de Mandelbrot est un ensemble fractal inventé par Benoit Mandelbrot permettant d'étudier la convergence ou la rapidité de divergence dans le plan complexe de la suite récursive suivante :

$$\begin{cases} c, \text{ valeurs complexes données} \\ z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases} \tag{1}$$

dépendant du paramétre c.

Il est facile de montrer que s'il existe un N tel que $\mid z_N \mid > 2$, alors la suite z_n diverge. Cette propriété est très utile pour arrêter le calcul de la suite puisqu'on aura détecté que la suite a divergé. La rapidité de divergence est le plus petit N trouvé pour la suite tel que $\mid z_N \mid > 2$.

On fixe un nombre d'itérations maximal $N_{\max}$. Si jusqu'à cette itération, aucune valeur de z_N ne dépasse en module 2, on considère que la suite converge.

L'ensemble de Mandelbrot sur le plan complexe est l'ensemble des valeurs de c pour lesquelles la suite converge.

Pour l'affichage de cette suite, on calcule une image de $W \times H$ pixels telle qu'à chaque pixel (p_i, p_j), de l'espace image, on associe une valeur complexe

$$c = x_{\min} + p_i \frac{x_{\max} - x_{\min}}{W} + i \left(y_{\min} + p_j \frac{y_{\max} - y_{\min}}{H} \right).$$

Pour chacune des valeurs c associées à chaque pixel, on teste si la suite converge ou diverge.

— Si la suite converge, on affiche le pixel correspondant en noir.

— Si la suite diverge, on affiche le pixel avec une couleur correspondant à la rapidité de divergence.

1.1 Travail demandé

1.1.1 Question 1 :

À partir du code séquentiel `mandelbrot.py`, faire une partition équitable par bloc suivant les lignes de l'image pour distribuer le calcul sur `nbp` processus, puis rassembler l'image sur le processus zéro pour la sauvegarder. Calculer le temps d'exécution pour différents nombres de tâches et calculer le speedup. Comment interpréter les résultats obtenus ?

Nombre de Processus (nbp)	Temps d'exécution (s)	Speedup $S(n)$
1	3.9903	1.0000
4	1.4278	2.7963
8	1.4921	2.6762
16	2.0714	1.9260

TABLE 1 – Temps d'exécution et speedup pour différents nombres de processus

Résultats Les résultats de la table 1 montrent une accélération notable jusqu'à $nbp = 4$, où le speedup atteint 2.7963, mais au-delà, l'efficacité diminue. Pour $nbp = 8$, le gain est marginal, et pour $nbp = 16$, le temps d'exécution augmente, indiquant une surcharge due aux échanges de données et à la gestion des

threads. Cette tendance est cohérente avec l'architecture du processeur utilisé qui possède 4 cœurs physiques et 8 threads. L'hyper-threading ne semble pas apporter de bénéfice significatif, et l'ajout excessif de processus entraîne une contention des ressources et une perte d'efficacité. Ainsi, un nombre optimal de processus pour cette configuration semble être autour de 4.

1.1.2 Question 2 :

Réfléchissez à une meilleure répartition statique des lignes au vu de l'ensemble obtenu sur notre exemple et mettez-la en œuvre. Calculer le temps d'exécution pour différents nombres de tâches et calculer le speedup et comparez avec l'ancienne répartition. Quel problème pourrait se poser avec une telle stratégie ?

Cette nouvelle approche utilise une répartition statique intercalée des lignes pour améliorer l'équilibrage de charge. Contrairement à la version précédente où chaque processus recevait un bloc contigu, les lignes sont ici distribuées de manière alternée afin de réduire les déséquilibres et d'assurer une utilisation plus homogène des ressources.

Nombre de Processus (nbp)	Temps d'exécution (s)	Speedup $S(n)$
1	3.7949	1.0000
4	1.2330	3.0781
8	1.4688	2.5831
16	1.6263	2.3342

TABLE 2 – Temps d'exécution et speedup avec la répartition intercalée

Résultats Les résultats montrent une amélioration significative par rapport à la répartition en blocs de lignes, surtout pour $p = 4$, où on arrive à un speedup de $3,0781$, qui est supérieur à celui de $2,7963$ obtenus précédemment. Cela confirme que la répartition intercalée permet un meilleur équilibrage de la charge. Toutefois, pour $nbp = 8$ et $nbp = 16$, le gain de performance reste limité et montre une tendance à la saturation du speedup. L'augmentation du nombre de processus au-delà du nombre de cœurs physiques (4) n'apporte pas d'amélioration notable en raison de la surcharge de communication et des accès mémoire moins efficaces.

Un problème potentiel de cette stratégie est l'augmentation des échanges de données entre les processus, ce qui peut réduire les bénéfices du parallélisme. De plus, la fragmentation des lignes peut ompromettre la localité mémoire et entraîner un gaspillage d'efficacité dans la gestion du cache. Cependant, cela reste une solution plus efficace que la répartition en blocs, en particulier lorsque le nombre de processus est limité.

1.1.3 Question 3 :

Mettre en œuvre une stratégie maître-esclave pour distribuer les différentes lignes de l'image à calculer. Calculer le speedup avec cette approche et comparez avec les solutions différentes. Qu'en concluez-vous ?

Dans cette partie, une stratégie maître-esclave a été mise en œuvre afin de distribuer dynamiquement les lignes de l'image entre les processus. Contrairement aux approches statiques où les lignes étaient prédéfinies pour chaque processus, ici, le maître assigne les tâches au fur et à mesure en fonction de la disponibilité des esclaves, assurant ainsi un meilleur équilibrage de charge.

Résultats Les résultats montrent une amélioration de l'exécution jusqu'à $nbp = 8$, où le speedup atteint 2.3577, ce qui est légèrement supérieur aux approches statiques précédentes. Toutefois, à $nbp = 16$, le speedup diminue à 1.5546, indiquant que l'overhead de communication devient dominant. Par rapport à la répartition en blocs et la répartition intercalée, cette approche permet un meilleur équilibrage des charges, mais elle souffre également d'une surcharge due aux échanges fréquents entre le maître et les esclaves.

Nombre de Processus (nbp)	Temps d'exécution (s)	Speedup $S(n)$
1	3.4640	1.0000
4	1.5732	2.2018
8	1.4695	2.3577
16	2.2283	1.5546

TABLE 3 – Temps d'exécution et speedup avec la stratégie maître-esclave

Comparé aux approches statiques, celui-ci est plus performant si le nombre de tâches est moins, mais il ne monte pas bien quand les tailles de lot sont plus grands en raison de la synchronisation et la gestion des files d'attente des lots.

2 Produit matrice-vecteur

On considère le produit d'une matrice carrée A de dimension N par un vecteur u de même dimension dans $\mathbb{R}$. La matrice est constituée des coefficients définis par :

$$A_{ij} = (i + j) \mod N.$$

Par souci de simplification, on supposera N divisible par le nombre de tâches `nbp` exécutées.

2.1 Produit paralléle matrice-vecteur par colonne

— Calculer en fonction du nombre de tâches la valeur de N_{loc}.

— Paralléliser le code séquentiel `matvec.py` en veillant à ce que chaque tâche n'assemble que la partie de la matrice utile à sa somme partielle du produit matrice-vecteur. On s'assurera que toutes les tâches à la fin du programme contiennent le vecteur résultat complet.

— Calculer le speedup obtenu avec une telle approche.

2.1.1 Développement

Pour paralléliser le produit matrice-vecteur, la matrice A de dimension $N \times N$ a été découpée en blocs de colonnes de taille $N_{\text{loc}} = \frac{N}{\text{nbp}}$, chaque processus recevant N_{loc} colonnes. Cette approche assure une distribution équilibrée des calculs entre les processus. Chaque tâche calcule le produit de sa sous-matrice locale avec le vecteur u, puis utilise une réduction MPI pour assembler le vecteur résultat global.

Nombre de Processus (nbp)	Temps d'exécution (s)	Speedup $S(n)$
1	0.0002	1.0000
4	0.0047	0.0426
8	0.0716	0.0028
16	0.0179	0.0112

TABLE 4 – Temps d'exécution et speedup pour le produit matrice-vecteur parallélisé par colonnes

Résultats Les résultats montrent une diminution de performance lorsque le nombre de processus augmente. En effet, contrairement à ce qui était prévu, le speedup diminue alors qu'on attendait qu'il augmente, ce qui témoigne d'une surcharge MPI et d'un coût de communication excessif. Les résultats non satisfaisants pour $nbp = 8$ et $nbp = 16$ sont à mettre en rapport avec soit un déséquilibre dans la répartition du calcul, soit avec une latence MPI trop élevée. Dans la situation actuelle, cette méthode de parallélisation ne semble pas

adaptée, le temps de ça calcul initial étant suffisamment faible pour que le coût induit pour distribuer des tâches soit plus élevé que les gains obtenus.

2.2 Produit paralléle matrice-vecteur par ligne

— Calculer en fonction du nombre de tâches la valeur de N_{loc}.

— Paralléliser le code séquentiel `matvec.py` en veillant à ce que chaque tâche n'assemble que la partie de la matrice utile à son produit matrice-vecteur partiel. On s'assurera que toutes les tâches à la fin du programme contiennent le vecteur résultat complet.

— Calculer le speedup obtenu avec une telle approche.

2.2.1 Développement

Dans cette approche, la parallélisation du produit matrice-vecteur a été réalisée en découpant la matrice A en blocs de lignes de taille $N_{\text{loc}} = \frac{N}{\text{nbp}}$. Chaque processus a ainsi reçu un ensemble de lignes et a effectué le calcul partiel de $v_{\text{loc}} = A_{\text{loc}} \cdot u$. Enfin, une opération de collecte avec 'MPI.Gatherv' a permis d'assembler le vecteur résultat complet.

Nombre de Processus (nbp)	Temps d'exécution (s)	Speedup $S(n)$
1	0.0002	1.0000
2	0.0007	0.2857
4	0.0138	0.0145
8	0.0077	0.0260
16	0.0711	0.0028

TABLE 5 – Temps d'exécution et speedup pour le produit matrice-vecteur parallélisé par lignes

Résultats Les résultats obtenus indiquent que l'efficacité de la parallélisation est très faible car pour toutes les valeurs de nbp, le speedup est inférieur à 1. Le fait d'augmenter le nombre de processus ne semble pas contribuer à améliorer les performances, et même pour $nbp = 16$, on observe une défaillance significative du temps d'exécution. Cela pointe vers une surcharge de communication très importante ainsi qu'un défaut d'équilibrage de la charge de travail entre les processus.