TD n° 2 -Nicolas Lopez Nieto

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 :

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

dépendant du paramètre 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 lesquels 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 :

$$c = x_{min} + p_i \frac{x_{\max} - x_{\min}}{W} + i \left(y_{\min} + p_j \frac{y_{\max} - y_{\min}}{H} \right) \tag{2}$$

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.

Exercices

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 ?

Valuer	**Processeur 1**	**Processeur 2**	**Processeur 3**	**Processeur 4**
Temps ens de Mand	5.13415	5.27955	4.99814	5.18765
Temps consti de l'img	0.07053	0.07053	0.07053	0.07053
Speed up	—	1.04	2.86	1.49

TABLE 1 – Table de temps

L'analyse des résultats montre que il existe une variation dans les temps d'exécution individuels. Les différences de temps entre les processeurs (de 4.99814 à 5.27955 secondes) indiquent un léger déséquilibre dans la répartition des tâches, ce qui peut être dû à une distribution inégale des lignes de l'image ou à des variations dans les performances des nœuds de calcul.

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 ?

   La stratégie statique a été développée en divisant l'image en blocs de lignes, car chaque ligne correspond à une plage continue de valeurs complexes, ce qui permet une meilleure localité des données et réduit la quantité d'échanges nécessaires entre les processus MPI. Cette approche s'est révélée plus performante que la répartition par colonnes, car elle exploite mieux la mémoire cache. Mais, un problème de cette stratégie peut etre le déséquilibre de charge, puisque certaines lignes nécessitent plus de calculs en fonction de la structure de l'ensemble de Mandelbrot.

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 ?

Valuer	**Mandelbrot**	**Parallele**	**Static**	**Master-slave**
Temps ensemble de Mandelbrot	5.31186	5.09715	1.85852	3.56646
Temps constitution de l'image	0.14172	0.08539	0.13457	0.09426
Speed up	—	1.04	2.86	1.49

TABLE 2 – Table de temps

Selon les résultats de la table, la répartition statique obtient le

meilleur speed-up. Cela peut s'expliquer car bien que la charge de travail ne soit pas parfaitement équilibrée, la gestion simple des tâches réduit les coûts de communication entre les processus MPI. Mais, au contraire, la stratégie maître-esclave, bien qu'efficace pour éviter l'inactivité des processus, introduit une surcharge due aux communications fréquentes entre le maître et les esclaves, ce qui peut limiter l'accélération. La parallélisation naïve par colonnes est la moins performante car elle ne tient pas compte des variations de complexité computationnelle à travers l'image. Ces résultats montrent que la meilleure approche dépend d'un compromis entre équilibrage de charge et minimisation des communications inter-processus.

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 :

$$A_{ij} = (i + j) \mod N. \tag{3}$$

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

2.1 Produit parallèle matrice-vecteur par colonne

Afin de paralléliser le produit matrice–vecteur, on décide dans un premier temps de partitionner la matrice par un découpage par bloc de colonnes. Chaque tâche contiendra N_{loc} colonnes de la matrice.

- 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 speed-up obtenu avec une telle approche.

2.2 Produit parallèle matrice-vecteur par ligne

Afin de paralléliser le produit matrice–vecteur, on décide dans un deuxième temps de partitionner la matrice par un découpage par bloc de lignes. Chaque tâche contiendra N_{loc} lignes de la matrice.

- 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 speed-up obtenu avec une telle approche.

Valeur	MatVec	Columnes	Lines
Temps du calcul	0.000465	0.000418	0.000057
Speed up	———	1.1	8.16

Table 3 – Tabla de ejemplo con 4 columnas y 2 filas

3 Entraînement pour l'examen écrit

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'Amdahl, pouvez-vous prédire l'accélération maximale que pourra obtenir Alice avec son code (en considérant $n \gg 1$) ?

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

— 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 ?