Travaux Dirigés N°2 OS202

Jair VASQUEZ TORRES

Introduction

Ce rapport présente la parallélisation de calculs numériques intensifs à l'aide de la bibliothèque **mpi4py** sur une architecture à mémoire distribuée. L'enjeu est de transformer des algorithmes séquentiels en programmes performants capables de distribuer efficacement la charge de travail entre plusieurs cœurs.

Nous étudions deux cas d'application fondamentaux :

- **L'ensemble de Mandelbrot :** pour analyser l'équilibrage de charge face à un problème dont la complexité varie selon la zone du plan complexe.
- **Le produit Matrice-Vecteur :** pour évaluer l'impact des communications collectives (`Allreduce`, `Allgather`) sur les performances selon la décomposition choisie (lignes vs colonnes).

Enfin, une analyse théorique basée sur les lois d'Amdahl et de Gustafson permettra de définir les limites d'accélération de ces méthodes et les conditions optimales pour le passage à l'échelle.

1 Environnement de Test

Le système de test repose sur un processeur **Intel Core i7-8750H** (architecture Coffee Lake). Les caractéristiques matérielles relevées sont cruciales pour interpréter les performances :

- **Fréquence :** 2.20 GHz (Base) jusqu'à 4.10 GHz (Turbo).
- **Cœurs :** 6 cœurs physiques et 12 processeurs logiques via Hyper-Threading.
- **Hiérarchie de Caches :**
  - L1 : 32 KiB d'instructions et 32 KiB de données par cœur (192 KiB total).
  - L2 : 256 KiB par cœur (1.5 MiB total).
  - L3 : 9 MiB partagés.
- **Mémoire vive :** 16 GiB DDR4 à 2666 MHz.
- **Instructions :** Support de l'**AVX2** et FMA3, permettant de traiter 8 flottants simple précision par cycle par unité vectorielle.

2 Parallélisation ensemble de Mandelbrot

L'objectif de cette section est de paralléliser le calcul de l'ensemble de Mandelbrot en utilisant MPI. Nous explorons trois stratégies de distribution de charge pour optimiser le temps de calcul.

2.1 Distribution par Blocs (Statique)

Dans cette première approche, le domaine est divisé en n blocs consécutifs de lignes, où n est le nombre de processus.

TABLE 1 – Performance : Distribution par Blocs

n	Temps (s)	Speedup (S)	Efficacité (E)
1	2.741	1.000	100%
2	1.698	1.614	80.7%
4	1.193	2.298	57.4%

Analyse : On observe un déséquilibre de charge significatif. Par exemple, avec $n = 4$, le processus 1 termine en 0.975s tandis que le processus 0 nécessite 1.193s. Cela s'explique par le fait que les zones centrales de l'ensemble de Mandelbrot demandent beaucoup plus d'itérations que les zones périphériques. Le temps total est dicté par le processus le plus lent.

2.2 Distribution Cyclique (Statique)

Pour remédier au déséquilibre, nous avons implémenté une distribution cyclique : chaque processus i traite les lignes $i, i + n, i + 2n, \ldots$.

TABLE 2 – Performance : Distribution Cyclique

n	Temps (s)	Speedup (S)	Efficacité (E)
1	2.802	1.000	100%
2	1.594	1.758	87.9%
4	1.169	2.397	59.9%

Analyse : La distribution cyclique améliore l'équilibre. Avec $n = 4$, l'écart entre le processus le plus rapide (1.117s) et le plus lent (1.169s) est réduit à seulement 0.052s. Le Speedup est légèrement supérieur à la méthode par blocs, car les parties "lourdes" du calcul sont distribuées plus équitablement entre tous les cœurs.

2.3 Stratégie Maître-Esclave (Dynamique)

Ici, le processus 0 (Maître) distribue les lignes à la demande aux processus esclaves.

Analyse :

- **Surcoût du Maître :** Avec $n = 2$ (1 seul esclave), le temps augmente par rapport au séquentiel car le Maître ne calcule rien et la communication sature les ressources.

Table 3 – Performance : Maître-Esclave

n (Total)	Esclaves	Temps (s)	Speedup (S)
2	1	3.924	0.714
3	2	2.217	1.264
4	3	1.629	1.720
5	4	1.443	1.942
6	5	1.451	1.931

— **Équilibre dynamique :** Cette méthode offre le meilleur équilibre théorique. Cependant, on remarque une stagnation à $n = 6$. Cela est dû au fait que le gain de calcul est compensé par le coût de la communication MPI (latence des messages pour chaque ligne).

2.4 Comparaison et Synthèse

La distribution **cyclique** s'avère la plus efficace pour ce problème à taille fixe (1024×1024) car elle équilibre bien la charge sans le surcoût de gestion du Maître-Esclave. La stratégie **Maître-Esclave** serait plus avantageuse pour des calculs beaucoup plus complexes ou sur des architectures hétérogènes.

3 Produit matrice-vecteur

Dans cette section, nous étudions la parallélisation du produit $A \cdot u = v$ pour une matrice carrée de dimension $N = 120$. Cette taille, relativement petite, met en évidence l'impact de la latence de communication par rapport au temps de calcul.

3.1 Considérations techniques : Contrôle du multithreading

Pour obtenir des mesures de performance cohérentes et isoler le parallélisme de processus (MPI), il a été nécessaire de limiter le multithreading interne des bibliothèques d'algèbre linéaire (BLAS/MKL) utilisées par NumPy. Au début de chaque script, nous avons configuré les variables d'environnement suivantes :

— `OMP_NUM_THREADS=1`
— `MKL_NUM_THREADS=1`

Sans cette configuration, chaque processus MPI tenterait d'utiliser tous les cœurs disponibles, provoquant une *oversubscription* des ressources et dégradant les performances.

3.2 a - Produit parallèle par ligne

Dans cette approche, chaque processus possède un bloc de lignes de la matrice A et le vecteur complet u. Après le calcul local, un `Allgather` est utilisé pour reconstruire le vecteur résultat v sur tous les processus.

Table 4 – Performance : Produit par ligne

n	Temps (s)	Speedup (S)	Efficacité (E)
1	0.000571	1.000	100%
2	0.000048	11.895	594.7%
4	0.000113	5.053	126.3%

Analyse : On observe un **Speedup super-linéaire** pour $n = 2$. Cela s'explique par le fait qu'en divisant la matrice, les données locales de chaque processus s'insèrent plus efficacement dans la mémoire cache L1/L2 du processeur, réduisant ainsi drastiquement les accès à la RAM. Cependant, pour $n = 4$, le temps augmente car le coût de communication de `Allgather` commence à dominer le gain de calcul sur une matrice aussi petite.

3.3 b - Produit parallèle par colonne

Ici, chaque processus possède un bloc de colonnes de A et un segment du vecteur u. Le résultat est obtenu via une réduction globale (`Allreduce`) avec l'opération `MPI.SUM`.

Table 5 – Performance : Produit par colonne

n	Temps (s)	Speedup (S)	Efficacité (E)
1	0.000751	1.000	100%
2	0.025309	0.029	1.4%
4	0.002650	0.283	7.1%

Analyse : La performance est nettement inférieure à la méthode par lignes. Pour $n = 2$, on note une chute brutale de performance due au coût de l'`Allreduce`. Le volume de données à réduire (vecteur de taille 120) est disproportionné par rapport à la faible charge de calcul local (120×60 opérations).

3.4 Synthèse des résultats

Pour de petites dimensions ($N = 120$), la parallélisation est extrêmement sensible à l'**overhead** de communication. La méthode par **lignes** est ici préférable car la multiplication locale est plus directe et la synchronisation moins coûteuse que la réduction arithmétique globale requise par la méthode par colonnes.

4 Entraînement pour l'examen écrit

Cette section analyse les performances théoriques du code d'Alice en utilisant les lois fondamentales du calcul parallèle : Amdahl et Gustafson.

4.1 Loi d'Amdahl et Accélération Maximale

La loi d'Amdahl définit l'accélération maximale d'un programme en fonction de sa fraction séquentielle s. Alice indique que la partie parallélisable p représente 90% du temps d'exécution ($p = 0.9$). La fraction séquentielle est donc $s = 1 - p = 0.1$.

L'accélération S pour n nœuds est donnée par :

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Lorsque le nombre de nœuds tend vers l'infini ($n \gg 1$), l'accélération maximale devient :

$$S_{max} = \lim_{n\to\infty} \frac{1}{0.1 + \frac{0.9}{n}} = \frac{1}{0.1} = 10$$

Analyse : Alice ne pourra jamais obtenir une accélération supérieure à **10**, quelle que soit la puissance de calcul utilisée, car la partie séquentielle devient le goulot d'étranglement.

4.2 Nombre de nœuds raisonnable

Pour ne pas gaspiller de ressources, il faut analyser l'efficacité $E = S(n)/n$. En utilisant 10 nœuds, l'accélération serait $S(10) = 1/(0.1 + 0.09) \approx 5.26$, soit une efficacité de seulement 52%.

Un choix raisonnable se situe souvent au point où l'ajout de ressources n'apporte plus qu'un gain marginal. Ici, prendre entre **4 et 6 nœuds** semble judicieux. Avec $n = 4$, $S(4) \approx 3.07$ ($E = 76\%$), alors qu'avec $n = 10$, on double presque les ressources pour un gain minime.

4.3 Loi de Gustafson et Passage à l'Échelle

Alice remarque que l'accélération réelle est plafonnée à 4. En doublant la quantité de données, la loi de Gustafson s'applique. Contrairement à Amdahl, elle suppose que le temps de calcul parallèle reste fixe et que c'est la charge de travail qui augmente.

Si Alice obtient une accélération $S = 4$ avec n nœuds, on peut déduire la fraction séquentielle réelle $s_{réelle}$ (incluant les coûts de communication) par la formule de Gustafson :

$$S(n) = n + (1 - n)s$$

Cependant, la loi de Gustafson exprime l'accélération pour une charge de travail augmentée comme :

$$S_{gustafson}(n) = s + n(1 - s)$$

Si la complexité est linéaire et que l'on double les données, la part parallélisable p augmente proportionnellement. Si initialement $p = 0.9$ pour N données, pour $2N$ données, la nouvelle fraction séquentielle s' devient :

$$s' = \frac{s}{s + 2p} = \frac{0.1}{0.1 + 1.8} \approx 0.0526$$

L'accélération maximale espérée avec la loi de Gustafson pour n nœuds devient :

$$S_{espérée} = n - s'(n - 1)$$

Conclusion : En doublant les données, la part relative de la partie séquentielle diminue. Alice peut espérer une accélération qui tend vers **19** si elle dispose de suffisamment de nœuds ($n \to 20$), car la loi de Gustafson montre que le passage à l'échelle est bien plus efficace lorsque la taille du problème augmente avec la puissance de calcul.

5 Conclusion Générale

Ce travail dirigé nous a permis d'explorer les principes fondamentaux de la programmation parallèle avec MPI en Python. À travers les différentes expérimentations, nous avons dégagé plusieurs conclusions clés :

1. **L'importance de l'équilibrage de charge :** Le calcul de l'ensemble de Mandelbrot a illustré qu'une distribution statique par blocs est inefficace pour les problèmes hétérogènes. La stratégie *cyclique* s'est révélée être un excellent compromis entre simplicité et performance, tandis que l'approche *Maître-Esclave* offre la flexibilité maximale au prix d'une complexité de communication accrue.
2. **Le coût de la communication (Overhead) :** Les tests sur le produit matrice-vecteur ont montré que pour des problèmes de petite taille ($N = 120$), le coût de synchronisation et de transfert de données (via `Allgather` ou `Allreduce`) peut rapidement surpasser le gain de calcul. Cela confirme que la parallélisation n'est bénéfique que si la charge de travail par processus est suffisante.
3. **Gestion des ressources matérielles :** L'utilisation de variables d'environnement pour limiter le multithreading de NumPy a été cruciale. Elle nous a appris que, sur une machine moderne, le parallélisme de processus (MPI) peut entrer en conflit avec le parallélisme de threads (OpenMP/MKL), nécessitant un contrôle strict pour garantir l'efficacité.
4. **Lois de passage à l'échelle :** Enfin, l'étude théorique d'Alice nous rappelle que si la loi d'Amdahl limite nos ambitions à taille de problème fixe, la loi de Gustafson ouvre la voie au calcul haute performance : pour exploiter réellement de grandes architectures, il faut augmenter la taille des données traitées.

En conclusion, la réussite d'une parallélisation ne dépend pas seulement du nombre de cœurs utilisés, mais d'une adéquation fine entre la structure des données, la topologie des communications et les spécificités du matériel.

Ressources et Reproductibilité

L'intégralité du code et du Makefile est accessible sur : `https://github.com/JairVasquezT/OS202-2026.git`

Références

[1] L. Dalcin, *MPI for Python*, `https://mpi4py.readthedocs.io/`.

[2] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, 1967.

[3] J. L. Gustafson, *Reevaluating Amdahl's Law*, Communications of the ACM, 1988.