Master 1 ARIAS, Programmation parallèle

Parallélisation MPI d'un pipeline de traitement : Distances de Hamming, Floyd–Warshall par blocs et clustering PAM

Rabah TOUBAL
N° Étudiant : 22 40 70 21

1. Introduction

Ce projet met en œuvre une chaîne complète de traitement parallèle basée sur trois algorithmes utilisant MPI :

- le calcul distribué des distances de Hamming entre des séquences d'ARN pour construire un graphe pondéré au format DOT ;
- une version parallèle par blocs de l'algorithme Floyd–Warshall pour obtenir les plus courts chemins entre tous les sommets du graphe ;
- l'algorithme PAM (Partitioning Around Medoids), adapté en MPI pour effectuer un classement des séquences à partir de la matrice finale des distances.

L'ensemble forme un pipeline cohérent :

FASTA →Hamming → Graphe DOT → Floyd–Warshall parallèle → PAM parallèle

L'objectif du rapport est de présenter les principales idées de parallélisation utilisées, d'expliquer le fonctionnement général de chaque étape et de fournir une première analyse de performance.

Mes choix d'implémentation ont été guidés principalement par les explications du sujet, ainsi que par plusieurs ressources externes comme :

- cse.buffalo.edu/faculty/miller/Courses/CSE633/Asmita-Gautam-Spring-2019, qui donne une vision claire du découpage en blocs et des échanges MPI.
- Une vidéo explicative sur le fonctionnement de l'algorithme de Floyd-Warshall, qui m'a aidé à mieux visualiser la dynamique des mises à jour par couples de sommets.

2. Construction parallèle de la matrice de distances

La première étape du pipeline consiste à construire en parallèle la matrice des distances de Hamming entre toutes les séquences d'ARN.
Le rang 0 lit le fichier FASTA, vérifie que toutes les séquences ont la même longueur, puis diffuse à l'ensemble des processus :

- le nombre de séquences,
- la longueur des séquences,
- un tableau contigu contenant toutes les séquences.

Chaque processus possède alors toutes les données nécessaires pour effectuer ses calculs localement.
La parallélisation repose sur une répartition simple : on divise les indices de lignes entre les processus, et chacun calcule les distances pour les lignes dont il est responsable, en comparant ses séquences avec toutes les autres. Cette méthode équilibre bien la charge et ne nécessite aucune communication supplémentaire pendant le calcul.

Une fois les blocs de résultats produits, chaque processus envoie au rang 0 la portion de la matrice qu'il a calculée. Le rang 0 reconstruit alors la matrice n×n complète et génère un fichier DOT contenant uniquement les arêtes dont la distance est inférieure au seuil ε.

Cette matrice sert ensuite d'entrée à l'algorithme parallèle de Floyd–Warshall.

3. Parallélisation de l'algorithme de Floyd–Warshall

3.1 Principe général

L'algorithme de Floyd–Warshall applique la relation

$$D[i][j] = min(D[i][j], D[i][k] + D[k][j])$$

La version parallèle utilise :

- un découpage de la matrice en blocs,
- une grille de processus qui se partagent ces blocs,
- la diffusion du bloc pivot à chaque itération.

Cette approche est conforme à la méthode décrite dans le sujet du projet ainsi qu'au document de référence d'Asmita Gautam.

3.2 Distribution initiale

Chaque processus reçoit un ensemble de blocs de la matrice, déterminé par une fonction de répartition sur la grille de processus.
Chaque bloc est initialisé à partir de la matrice d'adjacence (poids des arêtes ou absence de chemin direct).

3.3 Mise à jour parallèle

Pour chaque étape (k) :

- **Bloc pivot (k,k)** : mis à jour localement par son propriétaire, puis diffusé à tous les processus.

- **Ligne k et colonne k** : les processus propriétaires mettent à jour leurs blocs de ligne/colonne, puis les diffusent également.
  Tous les processus reçoivent ces blocs, mais chacun n'utilise que ceux nécessaires pour ses propres blocs.

- **Blocs internes** : chaque processus met ensuite à jour les blocs qu'il détient, en combinant le bloc de colonne ((i,k)) et le bloc de ligne ((k,j)).

L'utilisation de diffusions non bloquantes permet de recouvrir une partie des communications par du calcul.

À la fin, le rang 0 rassemble tous les blocs pour reconstruire la matrice complète, qui servira d'entrée à l'algorithme PAM.

4. Parallélisation de l'algorithme PAM (clustering)

L'algorithme PAM consiste à choisir k médiods parmi les n points, puis à associer chaque élément au médiod le plus proche. L'évaluation d'une configuration de médiods demande de recalculer un coût global à partir de la matrice des distances.

4.1 Parallélisation du calcul du coût

Pour une liste de médiods donnée, le coût total est la somme, pour chaque point i, de la distance au médiod le plus proche.
La parallélisation suit l'idée suivante.
– chaque processus traite un sous-ensemble de sommets (une tranche de lignes de la matrice de distances),
– un appel à MPI_Allreduce (opération SUM) additionne les coûts partiels de chaque processus et fournit le coût global.

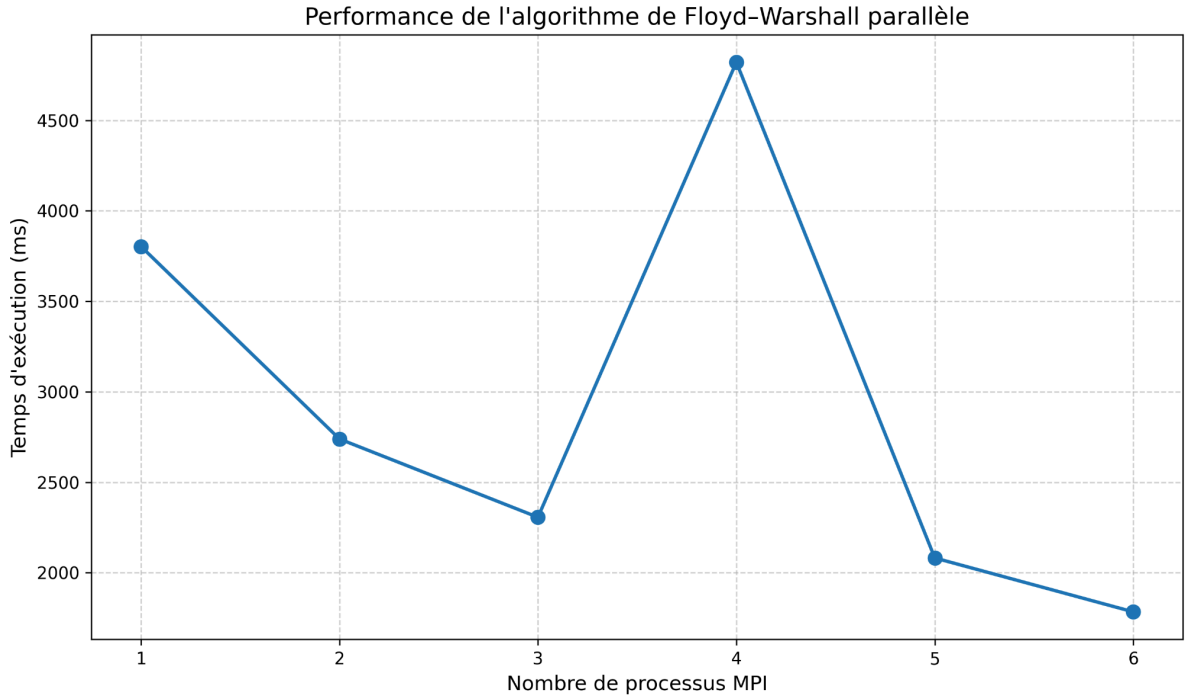
4.2 Recherche du meilleur échange

À chaque itération, l'algorithme teste toutes les paires *(m,h)* où *m* est l'index d'un médiod actuel et *h* un point non-médiod.

Pour chaque configuration candidate, le coût global est évalué en parallèle ; le rang 0 garde uniquement la meilleure amélioration trouvée pendant l'itération, met à jour la liste de médiods puis la diffuse aux autres processus.

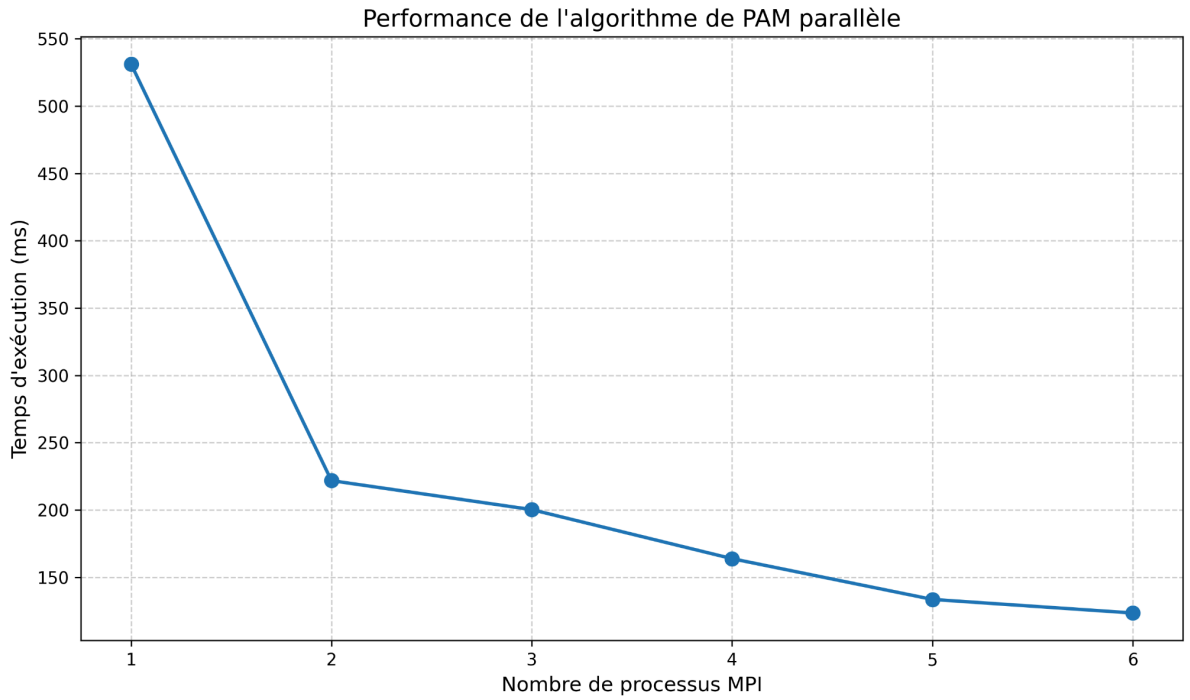
Ce mécanisme permet à PAM de converger progressivement vers un optimum local.

5. Analyse synthétique des performances

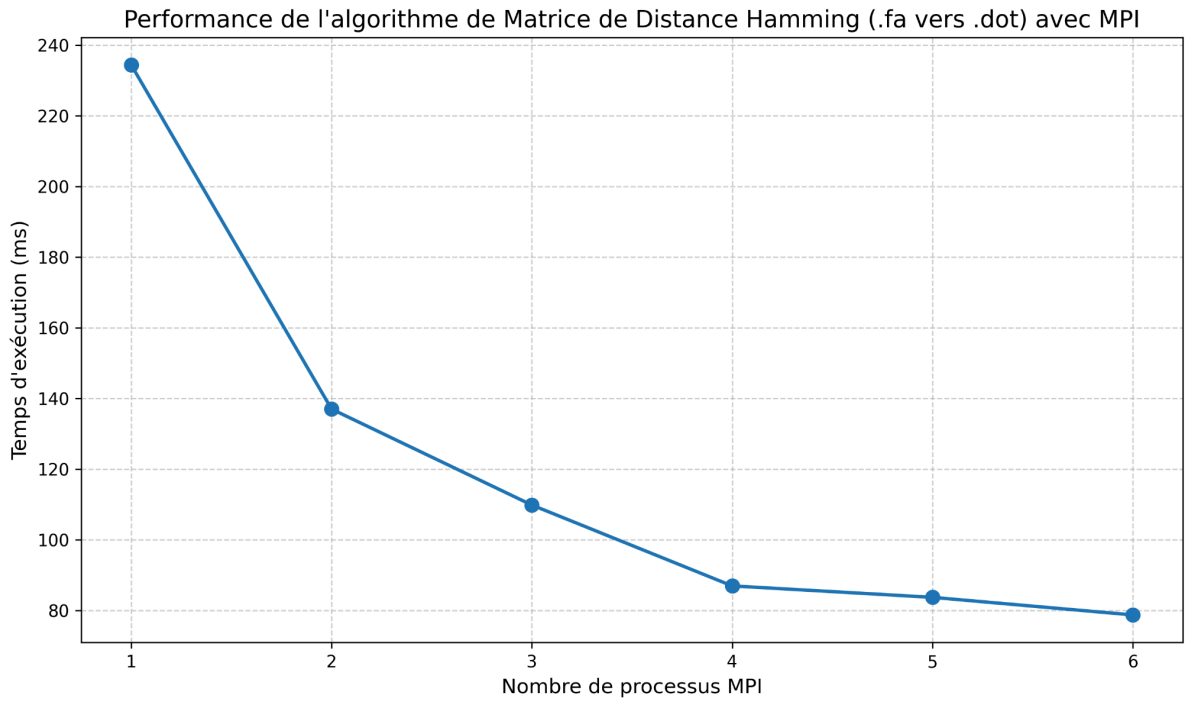


Les performances ont été évaluées sur le jeu de test contenant 2000 séquences d'ARN.
On observe une amélioration nette entre 1 et 3 processus, suivie d'une forte dégradation à 4 processus, puis une amélioration à nouveau pour 5 et 6 processus.

Ce comportement s'explique par le choix de la taille de bloc : avec p = 4, l'algorithme applique la formule du sujet ($b = \frac{n}{\sqrt{p}}$)*, produisant des blocs très grands (1000×1000) et un parallélisme très limité.*
À l'inverse, lorsque p n'est pas un carré parfait (p = 2, 3, 5, 6), l'algorithme bascule sur ma stratégie de bloc adaptative (256×256), mieux équilibrée et plus efficace.
Ce test montre donc que la taille des blocs influence plus les performances que le nombre de processus lui-même.



L'algorithme PAM parallèle est testé sur la matrice de distances produite par Floyd–Warshall, pour le jeu de 2000 séquences d'ARN, avec k = 4 classes.
Le temps d'exécution passe d'environ 530 ms (1 processus) à ≃ 120–130 ms (6 processus), ce qui montre une bonne accélération quand on augmente le nombre de processus.
La décroissance n'est pas parfaitement linéaire, mais on observe clairement que la distribution des lignes de la matrice entre les processus permet de réduire le coût global du calcul du PAM.



Pour cet algorithme, j'ai utilisé le même jeu de test que pour le reste du pipeline : 2000 séquences d'ARN de longueur 100, avec un seuil ε = 70 pour filtrer les arêtes du graphe DOT.
Le temps total (calcul des distances + rassemblement) passe d'environ 234 ms avec 1 processus à environ 79 ms avec 6 processus, ce qui montre une nette accélération lorsque l'on augmente le nombre de processus.

La courbe de performance est quasi monotone décroissante : on observe un gain important entre 1 et 4 processus, puis un ralentissement de l'amélioration entre 4 et 6 processus. Cette saturation provient du coût fixe des communications MPI et du rassemblement final de la matrice sur le rang 0.

6. Conclusion

Ce projet m'a demandé beaucoup de temps et d'efforts. Rien que comprendre comment toutes les étapes s'enchaînent, lire les séquences, construire la matrice de Hamming, appliquer Floyd–Warshall puis lancer PAM m'a pris plusieurs jours.

Finalement, ce travail m'a vraiment fait progresser : j'ai appris à utiliser MPI dans un vrai projet et organiser plusieurs algorithmes qui dépendent les uns des autres, ce que je n'avais encore jamais fait à cette échelle.

Même si ce n'était pas simple, ce projet m'a beaucoup apporté et m'a donné une meilleure compréhension concrète de la programmation parallèle.