UNIVERSITÉ D'ORLÉANS **M1 Info**

Informatique Année **2025 - 2026**

Projet **Floyd - k-médoïdes/PAM + Application aux séquences d'ARN**

Soit un graphe pondéré non orienté $G(V, E, w)$ où V est un ensemble de $|V| = n$ sommets, E l'ensemble des arêtes et w une fonction $E \to \mathbf{N}$. La fonction w formalise un graphe pondéré dont le poids des arêtes sont des entiers strictement positifs.

L'objectif global du projet est de

1. calculer par l'**algorithme de Floyd** la matrice des plus courtes distances entre tous les couples de sommets d'un graphe pondéré non orienté
2. appliquer un algorithme **PAM Partitioning Around Medoids** pour construire k partitions de nœuds du graphe.
3. d'utiliser ces deux algorithmes sur un exemple de jeu de données de la construction du graphe à la construction des k partitions.
4. d'implémenter une version parallèle pour une architecture à mémoire distribuée en utilisant la librairie MPI.

1 L'algorithme de Floyd

1. **La description de l'algorithme**

   L'algorithme de Floyd[1] permet de calculer la matrice D des plus courtes distances entre tous les sommets d'un graphe. On part de la matrice d'adjacence adaptée

   $$a_{ij} = \left| \begin{array}{c} w(v_i, v_j) \ si \ (v_i, v_j) \in E \\ 0 \ si \ i = j \\ \infty \ sinon \end{array} \right.$$

   En posant $D^{(-1)} = A$, on met à jour la matrice à chaque itération l en autorisant le sommet v_l comme point intermédiaire possible dans les chemins reliant v_i à v_j :

   $$d_{ij}^{(l)} = \min(d_{ij}^{(l)}, d_{il}^{(l-1)} + d_{lj}^{(l-1)}).$$

   Ainsi, après n itérations, $D^{(n-1)}$ contient les longueurs des plus courts chemins entre tous les couples de sommets.

2. **La parallélisation**

   Le schéma de la parallélisation repose sur un découpage de la matrice $D^{(l)}$ par blocs de taille $b \times b$. On note $D^{(l)} = (D_{ij}^{(l)})_{0 \leq i,j < \frac{n}{b}}$ la matrice des blocs. Lors de l'itération $l+1$, le calcul de chaque bloc $D_{ij}^{(l+1)}$ dépend uniquement d'une colonne de $D_{i(l\%\frac{n}{b})}^{(l)}$ et d'une ligne de $D_{(l\%\frac{n}{b})j}^{(l)}$. Une fois ces éléments disponibles, tous les blocs $D_{ij}^{(l+1)}$ peuvent être calculés indépendamment les uns des autres.

3. **L'implémentation**

   Vous disposez d'une première version séquentielle utilisant des graphes décrits par un fichier .dot (Graphviz). Le code est commenté et peut être modifié si nécessaire pour l'implémentation de votre parallélisation. Vous pouvez également ajouter une fonction de génération aléatoire d'une matrice d'adjacence pour tester des graphes de différentes tailles.

   Vous pouvez formuler des hypothèses favorables sur les paramètre n, b et le nombre *nprocs* de processus utilisés dans l'exécution parallèle :

[1] Pour une autre explication de cet algorithme vous pourrez vous référer à la section 10.4.2 du livre *"Introduction to Parallel Computing (ananth Grama et al.)*. Vous trouverez le lien correspondant sur Celene.

- n est divisible par b
- $\frac{n}{b} \times \frac{n}{b} = \sqrt{nprocs} \times \sqrt{nprocs}$

2 L'algorithme PAM

1. **La description de l'algorithme**

   On souhaite partitionner les sommets du graphe en k groupes à partir de la matrice de distance $D^{(n)}$. L'algorithme PAM est une alternative à l'algorithme des k-moyennes quand la notion de moyenne n'est pas définie sur les données. Il repose sur le principe des k-médoïdes qui cherche à déterminer k médoïdes (ici les sommets du graphe) qui minimisent un coût total défini par la somme des distances entre chaque sommet et le médoïde le plus proche.

   On note

   $$E_{k,n} = \{(x_0, \ldots, x_{k-1}) | 0 \leq x_i < n\}$$

   l'ensemble de tous les k-uplets d'entiers entre 0 et $n-1$. Pour un k uplet $(x_0, \ldots, x_{k-1}) \in E_{k,n}$ le coût associé est

   $$C_{(x_0,\ldots,x_{k-1})} = \sum_{i=0}^{n-1} min(D_{ix_0}, \ldots, D_{ix_{k-1}})$$

   L'algorithme des k-médoïdes consiste alors à déterminer l'ensemble optimal de médoïdes (on représente le médoïde par l'indice du sommet correspondant)

   $$(y_0, \ldots, y_{n-1}) = argmin_{(x_0,\ldots,x_{k-1}) \in E_{k,n}} C_{(x_0,\ldots,x_{k-1})}$$

   Chaque partition P_{y_i} est alors constitué du médoïde v_{y_i} et de tous les sommets qui sont les plus proches de lui parmi tous les médoïdes. L'algorithme PAM est une version heuristique et sous optimal de cet algorithme exact. Il procède comme suit :

   (a) Choisir aléatoirement k sommets comme médoïdes initiaux et calculer le coût correspondant.

   (b) Pour chaque sommet non médoïde, tester l'échange avec un des médoïdes actuels. Si le coût total diminue, valider l'échange et mettre à jour l'ensemble des médoïdes.

2. **La parallélisation**

   Aucune consigne particulière n'est donnée pour la parallélisation de cet algorithme. Vous pouvez chercher à répondre aux questions suivantes

   - Quels sont les calculs indépendants ?
   - Les données comme la matrice de distance sont-elles réparties de manière adaptée à la stratégie de parallélisation envisagée ?
   - Quelles sont les communications nécessaires aux itérations d'échange des médoïdes ?

3 Application à des données séquences d'ARN

Dans cette partie il s'agit d'utiliser le travail précédent sur des séquences d'ARN qu'on peut voir comme des suites de lettres 'A', 'C', 'T' et 'G'. Afin de produire une classification de ces séquences il faut construire le graphe $G(V, E, w)$ où les sommets de V sont les séquences et où E et w sont définies par :

$$\forall v_i, v_j \in V, (v_i, v_j) \in E \; si \; w((v_i, v_j)) < \epsilon$$

$$\forall e = (v_i, v_j) \in E \; w(e) = d(v_i, v_j)$$

La distance d utilisée sera la distance de Hamming qui calcule le nombre de positions pour lesquelles les caractères différent.

Sur Celene vous trouverez deux jeux de données de respectivement 500 et 2000 séquences (`dataset_500seq` et `dataset_2000seq`) qui sont toutes de longueur 100. Vous pourrez faire l'hypothèse que toutes les séquences sont de la même longueur et c'est le cas dans les 2 jeux de données disponibles.

Vous devrez pour cette partie

1. Etudier comment calculer efficacement l'ensemble des distances des séquences deux à deux pour générer la matrice d'adjacence.
2. Appliquer ce que vous avez fait à partir de la matrice d'adjacence pour calculer la matrice des plus courtes distances entre toutes les séquences
3. Appliquer l'algorithme PAM pour proposer une classification des séquences

Dans un premier temps vous utiliserez les paramètres $\epsilon = 70$ et $k = 4$. Vous pourrez également tester d'autres valeurs de votre choix pour vous permettre de voir leur influence sur la performance de votre parallélisation.

4 Consignes pour le rendu : partie 1 (MPI)

Au final vous devrez rendre une archive via Celene **avant le 3 décembre 23h59** avec les éléments suivants

1. **Un rapport**

   Le rapport entre 2 et 4 pages devra faire un bilan du travail réalisé. Il devra également donner des éléments sur l'efficacité de vos différentes parallélisations. Vous pourrez donner des courbes d'accélération ou d'efficacité en faisant varier le nombre de processus utilisés pour différentes tailles du problème. Vous devrez montrer que vous avez analysé quels sont les paramètres qui influencent la performance de votre programme.

2. **Rendu pour la parallélisation de l'algorithme de Floyd**

   Vous devrez dans un dossier ***Floyd*** placer les codes sources, le `Makefile`, des exemples de graphes et un `ReadMe` donnant les instructions pour compiler et exécuter la parallélisation de l'algorithme de Floyd.

3. **Rendu de l'application complète**

   Dans un deuxième dossier vous devrez rendre l'intégralité de votre projet avec également un `Makefile` et un `ReadMe` expliquant comment compiler et exécuter sur différents jeux de données.

   Tous vos codes devront être commentés au format Doxygen (https://www.doxygen.nl/index.html)

4. **Consignes supplémentaires**

   - Vous pouvez travailler seul ou par binôme. Sur le wiki sur Celene, remplissez le tableau avec les noms de votre binôme ou votre nom si vous travaillez seul.
   - Un seul rendu par binôme.

5 Alignement, score par l'algorithme de Needleman-Wunsch

Dorénavant, pour construire le graphe sur les séquences, on souhaite utiliser une mesure plus précise que la distance de Hamming. Pour cela, vous utiliserez un algorithme d'alignement global permettant de calculer un score représentatif de la similarité entre deux séquences. Le score retourné par cet algorithme remplacera la distance utilisée dans la première partie.

L'algorithme imposé est celui de Needleman–Wunsch, dont une description détaillée est disponible sur Wikipédia `https://fr.wikipedia.org/wiki/Algorithme_de_Needleman-Wunsch`. Le schéma de pénalités retenu est le suivant :

- +1 pour un match (caractères identiques),
- –1 pour un mismatch (caractères différents),
- –3 pour l'ouverture d'un trou (gap opening),
- –1 pour l'extension d'un trou (gap extension).

Un trou (gap) correspond à l'insertion d'un ou plusieurs symboles - dans l'alignement afin de modéliser une insertion ou une suppression dans l'une des deux séquences. Un nouveau trou entraîne un coût d'ouverture, tandis que la poursuite du même trou entraîne un coût d'extension.

Dans cette partie vous devez implémenter une version séquentielle de cet algorithme et utiliser des directives OpenMP pour paralléliser votre code. Pour valider votre implémentation vous utiliserez les séquences déjà fournies. L'algorithme de Needleman–Wunsch permet de trouver l'alignement mais ici seul le score est à calculer.

6 Reprise de l'application complète

En remplacant la distance de Hamming par le score défini par l'algorithme de Needleman-Wunsch vous devez reprendre le projet complet en essayant de travailler sur toutes les parties du code qui peuvent être optimisées grâce à de la programmation hybride. Autrement dit chaque processus MPI peut être également multithreads. Le programme ci-dessous montre comment initialiser l'environnement MPI pour permettre à un processus d'être multithreads.

```
#include <iostream>
#include <mpi.h>
#include <omp.h>

using namespace std;

int main(int argc, char **argv) {
    int pid, nprocs;
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    cout << "provided=" << provided << endl;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

#pragma omp parallel
    {
        int num = omp_get_thread_num();
#pragma omp critical
     cout << "je suis " << pid << "/" << num << endl;
    }

    MPI_Finalize();
    return 0;
}
```

Attention au paramètre `provided` en sortie de l'initialisation. Si sa valeur est inférieure à 3 (`MPI_THREAD_MULTIPLE`) ça signifie que les threads ne pourront pas accéder aux fonctions MPI en concurrence. Il sera nécessaire d'ajouter de la synchronisation.

En suivant cette approche, l'exécution utilise p processus MPI, chacun exploitant th threads OpenMP. Vous devrez étudier comment choisir ces paramètres pour obtenir les meilleures performances. En particulier, il s'agit d'identifier, en fonction de la taille du problème, la répartition optimale du travail entre :

- une parallélisation à gros grain réalisée avec MPI, où chaque processus traite une partie des données ou du traitement
- une parallélisation à grain fin au sein de chaque processus (directives OpenMP), pour accélérer les calculs locaux.

L'objectif est de montrer comment la combinaison entre nombre de processus MPI et nombre de threads par processus influence les performances, et dans quelles configurations le programme atteint les meilleurs performances.

7 Consignes pour le rendu : partie 2 (OpenMP - hybride)

Vous devrez rendre une archive via Celene **avant le ?? janvier 23h59** avec les éléments suivants

1. **Un rapport**

   Le rapport entre 2 et 4 pages expliquera la parallélisation du calcul du score en remplacement de la distance de Hamming. Vous devrez également justifier ce que vous avez optimisé dans le projet complet grâce à une parallélisation des processus MPI par des directives OpenMP.

2. **Rendu pour la parallélisation de l'algorithme de Needleman-Wunsch**

   Vous devrez dans un dossier ***Needleman*** placer les codes sources, le `Makefile` et un `ReadMe` donnant les instructions pour compiler et exécuter la parallélisation de l'algorithme de Needleman. Ce package doit permettre de tester cette partie indépendamment du reste de l'application

3. **Rendu de l'application complète**

   Dans un deuxième dossier vous devrez rendre l'intégralité de votre projet avec également un `Makefile` et un `ReadMe` expliquant comment compiler et exécuter sur différents jeux de données.

   Tous vos codes devront être commentés au format Doxygen (https://www.doxygen.nl/index.html)