Programmation Distribué

RAMBAL JULIEN , M1-IASD

October 18, 2023

1 Introduction :

1.1 Présentation du projet

Lors de ce projet nous, cherchons à développer une application capable de générer un graphe sous forme de réseau de processus à partir d'un fichier au format graphe DIMACS graph, puis de résoudre le problème de coloration de ce graphe avec un minimum de couleurs et en un minimum de temps.

1.2 Problématique et objectifs

Cette première partie concerne la modélisation de ce graphe en réseau de processus. Nous devons donc dans un premier temps récupérer les informations du fichier représentant le graphe et trouver un moyen de l'implémenter. Cette étape est à réaliser en C , nous allons donc nous servir de compétences personnelles pour développer un parseur de fichier DIMACS ainsi que de celles acquises en cours et TD (notamment lors du TP sur le réseau en anneau) pour modéliser le graphe sous la forme d'un réseau. Comme lors de ce TP, une arête entre deux nœuds se présentera sous la forme d'une connexion par socket. Il conviendra aussi d'implémenter une communication efficace entre ces processus afin de permettre l'utilisation d'un algorithme distribué pour résoudre le problème de coloration en seconde partie.

2 PARTIE 1 : Parsing

2.1 Analyse du format

Dans un premier temps, nous analysons les fichiers fournis à l'adresse :

http://cedric.cnam.fr/~porumbed/graphs/

Après de brèves recherches, nous apprenons que ce format DIMACS graph est assez simple.
Il est constitué de lignes correspondant soit à des commentaires, soit aux caractéristiques du graphe 'p' , soit à une arête 'e'.

```
c FILE: exemple.col
c Un commentaire pour un graphe de 10 n uds et 7 ar tes.
p edge 10 7
e 1 10
e 2 3
e 4 7
e 9 3
e 6 4
e 2 8
e 10 9
```

2.2 Réalisation du parseur

Tout comme pour le reste du projet, nous réalisons ce parseur en langage C afin de faciliter la compatibilité avec la partie du code qui générera le réseau.
Les informations essentielles pour générer le réseau seront :

- Le nombre de nœuds
- Le nombre d'arêtes
- Une matrice d'adjacence symétrique

Le retour du parseur devra donc être une structure contenant deux entiers pour le nombre de nœuds et d'arêtes ainsi qu'une matrice M en dimension deux ou l'élément M(i,j) sera un s'il existe une arête entre i et j, 0 sinon.

Nous avons dans un premier temps créer une telle structure avant d'implémenter un parsing en C à l'aide des fonctions liées au tokens qui ignore tout commentaire et toute ligne vide.

Le parseur se présente donc sous la forme d'une fonction qui demandera à l'utilisateur d'entrer le chemin vers le fichier à parser et retournera la structure GraphData créée au début. Nous implémentons aussi un affichage en console de la matrice d'adjacence sous la forme d'une méthode printTabtab_int(), utile à la fois pour les tests et pour visualiser les arêtes d'un graphe de petite taille.

```
struct GraphData {
    int ** matrix;
    int nodesQuantity; // nombre de noeuds
    int  edgesQuantity; // nombre d'arretes
    int * taille_r;
    int * tabNbEdges; //tableau du nombre d'arrete par noeud
};
```

3 PARTIE2 : Réseau

3.1 Specification :

Il est prévu que ce réseau de machines interconnecté puisse à terme être utilisé pour effectuer des opérations complexes nécessitant plusieurs machines.

Nous distinguons deux parties l'une gérant les clients et l'autre la gestion des nœuds appartenant déjà au réseau. Un client peut une fois connecté formuler une demande de connexion au réseau (client-noeud) ou une demande de calcul.

Nous supposons pour le moment que ces deux composantes sont gérées par une même machine, mais dans deux processus distincts et que nous n'avons pas à gérer les clients-noeuds.

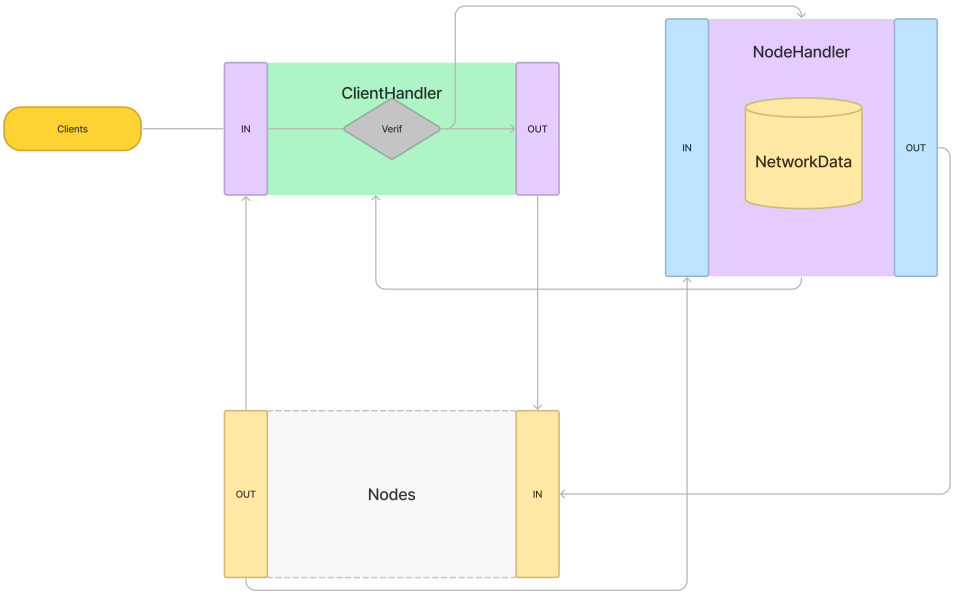
Finalement, chaque membre de ce réseau pourra effectuer plusieurs opérations simultanément grâce à sa threadpool (l'ensemble des threads qui lui ont été alloués) .

3.1.1 Organisation:

Tout d'abord nous allons présenter chacune des rôles des objets ci-dessous .

Ensuite nous verrons concrètement comment ils permettent de construire le réseau (graphe) de machines .

Finalement nous allons illustrer l'utilisation de la threadPool pour gérer les requêtes des clients .



3.2 Rôles

3.2.1 Machine

La classe machine est la classe centrale de cette modélisation , elle représente une machine du réseau .Elle est constituée de l'identifiant de la machine sur le réseau (uid) d'un ensemble de descripteurs de fichier d'entrée et de sortie (socket unidirectionnel) et finalement d'une threadPool .

Le but d'une machine est de pouvoir effectuer parallèlement des calculs demandés par le réseau puis de les retransmettre soit directement au clientHandler (si le calcul est le dernier requis pour la requête cliente considérée) soit à un autre nœud .

3.2.2 ComputeRequest (pas encore implémenté)

Les requêtes se distinguent en 2 sous-classes l'une est émise par un client , elle est constitué d'une opération complexe et est reçu par le ClientHandler , l'autre est émise par un Noeud ou par le ClientHandler et est reçu par un Noeud ou le ClientHandler .

La requête ClientRequest doit d'abord être décomposé et chacune de ses sous opérations assignées à un ensemble de nœud , finalement ces opérations sont envoyés sur le réseau .

Chaque nœud attend de recevoir un résultat de son précédent dans la "chaîne" de calcul avant d'effectuer l'opération qui lui a été demander sur ce résultat dans un thread de sa threadPool .Une fois l'opération terminée, il retransmet le résultat au suivant de la chaîne .

3.2.3 ClientHandler :

Un clientHandler est une Machine et possède en plus de cela un ensemble de sockets bidirectionnel destiné aux clients .

Tout d'abord le clientHandler attend de recevoir l'ensemble des demandes de connexion des clients-noeuds , pour chacune d'entre elles il échange l'adresse du NodeHandler avec le clients-noeud connecté .

Une fois que le réseau a été initialisé par le NodeHandler il s'apprête à recevoir des requêtes et demande de connexions uniquement clients . Cependant , s'il a été informé par le NodeHandler qu'un nœud du réseau a été déconnecté alors il peut à nouveau accepter une autre demande de connexion client-noeuds au réseau .

3.2.4 NodeHandler :

Le NodeHandler s'occupe de gérer le réseau et de stocker les informations importantes sur sa topologie .

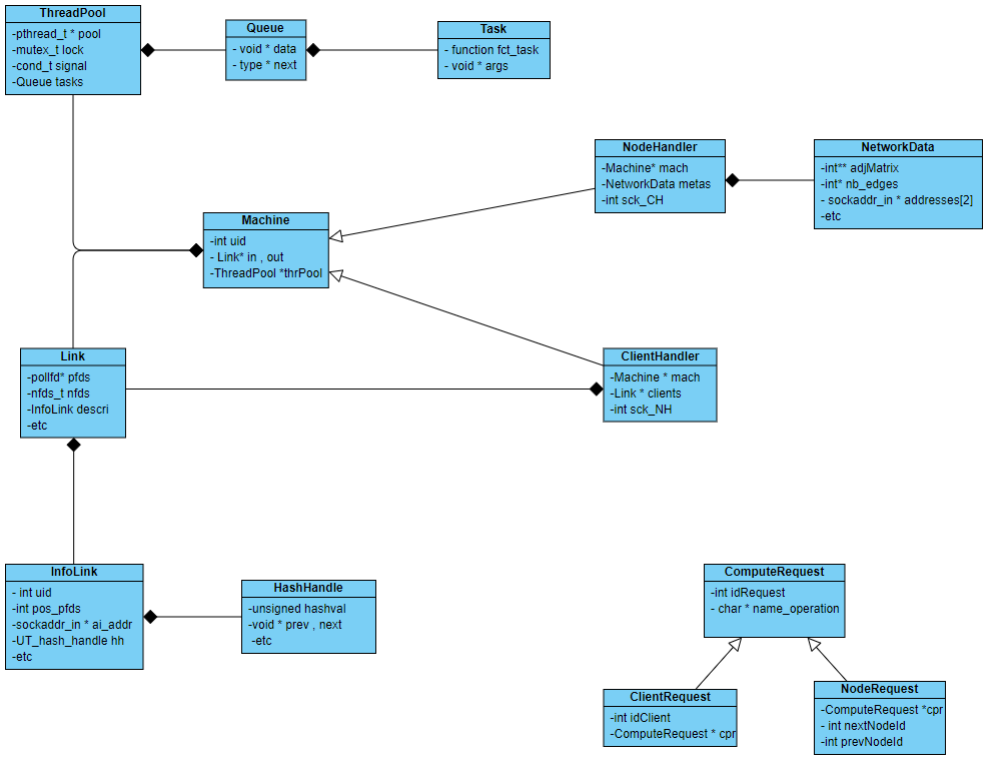
Son premier rôle est d'établir une connexion avec le clientHandler .Puis il parse un fichier DIMACS et stocke sont contenu sous la forme d'une matrice d'adjacence ainsi que d'un tableau représentant le nombre d'arêtes de chaque nœud .

Ensuite il informe le clientHandler du nombre de nœud qui doit être présent dans le réseau .

Puis le clientHandler lui fait parvenir les adresses des socket IN et OUT de chaque client-nœud , il s'y connecte et il les stocke dans son attribut addresses . Une fois qu'il a obtenu l'intégralité des adresses il leur envoie la liste des nœuds auxquels ils doivent se connecter (étape ou les client-noeud deviennent les noeuds du réseau) .

Finalement il se tient prêt à recevoir des demandes d'informations des nœuds du réseau, mais également des potentielles connexions dans le cas où il est informer qu'un nœud du réseau s'est déconnecté .

3.3 Implémentation :



3.3.1 NodeHandler Parser:

Le parser est invoqué dans le NodeHandler , il initialise une structure GraphData qui est ensuite utilisé pour remplir la structure NetworkData du NodeHandler

3.3.2 Setup

Le NodeHandler et le ClientHandler sont implémentés dans le même fichier , ils sont distingués par le numéro du processus qui les exécutes (cf. fork).

Tout d'abord , paire de socket est créée via la fonction socketpair , ils assureront la transmission d'informations entre les 2 processus . Cette approche permettra d'ultérieurement séparer le ClientHandler et le NodeHandler sur deux machines distinctes .

Ensuite les deux processus ClientHandler et NodeHandler sont créés .

3.3.3 ClientHandler :

Dans un premier temps le clientHandler mets en attente les demandes qui ne sont pas issu d'un client-noeud (dans le code ci-joint il est supposé qu'il n'y a pour l'instant pas de client autre que des client-noeuds) et accepte les demandes des N clients-noeuds .

Suite à la connection d'un client-noeud le clientHandler envoie l'adresse IN et OUT du client au NodeHandler puis celle du NodeHandler au client .

Une fois les N clients-noeuds connecté le clientHandler refuse les connexions de type client-noeud (si le NodeHandler ou le clientHandler s'aperçois qu'un nœud du réseau ne répond plus N est décrémenté) et reprend le traitement des demandes de type client .

3.3.4 NodeHandler :

Le NodeHandler refuse toute connexion si elles ne correspondent pas aux adresses stockées dans son attributs addresses à l'index 1 (addresses décrit des adresses IN:0 et OUT:1) . Notons qu'addresses est initialisé avec son adresse et rempli avec les adresses des clients-noeuds que le ClientHandler lui envoie .

Une fois que l'ensemble des adresses des clients-noeuds ont été renseigné et que leur connexion avec le NodeHandler a été établi , le NodeHandler envoie la liste des connexions que les noeuds-clients doivent établir pour être relié à leurs voisins IN et OUT . Cette liste est obtenue grâce à la matrice d'adjacence ainsi que l'attributs addresses .

3.4 Utilisation :

Avec *testServ* : l'éxecutable associé à leServeurTropPool3.c et *testClient* : l'éxecutable associé à testClient.c

- 1) Lancer les serveurs (clientHandler + nodeHandler) :

```
.\testServ PORT_CH_IN PORT_CH_OUT PORT_CH_INOUT NH_IN NH_OUT STRICT_PATH FILENAME

```

- 2) Lancer NB_NODE noeuds-clients :

```
.\testClient PORT_CH_IN IP_CH NB_NODE

```

Exemple d'utilisation :

- 1) Lancer les serveurs (clientHandler + nodeHandler) :

```
.\testServ 3490 3491 3492 3590 3591 . graph-test.txt
// si graph-test dans repertoire courant

```

- 2) Lancer 10 noeuds-clients :

```
.\testClient 3490 0.0.0.0 10

```

4 PARTIE 3 : Coloration

4.1 Algorithmie et Recherche

Dans cette deuxième partie, il nous a été demandé de concevoir un algorithme de coloration distribué entre les nœuds du réseau. Nos recherches nous ont conduites à considérer cinq algorithmes de coloration :

- Cole-Vishkin [5]
- Luby [6]
- Plassman Jones [7]
- Multi-trials Schneider & Wattenhofer[8]
- Maus Algorithm [9]

En considérant les informations que nous avions à notre disposition et notre organisation nous avons néanmoins décidé de nous orienter vers une recherche taboue.

Les raisons sont les suivantes :

- Les échanges entre les machines d'un réseau peuvent être moins rapides qu'un calcul raisonnable sur une machine singulière
- De nombreux travaux ont été effectuer sur les minimisations de conflits via tabu search (IA)
- De nombreuses architectures distribuent le temps de calcul sur le réseau (HDFS)
- Nous développons la partie réseau en parallèle de la partie algorithmie
- La topologie du graphe est connu et le graphe n'est pas interne au système d'une unique machine

Ainsi plutôt que d'effectuer atomiquement (au niveau du voisinage d'une machine) les décisions, nous considérerons une partie du graphe partageant une propriété commune : un cluster .

4.1.1 Cahier des charges :

- Convergence : en plus de la minimisation du nombre de conflits une liste taboue est utilisée.
- Topologie : un ensemble de configuration sera employé pour colorier chacune des topologies du graphe
- Distribution : la répartition de la charge est assuré par un algorithme de clustering
- Temps : une barrière de temps sera assuré par le serveur qui a pour tâche de regrouper les meilleures solutions (obtenues sous 1 minute)

Tout d'abord nous verrons la clusteurisation ,ensuite l'algorithme tabu search et finalement la réconciliation des solutions .

4.1.2 Clusteurisation :

Pour effectuer la clusteurisation nous nous appuierons sur l'algorithme proposé par J.Weiss (2002)[4] :

La matrice d'affinité sera calculée grâce au degré de chaque nœud ainsi que de la matrice d'adjacence .

$$M = DegreeMatrix - AdjMatrix$$

Ensuite le laplacien sera normalisé ce qui aura pour effet de borné la valeur des eigens .

$$M_{norm} = D^{-1/2}.M.D^{-1/2}$$

Finalement, nous appliquerons aux vecteurs propres de M_{norm} un algorithme de clustering : Kmeans .

Kmeans :

La mesure de distance employée est la *distance euclidienne* sur chacune des dimensions .

En supposant que les nœuds suivent une loi normale, les centroids seront initialisés selon les *nbCluster quantile* sur chacune des nbCluster dimensions .

Ces centroids seront recalculés autant de fois que nécessaire .

Remarque : Parfois des outliers (point disjoint du sous-graphe) se glisse dans cette clusterisation une étape de post-processing est envisageable pour distribuer plus convenablement ces nœuds .

4.1.3 Tabu Search :

Paramètres :

- Nombre de couleur
- Durée de tabou
- Durée Constant

La durée de tabou se décompose en plusieurs composante pondéré par des paramètres fixés qui ont été d'avance déterminer par un algorithme génétique .

La durée constant ou, autrement dit la durée limite sans amélioration de la solution de coloration sera quant à elle différente entre les nœuds calculant la coloration du cluster.

Finalement le la taille de la palette sera elle aussi différente pour chacun des nœuds .

Cette recherche mets donc en concurrence plusieurs configurations de paramètre . C'est pourquoi nous allons faire colorier à chaque nœud d'un cluster une coloration avec des paramètres distincts , le graphe étant trop important pour être colorié en temps convenable nous ferons colorier à chaque nœud son propre cluster .

4.1.4 Distribution :

La répartition du problème de coloration dans le graphe se fera comme suit :

Le serveur associe aux n_0 nœuds du cluster un nombre chromatique de départ parmi les [3;Δ] possibles . De nombreuses initialisations seront testées pour chacun des nombres chromatique (minimisant les chances d'avoir un manque de chance sur le tirage de départ) .

Dès qu'un nœud trouve une solution pour un nombre chromatique n , il l'envoie au serveur et passe au suivant : n-1 .

Le serveur de son côté mets à jour une solution avec les nombres chromatique les plus faibles de chaque cluster , l'expérience étant effectuer sous 1 minute le serveur lance un processus de coloration du graphe G à 60 - nbNoeudExterne/c seconde .

La coloration lancée par leur serveur laisse inchangées les couleurs des nœuds internes aux différents clusters tandis que les nœuds reliant des clusters différents ont une couleur modifiable .

Finalement, la coloration trouvée est envoyée à tous les nœuds ce qui signe l'arrêt de la recherche .

4.2 Communication (ouverture) :

Cette partie n'a pas été intégralement implémentée dans le code. Concernant son concept, nous voulions que les nœuds puissent s'envoyer des messages synchronisés de certains type avec différent contenus. Pour ceci, nous devions utiliser un header dans chaque message qui indiquerait le type de message et permettrait a la fonction d'interprétation des requêtes de savoir comment traiter le contenu du message.

Ce type de communication est adapté pour un algorithme de type tabu search et est suffisamment polyvalent pour s'adapter à divers besoins (demande de connexion , message d'erreur , clustering...etc).

References

[1] *Programmation multi-tâches* , Hinde Bouziane, 2021, https://moodle.umontpellier.fr/course/view.php?id=25054

[2] *Programmation concurrente et répartie* , Hinde Bouziane, 2022, https://moodle.umontpellier.fr/course/view.php?id=22441#section-3

[3] *Algorithmes distribués* , Rodolphe Giroudeau, 2022, https://moodle.umontpellier.fr/course/view.php?id=22441#section-4

[4] *On Spectral Clustering Analysis* ,Andrew Y. Ng,Michael I. Jordan and Yair Weiss, 2002 , https://ai.stanford.edu/~ang/papers/nips01-spectral.pdf

[5] *Deterministic coin tossing with applications to optimal parallel list ranking* ,Richard Cole and Uzi Vishkin, 1986 , https://dl.acm.org/doi/10.1016/S0019-9958%2886%2980023-7

[6] *A Simple Parallel Algorithm for the Maximal Independent Set Problem* ,Michael Luby, 1986 , https://epubs.siam.org/doi/10.1137/0215074

[7] *A Parallel Graph Coloring Heuristic* ,Mark T. Jones and Paul E. Plassmann, 1993 , https://epubs.siam.org/doi/10.1137/0914041

[8] *A New Technique For Distributed Symmetry Breaking* ,Johannes Schneider and Roger Wattenhofer, 2010 , https://dl.acm.org/doi/10.1145/1835698.1835760

[9] *Distributed Graph Coloring Made Easy* ,Yannic Maus, 2021 , https://dl.acm.org/doi/10.1145/3409964.3461804

[10] *Informed reactive tabu search for graph coloring* ,Daniel C. Porumbel, Jin-Kao Hao and Pascale Kuntz , 2013 , https://www.worldscientific.com/doi/abs/10.1142/S0217595913500103

[11] *A new genetic local search algorithm for graph coloring* ,RaphaËl Dorne and Jin-Kao Hao , 2006 , https://link.springer.com/chapter/10.1007/BFb0056916

A new genetic local search algorithm for graph coloring