Projet d'implementation de l'algorithme de Naïmi-Tréhel

Meryam RHADI
Lamyae KHAIROUN
Abla AMHARREF

29 décembre 2021

Master 1
génie logiciel
UE Programmation répartie

Responsable
Hinde Bouziane
Rodolphe Giroudeau

Sommaire

1 Introduction

1.1 Présentation du projet

Dans le cadre du module de programmation répartie, nos professeurs nous ont proposé d'implémenter l'un des trois algorithmes distribuées suivants :

- Le problème de l'élection d'un leader dans un graphe complet.
- le problème d'exclusion mutuelle basé sur l'algorithme de Maekawa.
- le problème d'exclusion mutuelle basé sur l'algorithme de Naïmi-Tréhel.

À notre tour on a choisi de travailler sur l'implementation du troisième algorithme de **Naïmi-Tréhel**. Dans ce rapport on va expliquer notre compréhension de l'algorithme, la technique qu'on a utilisée pour la programmation parallèle en donnant plus de détails sur notre code.

1.2 Diagramme de gant

Pour être mieux organisé dans notre travaille, on a réalisé un **diagramme de gant** qu'on à suivi à 85%.

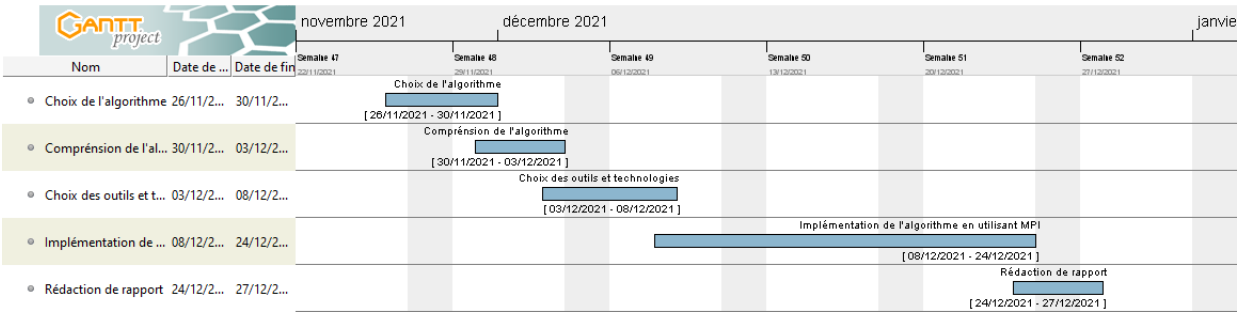

Figure 1. Diagramme de gantt

2 L'algorithme de Naïmi-Tréhel

2.1 Présentation de l'algorithme

L'algorithme de Naimi-Tréhel a été présenté en 1987 par Mohamed Naimi et Michel Trehel.

L'algorithme de Naimi-Tréhel résout le problème de l'exclusion mutuelle (*une primitive de synchronisation utilisée pour éviter l'utilisation d'une ressource partagée en même temps*) dans un réseau distribué, ce qui signifie que les processus de chaque site interagissent seulement en se

transmettant des messages, cet algorithme réduit le nombre moyen de messages à $\log(n)$ [3].

Il est basé sur le fait qu'un processus envoie sa requête à seulement un autre processus et attend sa permission, c'est une communication père à père, et le processus qui reçoit les requêtes changera en fonction de l'évolution des demandes.

La permission d'entrer en section critique est matérialisée par un jeton, le processus qui possède ce jeton peut entrer en section critique.

2.2 Spécification de l'algorithme de naimi-trehel

Dans cette partie on vas vous présenter les principaux règles de l'implementation de l'algorithme de Naimi-trehel [3], dont on est basés pour faire notre implémentation.

Algorithm 1 Initialisation du première processus qui prend le jeton qui a i égale à 0

$pere \leftarrow 0$
$suivant \leftarrow -1$
$estDemandeur \leftarrow false$
if $pere == i$ **then**
  $jetonPresent \leftarrow true$
  $pere \leftarrow -1$
end if

Algorithm 2 Demande de la section critique par le processus i

$demande \leftarrow true$
if $pere == -1$ **then**
  $entrerEnSC()$
else
  $envoyerRequete(i)$
  $pere \leftarrow -1$
end if

Algorithm 3 Procédure de fin d'utilisation de la section critique

$demande \leftarrow false$
if $suivant \neq NULL$ **then**
  $envoyerJeton(suivant)$
  $jetonPresent \leftarrow false$
  $suivant \leftarrow -1$
end if

Algorithm 4 Réception du message Req(i) avec i est le demandeur

```
if pere == −1 then
    if k.estDemandeur then
        suivant ← i
    else
        jetonPresent ← false
        envoyerToken(i)
    end if
else
    envoyerRequete(i)
    jetonPresent ← false
end if
pere ← i
```

Algorithm 5 Réception du message Jeton

```
jetonPresent ← true
```

3 Outils utilisées pour réaliser la distribution

Pour réaliser ce projet, on a utilisé principalement le langage C++, et on se qui concerne l'architecture à utiliser dans la réalisation de la partie distribuée, on s'est orienté vers l'utilisation des services de la bibliothèque **MPI(Message Passing interface)** pour simuler un environnement distribué. Puisque malheureusement on n'a pas déjà fait la programmation client/-serveur en **TCP/IP**, ainsi que pour les **sockets**, donc on n'a pas trop d'informations et de capacités pour les utiliser pour ce projet, cependant lors de nos recherches pour trouver une alternative, on a découvrit cette bibliothèque qui est très utile pour réaliser la distribution d'une manière simple, on va vous présenter cette bibliothèque dans la section suivante.

3.1 Présentation de MIP (Message Passing Interface)

Message Passing Interface (MPI), est une bibliothèque de fonctions, utilisable avec les langages C/C++ et Fortran conçue en 1993, elle permet le passage de messages entre ordinateurs distants ou dans un ordinateur multiprocesseur [4].
Il est devenu de facto un standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée, il permet de créer des processus (d'une manière automatique) dont chacun exécute le même programme à l'aide de la commande "mpirun -np nombreProcessus nomProgramme" [2].

Lors de l'initialisation de l'environnement, MPI regroupe tous les processus crées sous le communicateur prédéfinie *MPI_COMM_WORLD* (Communicateur= ensemble de processus qui ont la capacité de communiqué entre eux + contexte de communications), dans ce groupe de processus, chacun se voit attribuer un rang unique et ils communiquent explicitement les uns avec les autres par leurs rangs.

Généralement on peut dire que MPI gère :

- Les communications unidirectionnelles.
- La création dynamique de processus.
- Le multithreading.
- Les entrées/sorties parallèles.
- Les communications point à point qui consiste à permettre à deux processus à l'intérieur d'un même communicateur d'échanger une donnée ;

3.2 Explication de type de MPI et fonctions utilisées dans notre projet

Dans notre projet, on a travaillé sur les communications point-à-point de MPI, qui impliquent un expéditeur et un destinataire, ils permettent à ces deux processus qui sont à l'intérieur d'un même communicateur d'échanger une donnée sous forme de message, la base de la communication repose sur les opérations d'envoi et de réception entre les processus.
Un processus A peut envoyer un message à B en fournissant le rang du processus et l'étiquette unique pour identifier le message.
Le destinataire peut alors publier une réception pour un message avec une étiquette donnée, puis gérer les données en conséquence[1].

Dans cette partie, on vas expliquer les étapes qu'on a fait pour utiliser MPI, et les principales fonctions de cette bibliothèque qu'on a utilisé pour l'implémentation de notre algorithme :

- La première étape pour créer un programme qui utilise MPI consiste à inclure les fichiers d'en-tête MPI avec **include <mpi.h>**
- Après cela, l'environnement MPI doit être initialisé l'infrastructure nécessaire à la communication avec la fonction **MPI_Init(&argc, argv)**.
- la fonction **MPI_Comm_size(MPI_COMM_WORLD, &processus.size)**, dans notre implementation, **MPI_COMM_WORLD** est construit pour nous par MPI, englobe tous les processus dans le travail, donc cet appel doit renvoyer combien de processus totales sont actifs.
- **MPI_Comm_rank** qui renvoie le rang d'un processus dans un communicateur, chaque processus de notre environnement à l'intérieur d'un communicateur se voit attribuer un rang de 0 jusqu'au n-1(n

nombre de processus), les rangs sont pour l'identification lors de l'envoi et de la réception de messages.

Après on a utilisé MPI_Send et MPI_Recv pour effectuer une communication point à point standard.

- **MPI_Send(&msg, MSG_SIZE, MPI_INT, destination, type, MPI_COMM_WORLD)**, c'est une fonction qui permet d'effectuer un envoi bloquant d'un message d'un processus A vers B (il est bloquant car le processus B sait qu'il vas recevoir un message de A, donc il attends une fois cela est fait, les deux processus peuvent reprendre le travail) , le première argument contient le contenu à envoyer sous forme d'une structure, deuxième et troisième arguments décrivent le nombre et le type d'éléments qui résident dans le première argument, après on a destination, tag et communicator.
- **MPI_Recv(&msg, MSG_SIZE, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)**, cette fonction est utilisé dans MPI pour la réception d'un message envoyé par MPI_Send, les trois premières arguments sont comme MPI_Send, et le quatrième et cinquième arguments spécifient le rang du processus de réception et l'étiquette du message, le sixième c'est pour spécifié le communicateur et le dernier argument (pour MPI_Recvuniquement) fournit des informations sur le message reçu.
- **MPI_Finalize** c'est la dernière appel de MPI effectué, il est utilisé pour nettoyer l'environnement MPI.

4 Implémentation de l'algorithme

4.1 Explication des fonctions utilisées

Le fonctionnement de notre programme repose sur l'implémention des 5 parties qui définissent algorithme de Naimi-Trehel qu'on a décrit précédemment 2.2.
Donc on a implémenté dans un première temps trois structures :

- **Processus** : dont on définit tout les informations dont on a besoin pour un processus (pid, père, suivant, ainsi que deux variables booléen qui indique s'il est demandeur ou il a le jeton).
- **typeDeMessage** : Cette structure permet de définir le type de message envoyé par un processus donné. On a deux types de messages, soit REQUÊTE qu'on a initialisé à 0, sinon JETON qui est initialisé à 1.
- **Message** : Elle définit le type de message, ainsi que le dessinateur qui a envoyé le message en question. On l'a définit par la variable "from".

On a implémenté ensuite les 5 parties de l'algorithme 2.2 à travers les fonctions suivantes :

- **envoyerMessage()** : cette fonction permet à un processus A d'envoyer un message à un autre processus B, dans notre cas on a un processus qui veut accéder à la section critique envoie un message au processus qui possède le jeton, sinon un processus père qui possède le jeton veut l'envoyer à un processus demandeur, cette fonction est ajouté pour être utilisé par d'autres fonctions qu'on vas expliquer juste après, on a utilisé dans cette fonction **MPI_Send** dont sont fonctionnement est déjà expliquer dans la subsection 3.2.
- **recevoirMessage()** : cette fonction permet la réception d'un message envoyé par MPI_Send, et on a utilisé **MPI_Status** qui permet de représenter l'état d'une opération de réception renvoyé par des opérations de réception (MPI_Recv). 3.2
- **demanderAcces()** : cette fonction permet de décrire le scénario de l'envoie d'une requête de demande d'accès à la SC, par un processeur qui est dans l'état de demande et il a déjà un père, et pour vérifier que chaque processus qui fasse la demande il devient la nouvelle racine on initialise son père par -1, comme il est décrit dans l'algorithme 2.2.
- **libererAcces()** : Cette fonction permet au processus qui possède déjà le jeton de le libérer et envoyer le jeton à son suivant, on a appliqué la syntaxe de l'algorithme déjà cité 2.2.
- **recevoirRequete()** : cette fonction décrit le scénario de réception d'une requête de la part d'un processus demandeur au processus qui possède le jeton, pour appliquer cette condition on vérifie dans un première temps que le processus c'est bien une racine, et après qu'il est un demandeur donc son variable demandeur est égale à true, si c'est le cas on initialise que c'est le suivant pour utiliser le jeton, sinon on a traité aussi la cas dont c'est le première processus qui vas utilisé le jeton, donc on lui donne le jeton. Et si il a un déjà père donc il a pas encore fait la demande donc on fait une requête pour demander l'accès 2.2.
- **SECTION_CRITIQUE()** : A travers cette fonction on donne accès à un processus pour entrer en section critique. Pour ce faire on appelle en premier la fonction *demanderAccees()* pour demander et puis accéder à la section critique, et vers la fin on appelle la fonction *libererAcces()* pour que le processus libère l'accès et sors de la file d'attente.
- **lanceur()** : cette fonction nous permet de lancer la fonction *SECTION_CRITIQUE* pour tous nos processus.
- **main** Pour faire le teste de notre algorithme distribué, on appel dans un première temps la routine de l'environnement **MPI**, dont

les étapes sont déjà expliqué dans la section 3.2, donc tous les processus dont on choisi leurs nombres lors de l'exécution de notre programme font exactement la même chose, et après on vas faire une condition pour permettre l'initialisation d'un seule processus qui vas prendre la ressource et dans notre cas , on a choisi de le donnée au processus avec le rang égale à 0, maintenant on peut commencer l'algorithme donc on vas lancer tous les threads d'une manière séquentielles, et appeler la fonction lanceur pour enchaîner l'algorithme et après faire une boucle infini qui peut être arrêté seulement si on tue le processus, dans la boucle on vas faire un switch sur le type de message, donc on vas traiter le cas d'une requête et un jeton, et on finit par l'instruction de la commande **MPI_FINALIZE()**.

Pour tous ces fonctions et même le main on a essayé de faire un petite affichage pour faciliter le suivi et la compréhension de l'algorithme.

4.2 Affichage

```
Le prcessus numero 0 attend pour avoire l'acces à la ressource critique...
Processus numero 0 est entrer pour utiliser la ressource critique
Le processus numero 2 a envoyer une requete au processus numero 0
Le prcessus numero 2 attend pour avoire l'acces à la ressource critique...
Le processus numero 1 a envoyer une requete au processus numero 0
Le prcessus numero 1 attend pour avoire l'acces à la ressource critique...
Le processus numero 3 a envoyer une requete au processus numero 0
Le prcessus numero 3 attend pour avoire l'acces à la ressource critique...
Le processus numero 0 a recu   une reqeuete du processus numero 2
Le processus numero 0 a recu   une reqeuete du processus numero 1
Le processus numero 0 a envoyer une requete au processus numero 2
Le processus numero 0 a recu   une reqeuete du processus numero 3
Le processus numero 0 a envoyer une requete au processus numero 1
Le processus numero 2 a recu   une reqeuete du processus numero 1
Le processus numero 1 a recu   une reqeuete du processus numero 3
Processus numero 0 a quitté la ressource critique
Le processus numero 0 a envoyer un jeton au processus numero 2
Le processus numero 0 a envoyer une requete au processus numero 3
Le prcessus numero 0 attend pour avoire l'acces à la ressource critique...
Le processus numero 2 a recu   un jeton du processus numero 0
Processus numero 2 est entrer pour utiliser la ressource critique
Le processus numero 3 a recu   une reqeuete du processus numero 0
```

Figure 2. Résultat de l'éxecution

Dans l'exemple d'affichage affiché ci-dessus 2, on a choisi de travailler avec 4 processus, dans la première et la deuxième lignes on affiche que 0 attend, et après accède à la section critique, puisque dans notre code on a initialisé que le processus avec pid 0 c'est lui qui vas accéder en première à la ressource.
Après on a les processus numéro 2, 1 et 3 dans cet ordre on envoyé à tous au processus numéro 0 pour accéder à la ressource, puisque jusqu'à présent c'est lui le père.
Après on voit que le processus qui possède le jeton commence à recevoir les requêtes des processus, et déclare au processus concernés qu'il a bien reçu leurs demandes.
Et quand le processus 0 quitte la ressource, il envoi le jeton à son suivant qui est le processus 2 qui accède à la ressource, et maintenant on a processus 0, il veut ré-entrer à la SC, donc il cherche la racine(le dernière

demandeur) qui est 3 et il lui envoi la demande comme en le voie dans l'image 2, ainsi de suite...

5 Read-me :

Pour le lancement du projet dans un environnement Ubuntu, il suffit d'avoir les fichiers de notre projet :

- algoNaimi.cpp
- structure.h

Sur le terminale, on compile par la commande :
mpic++ -std=c++17 -o algoNami algoNaimi.cpp
Et on lance et exécute par la commande :
mpirun -host localhost:4 -np 4 ./algoNaimi
Dans cet exemple on a travaillé avec 4 processus, et puisque on a travaillé sur un seule ordinateur on a ajouté localhost.

0.8

Références

[1] OEIS Foundation. *MPI explication*. MPI. 2019. url : https://mpitutorial.com/tutorials/mpi-introduction/.

[2] techno-sciences. *Message Passing Interface*. MPI. 2019. url : https://www.techno-science.net/definition/1476.html.

[3] wiki. *Algorithme de Naimi-Trehel*. algorithmes distrubiées. 2018. url : https://fr.wikipedia.org/wiki/Algorithme_de_Naimi-Trehel.

[4] wikipedia. *Message Passing Interface*. Technologie pour la programmation en parallèle. 2021. url : https://fr.wikipedia.org/wiki/Message_Passing_Interface.