17/20

BERTRAND Olivier Devoir 1 AC

Exercice 1 6 /6

Q 1.1

L'algorithme simule la diffusion de la chaleur dans une barre homogène.

La barre sera modélisée par un tableau de flottants dont chaque élément représentera la température en un point de la barre.

Au lancement de l'algorithme, tous les éléments du tableau ont la même valeur sauf le 1er qui a une valeur beaucoup plus élevée (la barre est à une température donnée, on chauffe une extrémité et on veut observer l'évolution de la température).

ok

Exemple :

La barre, représentée ici par un tableau de 9 éléments, est au départ à une température de 20^oC lorsqu'on porte son extrémité gauche à la température 1000^oC.

1000	20	20	20	20	20	20	20	20

J'effectue la simulation séquentielle en Java dans une unique classe DiffusionSequentielle contenant le programme principal et quelques fonctions utilitaires.

3 variables sont définies :

- barre est le tableau de flottants représentant la barre à un instant donné.
- nouvelEtat est le tableau de flottants représentant la barre à l'instant suivant (évolution du tableau après une itération à partir de barre).
- n représente le nombre d'éléments des tableaux barre et nouvelEtat

Fonctions définies

- diffuse(float[ ] ancien,float[ ] nouveau)

  Cette fonction reçoit les références aux deux tableaux du programme. Elle s'appuie sur le 1er tableau (référençant la barre) pour modifier le 2e tableau représentant l'état de la barre à l'itération suivante du processus de diffusion (chaque case d'indice i du 2e tableau contient après l'exécution la moyenne des cases d'indices voisins de i du 1er tableau)

- recopie(float[ ] ancien,float[ ] nouveau) un échnage de références est suffisant, et plus rapide

  Cette fonction recopie les valeurs du 2e tableau passé en argument dans le 1er. Cela permet, après une itération, de préparer l'itération suivante en recopiant dans le tableau représentant la barre le tableau nouvelEtat qui représente les résultats d'une itération.

- stable(float[ ] ancien,float[ ] nouveau)

  Cette fonction renvoie un booléen indiquant si les deux tableaux passés en argument contiennent les mêmes valeurs.

  Elle permet de voir si le processus de diffusion est terminé en comparant l'évolution de l'état de la barre entre deux itérations (lorsque la diffusion est terminée, la température de tous les points de la barre est constante)

oui

Q 1.2 Comment paralléliser le calcul ?

La première idée qui vient pour aller plus vite est de délimiter dans la barre plusieurs sections et de lancer des itérations en parallèle sur chacune de ces sections.

Il y a cependant une difficulté : pour calculer la nouvelle valeur d'un élément, on utilise la valeur de cet élément et celles des deux éléments voisins. L'évolution d'un élément qui se trouve à l'extrémité d'une section est donc influencée par la valeur d'un élément qui se trouve à l'extrémité d'une autre section, comme l'illustre l'exemple ci-dessous :

Exemple :

La barre, représentée ici par un tableau de 9 éléments, est séparée en 3 sections.

1000	20	20	20	20	20	20	20	20

Evolution prévue des éléments lors des quatre premières itérations :

510	346.7	20	20	20	20	20	20	20

428.3	292.2	**128.9**	20	20	20	20	20	20

la prochaine valeur du 3e élément sera la moyenne de 292.2, 128.9 et 20

360.3	283.1	**147.0**	56.3	20	20	20	20	20

la valeur 56.3 sera maintenant prise en compte pour calculer l'évolution de ce 3e élément

321.7	263.5	162.2	74.4	32.1	20	20	20	20

Si on lance sur les différentes sections des processus non synchronisés, un processus qui s'exécuterait plus souvent que les autres utiliserait à l'extrémité de sa section des valeurs fausses. pas à jour

Imaginons par exemple que dans le cas de l'exemple illustré ci-dessus seul le processus traitant la 1ere section s'exécute. La 3e itération deviendrait :

360.3	283.1	147.0	20	20	20	20	20	20

La 2e section n'a pas été traitée car seul le processus 1 est actif

La non-évolution des valeurs de la 2e section influencerait alors l'évolution de la 1ere section :

4e itération dans le cas où seule la section 1 évolue

321.7	263.5	150.0	20	20	20	20	20	20

Si on considère le cas extrême dans lequel le processus traitant la 2e section n'intervient jamais, sa valeur constante va en fait se propager vers la gauche !

C'est évidemment une situation extrême mais elle illustre les défauts de la non-synchronisation. oui

Considérer chaque section comme une "sous-barre" en refusant à l'extrémité d'une section d'utiliser les valeurs de la section voisine ne résout rien car on empêcherait ainsi toute diffusion dans l'ensemble de la barre.

Une solution consiste à empêcher les processus de passer à une itération $n+1$ tant que tous les processus n'ont pas achevé l'itération n (on parallélise le calcul mais uniquement à l'intérieur de chaque itération). oui

On fixe alors un point de rendez-vous à la fin de chaque itération, ce qu'on peut implémenter avec des sémaphores ou des moniteurs.

Implémentation avec des moniteurs :

On crée 4 processus P_1, P_2, P_3, P_4 censés effectuer les calculs chacun sur une section du tableau.

On crée un moniteur Synchro pour gérer le rendez-vous.

Lorsqu'il a fini un calcul, chaque processus P_i appelle la fonction arrive() du moniteur. Cette fonction met en attente les processus tant qu'une itération n'a pas été achevée. Lorsqu'une itération est achevée, elle renvoie un booléen indiquant si les itérations doivent se poursuivre et réveille les processus endormis.

Remarque : pour savoir si les itérations doivent se poursuivre, on compare l'état de la barre avec celui qui doit être le sien à l'itération suivante. Lorsque la diffusion est terminée, ces deux états sont identiques.

Algorithme 1 : Algorithme de diffusion avec des moniteurs

```
variables
    n : entier taille du tableau représentant la barre
    barre : tableau de n flottants représentant l'état de la barre.
    nouveau : tableau de n flottants représentant la barre à l'itération suivante.
    Synchro : TDiffusionMoniteur

Programme principal
begin
    Initialisation de la barre :

    barre[0] = 1000
    for i=1 to n-1 do
        barre[i] = 20

    Création de 4 processus calculant la diffusion sur une partie de la barre :

    P1 pour les indices compris entre 0 et n/4 (non inclus)
    P2 pour les indices compris entre n/4 et n/2 (non inclus)
    P3 pour les indices compris entre n/2 et 3n/4 (non inclus)
    P4 pour les indices compris entre 3n/4 et n (non inclus)
```

Algorithme 2 : Code effectué par chaque processus de calcul P_i

```
Fonction Calcul
variables
    indice1 : entier
    indice2 : entier
    stop : booléen
begin
    tant que stop ≠ true faire
        for i = indice1 to indice2 do
            modifie nouveau[i] à partir des valeurs d'indices voisins de i dans barre
        stop = Synchro.arrive()
```

Algorithme 3 : Code du moniteur

```
Type TDiffusionMoniteur Moniteur

begin
    Variables
        compteur : entier
        nombreDeProcessus : entier
        TousArrives : condition
        stop : booléen

    Procedure Initialisation()
    begin
        compteur = 0

    Procedure arrive()
    begin
        compteur = compteur + 1
        if compteur < nombreDeProcessus then
            TousArrives.wait()
        Le dernier processus arrivé remet le compteur à 0, prépare la prochaine itération et réveille les
          autres processus
        if compteur == nombreDeProcessus then
            compteur = 0
            if barre == nouveau then stop = true else barre = nouveau;
            TousArrives.signal()
        renvoie la valeur de stop
```

Q 1.3 Remarques sur les performances de l'implémentation proposée en Java

La complexité des versions séquentielle et parallèle est linéaire.

1. Version séquentielle : parcours du tableau nouvelEtat pour le remplir, parcours des deux tableaux pour voir si la diffusion est terminée.

2. Version parallèlle : parcours d'1/4 du tableau nouvelEtat pour le remplir, parcours des deux tableaux pour voir si la diffusion est terminée.

Le fait que ma version "parallèle" effectue toujours une comparaison séquentielle des deux tableaux nuit à ses performances.

Paralléliser la comparaison des tableaux n'est pas simple car une section peut être stable pendant plusieurs itérations avant que des valeurs différentes soient propagées depuis une extrémité (cas d'une barre homogène dont on chauffe une des extrémités). C'est pour cette raison que la comparaison est faite dans le moniteur : elle considère l'ensemble des tableaux mais n'est faite qu'une fois entre deux itérations successives. ok

Il faudrait pour bien faire que la comparaison des deux tableaux soit effectuée en permanence par 4 autres threads. Une 2e rendez-vous entre ces threads permettrait d'arrêter les processus (mais il faut prévenir les processus de la fin du calcul)...

bien

Exercice 2 6,5/7,5 commentaires : +1

Cet exercice est une variante du problème producteurs/consommateurs : on a d'une part quatre producteurs et d'autre part un consommateur qui attend que la production ait atteint un niveau donné pour agir.

0,5 **Q 2.1** Problèmes de synchronisation à gérer :

- Un processus P_i ne peut pas déposer un biscuit sur un tapis plein
- Le processus Q ne peut pas commencer à ramasser des biscuits sur un tapis qui n'est pas complet.

oui

- Peut-on simultanément déposer plusieurs biscuits ? La gestion des emplacements libres doit se faire de toute façon en exclusion mutuelle pour éviter les accidents.
- Peut-on simultanément déposer et retirer des biscuits ? Tout dépend du type de ramassage.

3/3 **Q 2.2** Avec des sémaphores

Les machines considérées portent les noms *Biscuiteuse* et *Ramasseuse*...

L'algorithme ci-dessous utilise 2 sémaphores pour les procédures deposerBiscuit() et retirerBiscuit().

Le sémaphore *semVide*, initialisé à 10, gère le nombre d'emplacements vides. Il est décrémenté chaque fois qu'une biscuiteuse s'apprête à déposer un biscuit sur le tapis. Si le tapis est plein, l'appel à P(*semVide*) met le processus appelant en attente.

remarque : si deux biscuiteuses (par exemple) sont mises en attente, semVide prendra la valeur -2.
Les 2 prochains prélèvements sur le tapis ramèneront cette valeur à 0 et réveilleront ces machines. Les deux biscuits qu'elles déposeront ramènera bien le nombre d'emplacements vides à 0

Le sémaphore *semPaquet*, initialisé à 0, représente le nombre de biscuits disponible sur le tapis. Il permet de déclencher le ramassage/empaquetage : si on peut appeler P(*semPaquet*) 10 fois de suite cela siginifie que le tapis est plein. Dans le cas contraire le processus de ramassage est mis en attente.

Autrement dit :

- deposerBiscuit() diminue le nombre d'emplacements vide (mais attend si le tapis est plein) puis augmente le nombre d'emplacements occupés (et réveille 'retirerBiscuit()' s'il est en attente)

ok

- retirerBiscuit() vérifie qu'il peut diminuer 10 fois le nombre d'emplacements occupés (mais attend si ce nombre atteint 0) puis augmente 10 fois le nombre d'emplacements vides (et réveille éventuellement à chaque fois un processus 'deposerBiscuit()')

Algorithme 4 : Fonctions deposerBiscuit() et retirerBiscuit()

Variables

semVide : sémaphore gérant le nombre d'emplacements vides sur le tapis, initialisé à 10
semPaquet : sémaphore permettant de déclencher l'opération de ramassage/empaquetage, initialisé à 0

```
Procedure deposerBiscuit()
begin
    P(semvide)
    deposer()
    V(semPaquet)

Procedure retirerBiscuit()
begin
    for i = 1 to 10 do
        P(semPaquet)
    for i = 1 to 10 do
        retirer()
        V(semVide)
```

Q 2.3 Avec des moniteurs

Non, il faut les « appeler » depuis le squelete proposé

Les procedures deposerBiscuit() et retirerBiscuit() sont intégrées au moniteur. Sinon solution Ok

Le moniteur dispose d'une variable représentant le nombre de biscuits présents sur le tapis et de deux conditions, Plein et Incomplet, permettant de réagir respectivement aux états "le tapis est plein" et "le tapis compte moins de 10 biscuits".

Lorsqu'un processus P_i entre dans le moniteur pour exécuter deposerBiscuit(), il est mis en attente sur la condition Plein si le tapis compte déjà 10 biscuits.
Lorsqu'il a pu déposer un biscuit et que le tapis est complet, le processus Q est éventuellement réveillé s'il avait été mis en attente sur la condition Incomplet

Lorsque le processus Q entre dans le moniteur pour exécuter retirerBiscuit(), il est mis en attente sur la condition Incomplet si le tapis compte moins de 10 biscuits.
Lorsqu'il retire des biscuits, il réveille les éventuels processus P_i mis en attente sur la condition Plein

Algorithme 5 : Empaquetage avec des moniteurs

```
Type TBiscuitsMoniteur Moniteur

begin
    Variables
        compteur :entier        //nombre de biscuits sur le tapis
        Plein : condition
        Incomplet : condition

2/3 Procedure Initialisation()
    begin
        compteur = 0

    Procedure deposerBiscuit()
    begin
        if compteur == 10 then
            Plein.wait()        //mise en attente si le tapis est plein
        deposer()
        compteur = compteur + 1         //On a déposé un biscuit
        if compteur == 10 then
            Incomplet.signal()          //Réveil de la ramasseuse si elle avait été mise en attente

    Procedure retirerBiscuit()
    begin
        if compteur < 10 then
            Incomplet.wait()        //mise en attente de la ramasseuse si le tapis n'est pas plein
        for i = 1 to 10 do
            retirer()
            compteur = compteur - 1
            Plein.signal()          //On a enlevé un biscuit, on réveille une éventuelle biscuiteuse arrêtée
```

Exercice 3 4.5 /6.5

Q 3.1 Implémentation

L'objet partagé par les différentes machines est le tapis. C'est la classe Tapis qui gère la synchronisation des différents processus selon le modèle du moniteur de la question 2.3 : elle met en attente une biscuiteuse qui sollicite le tapis pour un dépôt ou une ramasseuse qui sollicite trop tôt le tapis pour un retrait.

La méthode Tapis.deposerBiscuit() est appelée par la classe Biscuiteuse lorsqu'un biscuit a été produit. Si le tapis est plein, l'objet appelant est mis en attente, sinon le biscuit est déposé.

La méthode Tapis.retirerBiscuit() est appelée par la classe Ramasseuse dès qu'elle a fini un empaquetage. Si le tapis compte moins de 10 biscuits, l'objet appelant est mis en attente, sinon la méthode vide le tapis

Lorsqu'une biscuiteuse réussit à déposer un gâteau ou lorsque la ramasseuse vide le tapis, un processus en attente est réactivé :

- La biscuiteuse qui pose le dernier biscuit réveille la ramasseuse. Si celle-ci n'était pas en sommeil (elle empaquete), elle videra le tapis dès qu'elle aura fini l'empaquetage en cours.
- Les biscuiteuses mises en sommeil seront réveillés en cascade (la 1ere par la ramasseuse quand elle aura vidé le tapis, la 2e par le 1ere quand elle aura posé son biscuit, etc...)

Threads ?

3.2
Le code fonctionne mais il aurait été plus propre d'utiliser deux moniteurs. Ici ce sont les buiscuiteuses qui se réveillent entre elles alors que cela aurait du être la ramasseuse.