Xavier Lambein
54621300

Projet 2 : Architecture

Victor Lecomte
65531300

LSINF1252

`TL;DR.` Nous avons choisi l'algorithme de divisions successives, sur lequel nous lançons plusieurs threads qui testent la division avec des valeurs de départ décalées.

Fonctionnement du programme

Notre programme se décompose en trois parties différentes, qui possèdent leur fonction propre : la boucle principale, les *readers* et les *factorizers*. Les *readers* sont des threads servant à lire des entiers depuis les entrées pour les placer dans une liste interne ; les *factorizers* sont des threads parcourant un entier pour en trouver les facteurs ; enfin, la boucle principale sert à synchroniser les *factorizers* pour changer l'entier courant à factoriser `to_fact` et éventuellement terminer le programme.

Ces parties partagent deux listes d'entiers qui sont la *waiting list* et la *prime list*. La première contient tous les nombres à factoriser. Elle possède deux entrées : les *readers*, évidemment, mais aussi les *factorizers*, comme nous allons le voir plus loin.

La seconde contient tous les facteurs premiers trouvés, et n'est utilisée que temporairement, tant qu'il reste des *readers* actifs. En effet, tant que tous les nombres n'ont pas été lus, il est impossible d'être certain que ces facteurs ne sont pas solution, et il est donc nécessaire de les garder. Cette liste ne possède qu'une seule entrée, et il s'agit de la boucle principale.

Chaque nombre stocké dans *waiting list* et dans la *prime list* transporte également deux informations : le fichier d'où il provient et son nombre d'occurrences, c'est-à-dire, s'il s'agit d'un facteur (premier ou non), le nombre de fois qu'il apparaît (divise un nombre). Cela est nécessaire pour vérifier si le facteur est une solution.

Dans la suite, nous allons expliquer en détail le fonctionnement de chaque partie.

Les *factorizers*

Tous les *factorizers* partagent une même variable globale `to_fact`, le nombre qu'ils sont en train de factoriser. Ceux-ci parcourent tous les entiers de 2 à $\sqrt{\texttt{to\_fact}}$, en vérifiant à chaque fois si l'entier en question divise `to_fact`. Si c'est le cas, un facteur (pas nécessairement premier) a été trouvé, et la variable globale `to_fact` est effectivement divisée par celui-ci, autant de fois que possible. Ensuite, le facteur est ajouté à la *waiting list*, en notant son nombre d'occurrences—ici le nombre de fois qu'il a divisé `to_fact`—, puis les *factorizers* continuent leur travail.

Bien évidemment, les différents threads ne parcourent pas chacun tous les nombres entre 2 à $\sqrt{\texttt{to\_fact}}$. Ceux-ci se répartissent le travail : le premier thread itérant sur $i = 2, 2+n, 2+2n, \dots$; le second thread itérant sur $i = 3, 3+n, 3+2n, \dots$; etc. Si ce n'était pas le cas, augmenter le nombre de threads n'aurait strictement aucun intérêt !

Lorsque tous les *factorizers* ont dépassé $\sqrt{\texttt{to\_fact}}$, ceux-ci attendent que la boucle principale choisisse un nouvel entier à placer dans `to_fact`, puis recommencent leur travail.

La boucle principale

Il est important de noter que la valeur de `to_fact`, une fois que les *factorizers* ont terminé leur travail, est un facteur premier du nombre initial.

Le rôle de la boucle principal est double : tout d'abord, elle va récupérer ce facteur premier et le traiter ; ensuite, si la solution n'a pas encore été trouvée, elle va choisir un nouvel entier de la *waiting list* et le placer dans la variable `to_fact` avant de relancer les *factorizers*.

Traitement du facteur Si celui-ci n'est pas 1, alors la boucle principale va parcourir chaque entier de la *waiting list* et diviser celui-ci par le facteur. Ensuite, deux situations sont possibles :

— les *readers* n'ont pas fini de lire les fichiers ; dans ce cas, le facteur est simplement ajouté à la *prime list*, en prenant soin de noter le nombre de fois qu'il a divisé un nombre de la *waiting list* dans son nombre d'occurrences ;

— les *readers* ont terminé leur lecture ; dans ce cas, si le facteur n'a divisé aucun nombre de la *waiting list*, il s'agit de la solution, et le programme termine.

Choix d'un nouvel entier Si la solution n'a pas été trouvée, un nouvel entier est choisi pour `to_fact` : il s'agit du plus petit entier de la *waiting list*, trouvé en faisant une simple recherche en $\mathcal{O}(n)$; en effet, la liste n'est pas ordonnée.

Les *readers*

Les *readers* sont des threads génériques qui lisent des entiers au format BigEndian depuis *file descriptor* et placent ceux-ci dans la waiting list. Leur fonctionnement exact est en vérité un peu plus complexe que cela.

Pour chaque nombre lu, un *reader* va parcourir la *prime list* et diviser le nombre par chaque facteur premier autant de fois que possible, en mettant à jour le nombre de fois que ce facteur a divisé. Cela est nécessaire car les nombres lus n'ont pas eu l'occasion d'être divisés par la boucle principale lors du traitement du facteur.

C'est là la seule raison d'existence de la *prime list*—et celle-ci est d'ailleurs détruite lorsque le dernier *reader* a terminé son travail. Avant de la désallouer, cependant, le dernier *reader* la parcours une dernière fois : en effet, si l'un des facteurs a un nombre d'occurrences de 1, alors il s'agit de la solution et le programme doit se terminer.

Mécanismes de synchronisation

Les *readers* et la boucle principale

Ces deux parties doivent être synchronisées à la manière d'un producteur-consommateur où il n'y aurait qu'un seul consommateur : la boucle principale. Celle-ci attend qu'un *reader* lise un entier et le place dans la *waiting list*, afin de pouvoir le mettre dans `to_fact`.

Nous avons donc un sémaphore `sem_full`, dont la valeur est égale au nombre d'entiers présents dans la *waiting list*. À chaque fois qu'un *reader* ou un *factorizer* ajoute un entier dans celle-ci, le sémaphore est `posté`, et à chaque fois que les *factorizers* ont besoin d'un nouvel entier à factoriser, la boucle principale effectue un `wait` sur ce sémaphore pour s'assurer qu'il peut en effet récupérer un entier de la *waiting list*.

La modification de l'état global

L'état global, constitué notamment de la *waiting list*, de la *prime list* et de `to_fact`, peut être sans risque accédé en lecture mais doit toujours être protégé par une unique mutex lors d'un accès en écriture. Il s'agit de la même mutex pour toutes ces données, car la modification de d'une partie de l'état global a systématiquement un impact sur le reste.

Commande des *factorizers*

Puisque la parallélisation de la factorisation est interne, il faut que les *factorizers* se synchronisent avec la boucle principale qui va choisir les nombres à factoriser un par un. La boucle principale va envoyer un signal aux factorizers pour qu'ils commencent à factoriser, ce qu'ils vont faire, puis ils vont attendre qu'eux tous aient fini avant d'envoyer un signal à la boucle principale et d'attendre le prochain nombre à factoriser.

La solution que nous avons développée est une extension du problème du rendez-vous, et se présente ainsi (son implémentation se trouve dans le fichier `factorizer.c`) :

```
1  while true:
2      wait start
3      post start
4
5      factorisation...
6
7      lock mutex
8          num_waiting++
9          if num_waiting == num_factorizers:
10             wait start
11             post handshake
12     unlock mutex
13
14     wait handshake
15     num_waiting--
16     if num_waiting == 0:
17         post finish
18     else:
19         post handshake
```

On retrouve tout à fait ci-dessus l'algorithme classique du problème du rendez-vous, géré par la mutex `mutex` et le sémaphore `handshake`, mais des ajouts ont été faits pour gérér l'échange entre la boucle principale et les *factorizers*. Nous allons commenter les lignes que nous avons ajoutées :

— Lignes 2 et 3 : avant de commencer à factoriser, le *factorizer* attend le message envoyé par la boucle principale via le sémaphore `start` et le transmet au *factorizer* suivant. De cette manière, la boucle principale ne doit envoyer ce message qu'une seule fois.

— Ligne 10 : quand tous les *factorizers* sont arrivés au bout de leur factorisation, on en profite pour décrémenter le sémaphore `start` que le dernier *factorizer* à entrer avait posté en ligne 3 sans que personne ne l'attende. Il est important que cette opération se fasse à cet endroit précis pour éviter qu'un factorizer ne soit libéré par l'incrémentation du sémaphore `handshake` et ne commence sans attendre le signal de la boucle principale.

— Lignes 15 à 17 : tous les *factorizers* passant la barrière vont avoir le même comportement que pour le problème de rendez-vous classique sauf le dernier qui à la place d'incrémenter `handshake` va incrémenter `finish` pour signaler à la boucle principale que tous les *factorizers* ont fini et sont prêts à recevoir un nouveau incrément de `start` en ligne 2.

Il suffit ensuite à la boucle principale de poster `start` puis d'attendre `finish` dès qu'elle veut lancer les threads sur un nombre.

Trucs vieux :D

Déroulement

Nous commençons par lire tous les entiers à factoriser et nous les plaçons dans un heap (dans l'ordre inversé, avec le plus petit au-dessus). Ensuite, tant que le heap n'est pas vide, nous enlevons le plus petit entier du heap et nous lançons dessus les n threads à notre disposition. Dès qu'ils trouvent un facteur, ils l'ajoutent au heap et divisent la valeur à factoriser.

Une fois que les n threads ont tous atteint la racine, cela signifie que le nombre restant est premier. Nous essayons de diviser chaque nombre dans le heap par ce facteur premier. Si aucun n'est divisible, ce nombre est la solution. Sinon nous continuons.

Voici le pseudocode pour le thread principal :

```
launch readers

while not found:

    wait for sem_full

    lock mut_state
        toFact = minimum from waiting list
    unlock mut_state

    for i in 1..n:
        init thread i with (start=1+i, step=n)
    end

    for i in 1..n:
        join thread i
    end

    // toFact is a prime factor

    lock mut_state
        divide everything in the waiting list by toFact as much as
           possible
        remember the number of divisions

        if numReader == 0:
            if occurrences == 0:
                print toFact
                found = true
        else:
            add toFact to prime list with number of occurrences
    unlock mut_state
end

free ALL THE THINGS
```

Remarquons que si la boucle `while` se finit, l'input est incorrect, car cela signifie qu'aucun facteur premier ne divise exactement un nombre exactement une fois.

Les n threads recherchent des facteurs du nombre courant `toFact` en parcourant tous les entiers de 2 à $\sqrt{\texttt{toFact}}$, avec un pas de n : le premier thread itérant sur $i = 2, 2+n, 2+2n, \ldots$; le second thread itérant sur $i = 3, 3+n, 3+2n, \ldots$; etc. Pour chaque i, le programme regarde s'il divise `toFact`. Dans ce cas, il s'agit d'un facteur (pas nécessairement premier), et la procédure décrite ci-dessus est appliquée.

Voici le pseudocode pour les threads de recherche de facteurs :

```
1  for i = start, i*i <= toFact, i += step:
2      if i divides toFact:
3
4          lock mut_toFact
5              if i divides toFact:
6
7                  lock mut_state
8                      add i to waiting list
9                  unlock mut_state
10
11                 post sem_full
12                 divide toFact by i
13             end
14         unlock mut_toFact
15
16     end
17 end
```

Synchronisation des threads

Notre programme ne présente qu'une seule section critique à proprement parler : il s'agit du moment où un thread a trouvé un facteur de `toFact`. Celui-ci bloque alors une mutex qui empêche l'écriture sur la variable partagée `toFact`, ce qui permet au thread de diviser celle-ci par le facteur en toute sécurité. À l'intérieur de la section protégée par la mutex, le programme vérifie d'abord à nouveau si le facteur trouvé divise bel et bien `toFact`, pour être certain que la valeur de celui-ci n'aurait pas changée avant que la mutex ne soit bloquée.

Une deuxième zone de synchronisation apparaît lorsque tous les threads ont terminé de parcourir les entiers entre 2 et $\sqrt{\texttt{toFact}}$. Le thread principal est alors chargé de vérifier si une solution a été trouvée, et sinon de continuer l'algorithme avec le nombre suivant présent sur le heap. Il fait cela de manière très simple, en tuant les n threads, effectuant les vérifcations, puis éventuellement en relançant n nouveaux threads.