IF223 - Résumé des définitions

2021 - 2022

—

Modèles

Mémoire partagée (shared memory)

Définition

Dans ce modèle, les processeurs ne communiquent pas directement. Ils s'échangent des informations grâce à une mémoire commune.

Dans ce modèle, on néglige le coût des communications.

Application

Problème du consensus (cas basique): Les processeurs ont chacun en entrée une valeur, 0 ou 1. On élit un leader, les autres processeurs devront s'aligner sur sa valeur. La difficulté vient du fait que les processeurs peuvent tomber facilement en panne. Il faut donc anticiper les pannes.

Passage de messages (message passing)

Définition

Ce modèle traite explicitement des communications. Lorsqu'un processeur veut communiquer avec un autre, il doit lui envoyer un message qui transite via un médium de communications, grâce à des instructions du type **SEND** et **RECEIVE**

Dans ce modèle, c'est surtout l'efficacité des échanges des messages que l'on étudie, on néglige le coût des calculs locaux des processeurs.

Réseau de diffusion (broadcast/radio networks)

Définition

Chaque processeur peut communiquer le même message simultanément à un certain nombre de récepteurs, comme un émetteur radio le ferait.

Spécificités du calcul distribué

Coût des communications

Les communications ne sont pas gratuites. A 3km, on peut supposer que les communications coûtent 10000 fois plus que les calculs locaux.

Connaissance partielle

En distribué, le processeur P_i ne contrôle que son état local. Les données réparties sur d'autres processeurs changent sans aucun contrôle direct de P_i .

Parfois, cette connaissance comprend la topologie du graphe (taille, connexe ou non, complet, ..)

Erreurs pannes et défaillances

En séquentiel: si un élément est défectueux (mémoire, processeur, disque-dur,...) on le change et on continue ou on recommence le calcul.

En distribué, en absence de tout contrôle centralisé, la détection même d'une défaillance peut être difficile. Un lien défectueux peut causer de graves problèmes (omission, ajout ou corruption de messages).

C'est le concept de **tolérance aux pannes** où, même avec un lien ou processeur en panne, le programme peut encore fonctionner (peut-être avec des performances moindres).

Synchronie

La synchronie est un élément impactant fortement le fonctionnement d'un système distribué. On considère deux fonctionnements extrêmes: synchrone ou asynchrone.

Mode synchrone

Définition

Les tops de l'horloge interne (donc locale) de chacun des processeurs vérifient la propriété suivante: un message envoyé au top t de P_i vers son voisin P_j est reçu avant le top t + 1 de P_j , et ne peut pas dépendre de message envoyé à un top >= t .

On parle souvent de **ronde** pour désigner la période de temps séparant deux tops horloges consécutifs. Donc la propriété est que les messages émis à une **ronde** donnée sont reçus par leurs voisins pendant la même **ronde**. De plus, il n'est pas possible de répondre à un message reçu lors de la même **ronde**. La réponse devra être émise à une **ronde** ultérieure.

Programme en mode synchrone

L'exécution de chaque programme est alors dirigée par les tops horloges ou les rondes. Et entre chaque top successif, on exécute un certain cycle d'instructions qu'on appelle aussi **ronde.**

P_i :
au top 1, faire $ronde_1$
au top 2, faire $ronde_2$
....

où $ronde_t$ est un cycle **SEND/RECEIVE/COMPUTE.** Plus précisément, au top *t* :

$ronde_i$:
1. **SEND** à (zéro, un ou plusieurs de ses voisins)
2. **RECEIVE** de (zéro, un ou plusieurs voisins)
3. calculs locaux sur les messages reçus

Pour résumer, lors de chaque **ronde**, un sommet échange des messages avec un ou plusieurs de ses voisins (éventuellement aucun), le message émis pouvant être **différent** selon le voisin, puis effectue un calcul local avec les messages éventuellement reçus lors de cette même ronde.

Mode asynchrone

Définition

Les tops horloges de processeurs voisins n'ont plus aucune propriété de synchronisation si bien que leur utilisation n'est plus d'aucun recours. Les messages envoyés de P_i vers un voisin P_j arrivent un temps **fini** mais **imprédictible.**

Une façon unifiée est aussi de dire qu'en mode synchrone les événements sont déclenchés par les **tops horloges**, alors qu'en mode asynchrone ils le sont par la réception des messages.

Non déterminisme

Un algorithme déterministe a la propriété de fournir toujours le même résultat pour une même entrée. L'exécution est donc toujours la même si les entrées sont les mêmes.

Le résultat est ainsi entièrement déterminé par les entrées et l'algorithme.

En **séquentiel**, le non déterminisme est lié à la présence d'instructions comme **random()** dont la valeur varie à chaque nouvelle exécution.

En **distribué**, le résultat d'une exécution peut déprendre aussi du système sur lequel s'exécute l'algorithme, même en l'absence d'instruction comme **random()**.

En effet, la vitesse des messages circulant sur les liens n'est pas contrôlable par l'algorithme.

Mesures de complexité

En séquentiel, on utilise surtout la complexité en temps et moins souvent la complexité en espace.

En distribué les notions de complexité sont plus subtiles.

On utilisera essentiellement les notions de **nombre de messages échangés** ou de **volume de communication**. Ces complexités dépendent, le plus souvent, du graphe sur lequel on exécute l'algorithme.

Définition (temps synchrone)

La complexité en temps d'un algorithme distribué A sur un graphe G en mode synchrone, noté **Temps(A, G)**, est le nombre de **tops horloges** générés durant l'exécution de A sur G dans le pire des cas (c'est-à-dire sur toutes les entrées valides de A).

En synchrone, on parle aussi du nombre de **rondes**, puisqu'un algorithme synchrone est une succession de rondes **SEND/RECEIVE/COMPUTE.**

Définition (temps asynchrone)

La complexité en temps d'un algorithme distribué A sur un graphe G en mode asynchrone, noté **Temps(A, G)**, est le nombre d'unité de temps durant l'exécution de A sur G dans le pire des cas (c'est-à-dire sur toutes les entrées valides de A et tous les scénarios possibles), chaque message prenant un temps au plus unitaire pour traverser une arête.

Définition (nombre de messages)

La complexité en **nombre de messages** d'un algorithme distribué A sur G, noté **Message(A, G)**, est le nombre de messages échangés durant l'exécution de A sur G dans le pire des cas (sur toutes les entrées valides de A et tous les scénarios possibles en mode asynchrone).

Modèles représentatifs

Parmi les nombreux modèles qui existent dans la littérature, on en distingue trois qui correspondent à des ensembles d'hypothèses différentes. Chaque ensemble d'hypothèses permet d'étudier un aspect particulier du calcul distribué. Un modèle **n'a pas vocation à épouser au mieux la réalité.** c'est une **abstraction** permettant de **prédire** le résultat d'expériences bien réelles.

Les hypothèses suivantes sont communes aux trois modèles:

1. **Pas d'erreur**: il n'y a pas de crash lien ou processeur. Un message envoyé finit toujours par arriver.
2. **Calculs locaux gratuits**: seules les communications sont prises en compte
3. **Identité unique:** les processeurs ont une identité codée par un entier de $O(\log n)$ bits, n étant le nombre de sommets du graphe. Le réseau n'est pas anonyme.

Les trois modèles sont:

LOCAL

Il est utilisé pour étudier la nature locale d'un problème.

- mode synchrone, tous les processeurs démarrent le calcul au même top;
- message de taille illimité.

CONGEST

Il est utilisé pour étudier l'effet du volume des communications.

- mode synchrone, tous les processeurs démarrent le calcul au même top;
- message de taille limité à $O(\log n)$ bits.

ASYNC

Il est utilisé pour étudier l'effet de l'asynchronisme.

- mode asynchrone.

Exemple d'algorithme dans le modèle LOCAL

Problème: La racine d'un arbre de profondeur deux souhaite collecter les identifiants de toutes les feuilles.

Dans le modèle LOCAL, une solution en deux rondes pourrait être comme ci-dessous, ne supposant que chaque sommet connaisse son père:

Algorithme Collecter-ID-feuilles
(code du sommet u)

1. $S := \{\text{ID}(u)\}$
2. Pour $i \in \{1, 2\}$ faire :
   (a) NewPulse
   (b) Send$(S, \text{Père})$
   (c) $S := \bigcup_{v \in N(u) \setminus \text{Père}(u)} \text{Receive}(v)$

Diffusion

Il s'agit de transmettre un message M à tous les sommets d'un graphe connexe à n sommets depuis un sommet distingué que l'on notera r_0.

Propriété

Tout algorithme distribué de diffusion A sur G vérifie, en mode synchrone ou asynschrone et quelle que soit la connaissance à priori des sommet sur G:

- MESSAGE(A,G) >= n - 1
- TEMPS(A, G) >= $ecc_G(r_0)$

Preuve

Il faut informer n - 1 sommets distincts du graphe. L'envoi d'un message sur une arête ne peut informer qu'un seul sommet (celui situé à l'extrémité de l'arête). Donc il faut que le message circule sur au moins n - 1 arêtes différentes de *G* pour résoudre le problème. Donc Message(A, G) >= n - 1.

Arbre de diffusion

Une stratégie courante pour réaliser une diffusion est d'utiliser un arbre couvrant enraciné en r_0, que l'on appelle **T** . L'intérêt est qu'on ne diffuse le message M que sur les arêtes de **T**. On va supposer que chaque sommet *u* de G connait **T** par la donnée de son père Père(u) et de l'ensemble de ses fils Fils(u). L'algorithme ci-dessous dépend donc de l'arbre **T** et de sa racine r_0.

Algorithme $\textsc{Cast}_T(r_0)$
(code du sommet u)

1. Si $u \neq r_0$, alors $M :=$ RECEIVE()
2. Pour tout $v \in$ FILS(u), SEND(M, v)

Propriété

Pour tout graphe G et racine r_0 en mode synchrone ou asynchrone:

- MESSAGE($Cast_T(r_0)$, G) = $|E(T)|$ = n - 1
- TEMPS($Cast(r_0)$, G) = $ecc_T(r_0)$ = hauteur(T)

Inondation

On utilise cette technique (*flooding* en anglais) en l'absence de structure pré-calculée comme un arbre de diffusion. Un sommet *u* ne connaît ni Père(u) ni Fils(u).

Algorithme Flood(r_0)
(code du sommet u)

1. Si $u = r_0$, alors SEND(M, v) pour tout $v \in N(u)$.
2. Sinon,
   (a) $M :=$ RECEIVE() et v le voisin qui a envoyé M
   (b) SEND(M, w) pour tout voisin $w \neq v$ de u.

Propriété

Pour tout graphe G et racine r_0 en mode synchrone ou asynchrone:

- MESSAGE(Flood(r_0), G) <= 2m - n + 1
- TEMPS(Flood(r_0), G) <= $ecc_G(r_0)$ + 1

Concentration

La **concentration** est un problème similaire à celui de la diffusion.

Il s'agit pour chaque sommet u du graphe d'envoyer un message personnel M_u vers un sommet distingué r_0. C'est en quelque sorte le problème inverse, sauf que les messages ne sont pas tous identiques comme dans le cas d'une diffusion.

Voici une solution, lorsqu'un arbre T de racine r0 est disponible, c'est-à-dire connu de chaque sommet u par les variables Père(u) et Fils(u).

Algorithme Converge$_T(r_0)$
(code du sommet u)

1. Poser $c := |\textsc{Fils}(u)|$.
2. Si $c = 0$ et $u \neq r_0$, alors $\textsc{Send}(M_u, \textsc{Père}(u))$.
3. Tant que $c > 0$:
   (a) $M := \textsc{Receive}()$ et v le voisin qui a envoyé M.
   (b) $c := c - 1$.
   (c) Si $u \neq r_0$, $\textsc{Send}(M, \textsc{Père}(u))$.

Propriété

Il existe un algorithme distribué *Converge* qui réalise pour tout graphe G et sommet r_0 la concentration vers r_0 avec:

- en mode synchrone:
  - MESSAGE(Converge, G) = *O*(m)
  - TEMPS(Converge, G) = *O*(D)
- en mode asynchrone:
  - MESSAGE(Converge, G) = *O*(m)
  - TEMPS(Converge, G) = *O*(n)

Algorithme de Spanning Tree

1. racine envoie M à tous ses voisins
2. Quand un sommet qui n'est **pas** la racine reçoit en **premier** M
   a. définir l'envoyer comme son parent
   b. envoyer le message "parent" à l'envoyeur
   c. envoyer M à tout les autres voisins (s'il y en a sinon on s'arête)
3. Sinon lorsqu'il reçoit M
   a. On envoie "déjà reçu" à l'envoyeur

Note: il est possible qu'un processeur reçoit M pour la première fois de plusieurs processeurs en même temps. Dans ce cas, en choisir un et exécuter l'étape 1. Puis éxécuter 2 pour les autres messages.

Chaque processeur p_i gère trois variables locales:

1. *parent* est utilisé pour stocker le processeur qui sera le parent de p_i dans le spanning tree.
2. *childen* est utilisé pour stocker les processus qui seront les enfants de p_i dans le spanning tree.
3. *other* est utilisé pour enregistrer les autres processeurs.