Programmation Avancée des Architecture Multicoeurs

Gaël Thomas
gael.thomas@telecom-sudparis.eu
http://www-inf.telecom-sudparis.eu/COURS/chps/paam/

Table des matières

Technique d'optimisation bas niveau (ASM) sur des architectures de "petits multicolores" (environ 30 cœurs). Pas d'OpenMP, pas de MPI (trop haut niveau).

Référence *The Art of parallel programming*, M.Herlihy, N.Shavit (Bible, à acheter !)

Attention Les transparents n'ont pas été modifiés depuis 7 ans : ne pas trop s'y fier (50% d'information données ne sont plus dans les slides en fait).

1 Introduction

Depuis une dizaine d'année, les appareils informatiques ont complètement changé (bepuis les tours laides en plastiques...). D'un côté l'informatique embarqué (téléphones, voitures, trains, ...) s'est développée, avec peu de ressources de calcul → modèle du *mainframe* : serveur/terminaux. Les centres de données sont désormais propulsés par des architectures multicoeurs (le dégagement thermique augmente avec le carré de la fréquence, on augmente donc le nombre de coeurs car l'augmentation est linéaire avec la consommation). La loi de Moore restant toujours possible (miniaturisation des transistors), on intègre des clusters de machine complet sur un die.

De nos jours, pour gagner des performances, il faut augmenter le parallélisme, ce qui est difficile. Nous allons bien entendu travailler dessus, ainsi que sur le placement (caches).

On a un aspect de plus en plus nomade de l'informatique (on masque la localisation : possibilité de continuer un travail sur plusieurs machine), rendu possible par la centralisation des calcul. La technique utilisée est la *virtualisation*, ce qui est également possible pour le HPC. On n'utilise quasiment plus la machine telle qu'elle est. L'idée est d'utiliser une "machine dans une machine", ce qui introduit une *couche de virtualisation* qui modifie les comportement [1].

Plan

I Etude des multicoeurs (NUMA) → TP très sympa pour mesurer la latence mémoire

II Concurrence en mémoire partagée

III Algorithme classique et sans verrous

IV Mémoire transactionnelle

Rappel sur les verrous (mutex)

Les threads lisent et écrivent des valeurs dans un *registre* en mémoire, par exemple une base de donnée d'une banque lue et écrit par des distributeurs. Certains registres sont en lecture, lecture/écriture, écriture. On s'intéresse ici aux registres en lecture/écriture.

Le problème réside dans le fait qu'il est possile d'avoir des accès *concurrents* qui mènent à des données *incohérentes* (du point de vue du programmeur !). C'est le problème du *chat-mouton* (essayer de dessiner un chat et un mouton sur le même tableau par deux enfants, ça ne va pas trop marcher...).

Problème classique des bases de données sur lequel est basé tout le cours : Les comptes en banque. Un processus p_1 lit le compte et le met à jour en ajoutant deux euros. Un ami, processus p_2 est avec votre carte bleue et débite 100€(lire le compte puis le met à jour). Il est possible de miraculeusement gagner 100€ :

p_1 : a lit le contenu ($cpt = 100$€)
p_1 : ajoute 2 à a ($cpt = 100$€, $a = 102$€)
p_2 : b lit le compte ($cpt = 100$€)
p_2 : retirer 100 à b ($cpt = 100$€)
p_2 : b devient le nouveau solde ($cpt = 0$€)

1. il s'agit du sujet de recherche de notre professeur

p_1 : a devient le nouveau solde ($cpt = 102€$)

Le retrait a disparu ! Pour palier à ce fait, on utilise un verrou pour retirer l'accès concurrent. Les régions dangereuses sont appelées *sections critiques* et doivent être exécutés *atomiquement*. Le verrou joue le rôle de feu de signalisation pour les programmes : tant que le verrou n'est pas libre j'attends, si il est là je le prends et je suis le seul à pouvoir le prendre.

On entoure les section critiques par le verrou :

```
1 lock(mutex) ;                        // bloquant si jamais le mutex n'est pas disponible
  /* section critique 1;
  Par exemple ajouter 2€;
                                                                                    */
2 unlock(mutex);
```

```
1 lock(mutex) ;                        // bloquant si jamais le mutex n'est pas disponible
  /* section critique 2;
  Par exemple retirer 100€;
                                                                                    */
2 unlock(mutex);
```

Attention aux situations de famines ! Il ne faut pas qu'une suite d'opérations menant à un interblocquage des threads soit possible ! Il faut éviter d'inverser l'ordre des verrous (dans la doc Linux par exemple, il est indiqué dans quel sens prendre les verrous !).

Petit aparté Attention, cela revient rapidement costaud niveau math derrière, le parti pris ici est de retirer le maximum de la théorie pour passer au code.

Notation Un exam final sur 100% et un devoir surprise à un moment sur 10%. Une annale sont disponibles sur le site (attention, ne pas se fier à la difficulté car elle dépend de la promo).

Conseil Garder un œil sur les revues scientifique, même si vous vous destinez à un parcours d'ingénieur (grrrr... [2]).

2 Architectures des Machines Multicoeurs

Idée : comprendre au niveau matériel et pourquoi certains comportement au niveau logiciel peuvent sembler étranges.

Auparavant Loi de Moore : la densité d'intégration des transistors double tout les 18 mois. Cette puissance permettait d'augmenter la fréquence, qui s'est heurtée à la dissipation thermique (la consommation augmente avec le carré de la fréquence). On sait par contre que ça ne sera pas toujours le cas : les interconnexions entre coeurs peuvent également tirer beaucoup d'énergie, bien que ce ne soit pas le facteur limitant de nos jours.

De nos jours, il est possible de louer des serveurs vers 80 cœurs par Amazon : c'est le *cloud computing*.

2. Humour : L'auteur de ces notes ne se destine pas à ce parcours, ne vous vexez pas

Premier grand problème : Il faut paralléliser le code ! Et ce n'est pas suffisant.

Au niveau architecturale, il est impossible d'utiliser une topologie classique à base de bus, car environ une instruction sur quatre est un accès mémoire. Pour palier à cela, on change la topologie : *diviser pour mieux régner*. On utilise un bus pour relier plusieurs cœurs entre eux et former une *nœuds* de calcul, puis on les relies entre eux.

Terminologie

Unité de calcul : capable de faire des additions/opération (on appelait cela processeur auparavant

Unité mémoire : Contrôleur permettant la traduction adressage virtuelle/physique

Contrôleur d'interruption : Gère les interruptions (principalement par les périphériques

Un timer : Indépendant par cœur (imaginez si une seule horloge à 800 kHz devait communiquer chaque tick au 50 cœurs...). Lors de l'utilisation de l'instruction `rtsc` (read timestamp counter), ceux-ci sont légèrement différents entre cœurs.

Un cache (donnée et code)

Le TLB (translation lookout buffer) : Stocke les mapping mémoire virtuelle/physique récents

L'ensemble de ces éléments est appelé un *tuile* ou *cluster* qui sont intégrés sur des *die*. Ce dernier correspond à un circuit intégré unique (souvent, il s'agit d'un cluster). Une *socket* est l'objet physiquement déplaçable contenant un ou plusieurs dies. Attention, cela n'est pas lié au NUMA, qui est lui une construction logicielle (bien que proche).

Le *Network On Chip* est la partie qui connecte les différent coeurs d'un die. Ce qui connecte les dies ensemble est appelé *Inter-connect*.

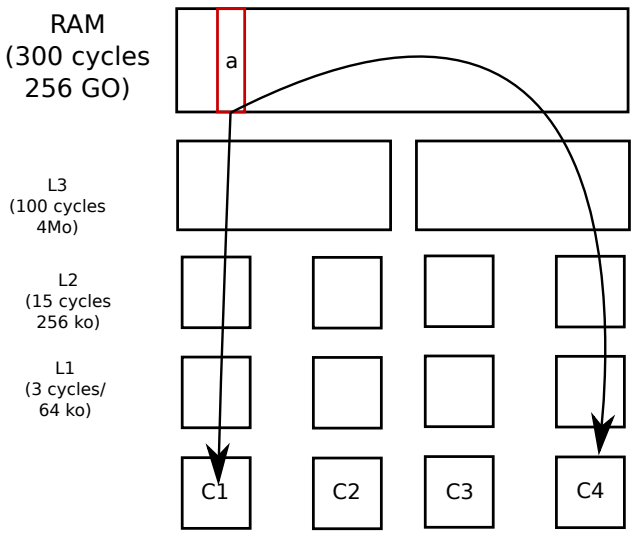

FIGURE 1 – Hiérarchie des caches

2.1 Problème du cache

Il assure la cohérence des données, donne un sens à ce que le processeur voit de la mémoire (cf figure 1). Comment invalider une ligne de cache en cas de modification ?

On utilise le processus MOESI : un verrou contenant un état par ligne de cache[3], la synchronisation s'effectuant entre cache de même hiérarchie :

E : Excusive (Écriture)

3. c'est ce problème qui est à l'origine du NUMA

I : Invalid (A été modifiée / plus présente)
S : Shared (Lecture)
Il faut cependant garder l'information "la valeur en cache est-elle plus récente que celle en RAM ?". Pour cela, on dédouble tous les états :
M : Modified, E mais incohérente avec la mémoire centrale
O : Owned, S mais incohérente avec la mémoire centrale
L'état I n'est pas répliqué puisque la ligne de cache n'a plus aucun sens.

Pour récapituler :
État M ou E : verrou en écriture
État O ou S : verrou en lecture
État I : pas de verrou

Où est le soucis ? Avant, on faisait du *snooping* (espionnage) : tout passait par le bus. Le soucis est que le snooping ne scale pas car il demande un *broadcast* (demander à toutes les autres cœurs).

Solution : On segmente la mémoire en centralisant l'accès à certaines lignes de caches qui sont gérés par certains clusters spécifiques. Des tables associent les adresses physiques aux cœurs qui cachent la ligne : ainsi les invalidation seront spécifiquement envoyés au bon processeur. Autre soucis : où mettre la table ? La segmentation se fait en divisant la mémoire par le nombre de cluster et en répartissant équitablement celle ci entre les cluster (noeud 1 $\rightarrow$ mémoire de 0 à 2 Go, noeud 2 $\rightarrow$ 2 à 4 Go, etc). Le cluster est alors un *nœud NUMA*, qui correspond souvent à un die (bien que ce soit trois dénominations différentes). Le routage s'effectue par chaque nœud NUMA qui connait la table (stockée dans le L3 et gérée par un petit contrôleur), spécifiée pas le BIOS au démarrage.

Note : Cette segmentation est nécessaire mais relativement peu utilisée (bcp de variables locales). Dans le meilleur des cas (pas de saturation), une lecture va couter environ 500 cycle. Dans le cas saturé, environ 2000 cycles !

Conséquences : L'accès à la mémoire centrale n'est plus uniforme (local : 200 cycles ; au pire, 400 cycles) ; les liens peuvent saturer (et là, 2000 cycles !). La partitionnement de la mémoire change la latence !

Principe/Solution : augmenter au maximum la localité quand on développe une appli. Par défaut, la mémoire virtuelle est mappée sur différents nœuds NUMA selon la politique *first touch* : si la mémoire n'est pas encore mappée, on lui donne de la mémoire à partir *de son propre noeud*. Dans le cas de gros malloc, le mappage se fait *au niveau des premiers accès* et non au niveau de la réservation.

Il existe également l'*interleaved* qui utilise un *round robin* (allocation aléatoirement de la mémoire), ce qui permet d'équilibrer les accès mais est très mauvais pour la localité (pas de saturation interne).

2.2 Placement de processus sous Linux

`pthread_setaffinity` pour sélectionner le cœur, `mbind` pour n'allouer la mémoire physique qu'à partir d'un ensemble de nœuds. `madvise` permet de libérer les pages physiques pour les migrer. Pour plus de détails, voir le manuel linux (`man fonction`).

3 Algorithmique sans verrou

Le principe même du verrou induit des threads inactifs lorsque ce dernier est pris (verouillé). En fait, plus il y a de verrous, moins il y a de parallélisme ! Le verrou force la séquencialisation de la section critique.

Le principe des *structures non bloquantes* est née dans les années 80, puis a connu un nouvel essor en 200 avec la liste chainée sans verrou.

Definition 1 (Loi d'Ambdahl). *Pour tout programme, si p est la proportion de code exécutable en parallèle, $\frac{p}{n}$ la répartition de la charge de travaille sur n cœurs, alors l'accélération maximal théorique est*

$$a = \frac{1}{1 - p + \frac{p}{n}} \xrightarrow[n \to +\infty]{} \frac{1}{1-p}$$

Cela signifie par exemple que pour une un programme parallélisé à 75%, l'accélération maximale sera de 4 (3,7 à 32 cœurs). Pour un taux de parallélisation de 95%, le speedup maximal est de 20, et atteint 100 pour un code à 99% parallélisé.

Note La formule donne également une idée du nombre de cœurs "utiles". Par exemple, pour un code à 95% parallèle, le speedup passe de 12,55 à 32 cœurs à 17,42 à 128 cœurs).

Conclusion Cela vaut le coup de se battre *pour quelques pourcents de parallélisation*, même pour une addition seule !

Exemple

- Une addition (`x++`) sur la suite SPARC 2) permet de multiplier les performances par 10 sur 50 cœurs !
- Le compteur d'ouverture des fichiers était le goulot d'étranglement de Linux sur le HPC (meilleur papier système 2013)

Comment se passer du code séquentiel ? Retirer la mémoire centrale. C'est cependant impossible, même sur MPI, les `gather`/`scatter` cachent une synchronisation dans la répartition des résultats.

Dans les grands centres web par exemple, tout est fait avec des processus (communication par TCP/IP au lieu de la mémoire partagée).

3.1 Solution naturelle

Qu'est-ce qu'un algorithm non bloquant ?

Wait-free : Toute opération se termine en un nombre fini de pas (i.e. pas de blocage de threads ni de famine). Cela est peu utile en HPC car il contient beaucoup de code séquentiel (et donc non parallélisable) ; mais est très recherché dans le domaine du temps réel.

Lock-free : Si des opérations sont appelée infiniment souvent, alors elles se terminent infiniment souvent (ceci est une garantie *asymptotique* donc pas de garantie temporelle, pas de bornes). → pas d'interblocage du programme, mais certains threads peuvent ne jamais être élus).

Obstruction-free : A tout moment, tout thread s'exécutant seul termine son opération en un nombre fini de pas (i.e. pour tout temps t je peux couper tous les threads sauf un et terminer l'exécution de ce threads en un temps fini).

Noter que

$$\text{Wait-free} \Rightarrow \text{Lock-free} \Rightarrow \text{Obstruction-free}$$

Les algorithmes à verrou ne sont pas obstrution-free, car une fois le verrou pris, tous les autres threads sont bloqués.

Du point de vue des adversaires :

Wait-free : On borne le temps maximal perdu par l'adversaire

Lock-free : L'adversaire peut retarder de manière arbitrairement grande mais pas infiniment souvent

Obstruction-free : Pas d'adversaire vu que l'on coupe les autres threads

3.2 Structures non bloquantes

3.2.1 Les outils

On redescend au niveau architectural : les outils sont fournis par le processeur :

— `old-value` ← `atomic-add(variable, value)` *sans verrou* (il s'agit d'un verrou sur une ligne mémoire pendant le temps de l'opération → temps d'exécution borné)

— `old-value` ← `atomic-swap(variable, value)` écrit un valeur en mémoire et renvoie son ancienne valeur

— `old-value` ← `atomic-CAS(variable, test, new)` : atomic compare and swap : compare et échange la variable si et seulement si elle vaut `test`. Dans tous les cas, la valeur initiale de la variable est retournée.

D'un point de vu théorique, $atomic - CAS$ à la puissance du consensus (si on vote sur la valeur, à la fin tout le monde à voté sur une valeur proposé et tout le monde est au courant du résultat). Sur le plan logique pure, `atomic-add` et `atomic-CAS` sont équivalents, `atomic-swap` est plus faible.

Exemple `add` atomic en obstruction-free en utilisant `atomic-CAS` :

```
1 void add (int* x, int a)
2 |  int old, new;
3 |  do
4 |  |  old = *x;
5 |  |  new = old + a;
6 |  while atomic-CAS(x, old, new);
```

Locking mechanism in Linux : ticket-lock

```
1 int ticket;
2 int billet;
3 void lock()
4 |  int billet = atomic-add(&ticket,1);
5 |  while guichet<billet do
6 |  |  wait();
7 void unlock()
8 |  atomic-add(&guichet,1);
```

Noter que `atomic-CAS` est un lock en lui-même, appelé `spinlock`.

```
1 void lock()
2 |  while CAS(&l, 0, 1) == 1 do
3 |  |  wait();
4 void unlock()
5 |  l=0;
```

Attention Ne JAMAIS l'utiliser : l'attente est *active* (besoin de lecture *et écriture*), ce qui sature le bus !

3.2.2 Modèles mémoire

Il définit les ordonnancements possibles des accès mémoires. Un exemple de problème :

```
1 send
2 |  msg = "coucou";
3 |  recu = true;
4 recv
5 |  while recu==false do
6 |  |  printf(msg);
```

A cause des écriture différés du processeur, il est pourrait être possible que la valeur lue par `printf` ne soient pas encore écrite en mémoire.

Chaque processeur donne des garanties spécifiques. Un principe commun est qu'en local ($\Rightarrow$ pour le même thread), toute lecture succédant à un écriture lit la dernière valeur écrite.

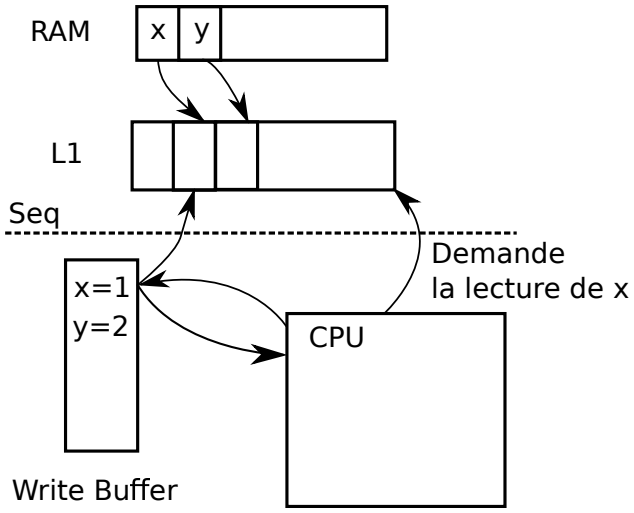

FIGURE 2 – Principe du Write Buffer

De manière similaire, il existe également un Read Buffer de principe similaire au Write Buffer (mais implémenté à des endroit différents).

Information Chaque écriture du cache est précédé d'une lecture, car le cache est accessible ligne par ligne, il faut donc conserver les anciennes données.

Le modèle du X86 est le *TSO : Total Store Order*
— Les lectures ne sont jamais réordonnancées entre elles
— Les écritures non plus
— Une écriture après une lecture n'est pas réordonnancée
— Une lecture après un écriture peut être réordonnancée s'il s'agit de deux cases mémoires distinctes

Si jamais un des ces cas devait arriver, le buffer responsable de la seconde opération consulte l'autre buffer, et on attend que la première opération soit propagée en mémoire (ce qui est un trou à performances, mais nécessaire pour la programmation.

Attention à la lecture après écriture dans deux cases mémoires distinctes Bien que en pratique ça ne soit pas dérangeant. Initialement : $t_1 = t_2 = 0$.
P_1 : écrit x, 2
$t_1 =$ lit y

P_2 : écrit y, 1
écrit x, 1
P_3 : t_2 = lit x
En cohérence séquentielle (sans réordonnancement), il est impossible d'avoir $x, t_1, t_2 = 2, 0, 1$. En effet $x = 2$ s'effectue avant $x = 1$, et la lecture de x s'effectue entre ces deux opérations. Or cela est possible en TSO (la lecture de t_1 peut être effectuée en tout premier lieu).

Les instructions autorisées sont `lock xadd`, `xchg` et `lock xchg`.

Pour éviter le seul réordonnancement possible, `mfence` permet une barrière mémoire, qui vide les buffers lecure et écriture. *Attention*, en C, il faut donc en mettre car on ne connait pas la plateforme d'exécution. Le préfixe d'instruction `lock` intègre automatiquement un `mfence` ainsi que la visibilité immédiate de l'écriture après l'instruction préfixée.

Il n'y a pas de garantie de réordonnancement en C avant le C11. Cependant il existe des instruction atomique pour les compilateurs, tels `__sync_fetch_and_add`, `__sync_lock_test_and_set` (pour `xchg`, vraiment bizarre) `__sync_val_compare_and_swap`, et la barrière mémoire `__sync_synchronisze`. Sur du X86, pas de soucis, mais attention si le code doit être exécuté sur des archi spécifiques. Il en existe un pour C11 ou C++11, mais très relâché.

En Java, tout réordonnancement est possible, mais les variables déclarées *volatiles* génèrent une barrière mémoire à chaque accès. De plus, des barrières seront présentes lors de l'entrée et la sortie des sections critiques.

En particulier, il faut éviter d'utiliser un lock non volatile pour initialiser une variable.

4 Structures de données

4.1 La pile

Structure *LIFO* : Last In First Out. La pile possède deux méthodes : `push` (empiler) et `pull` (dépiler).

Avec des locks : On verrouille lors des `push`/`pop`.

Sans lock : L'idée est de faire un `atomic-CAS` sur le pointeur de tête.

```
1 void push(head, element)
2 |  do
3 |  |  Node n = new Node (head, element);
4 |  while atomic-CAS(&head, n.next, n) != n.next;
  |  // Si la tête n'as pas changé, on insère en changeant la valeur de la tête
```

On voit ici un *point de linéarisabilité* : on ramène `push` à une seul opération atomique.

Le principe est la même pour `pop`

```
1 void pop(head)
2 |  do
3 |  |  Node n = head;
4 |  while atomic-CAS(&head, n, n.next) != n;
```

Au niveaux des garanties : `push` et `pop` sont obstruction-free (en coupant tous les autres threads, au pire on recommence un seule fois car la tête n'est plus modifiée. Elles sont également lock-free, (se prouve par induction sur le nombre de threads) : si on demande un infinité de push, une infinité est bien exécutée mais pas forcément par tous les threads. Il peut par contre y avoir famine (un thread peut être infiniment bloqué). Par contre, il n'est pas wait-free.

4.2 La queue

Il s'agit d'une structure *FIFO* : first in first out. Deux fonctions sont possibles : `enqueue(element e)` pour ajouter un élément à la tête, et `dequeue()` pour le retirer. Il y a donc deux pointeurs : `head` et `tail` (pointeur de début pour dequeue de fin pour l'ajout de nouveaux élément). Il faut faire attention au cas de liste vide, où l'on doit changer à la fois `head` et `tail`.

Pour gérer ces cas embêtants, on rajout un faux noeud (la list ne peut plus être vide). Pour le `dequeue()` (côté tail), on va utiliser le même algorithme que `pop()`. On va conserver à tout instant trois invariants :

— Le premier noeud est en seconde position (la première est le faux noeud)
— Les listes commençant par `head` et `tail` se rejoignent
— Le dernier noeud est toujours le noeud de queue

```
1 void enqueue(Element e)
2 |   Node node = new Node (null, e);
3 |   do
4 |   |   Node old = tail;
5 |   |   while old.next != null do
6 |   |   |   CAS(&tail, old, old.next);
7 |   |   |   old = tail;
8 |   while CAS(&old.next, null, node) != node;
9 |   CAS(&tail, old, node);
```

Le `dequeue` est le même qu'avant à l'exception près du faux noeud à ne pas prendre en compte. Dans ce cas, le fake évolue pour être l'élément retiré.

4.3 La liste chainée

On va autoriser des noeuds non valides à exister dans la liste : on les marques gris et on les ignore. Pour les griser, on va mettre à 1 le bit de poids faible de pointeur (ce qui ne l'invalide pas car les adresses mémoires sont alignées). Pour accéder au pointeur, on effectue $ptr \ \& \ -2$, pour accéder à la couleur, $ptr \ \& \ 1$.

Les invariants seront alors :

— La liste est toujours connectée et ordonnée
— Certains noeuds supprimées peuvent rester dans la liste

```
1  void del(coloredPointer* plist, int value)
2      restart :
3          coloredPointer* pred = plist;
4          while !found do
5              Node curr = pointer(*pred);
6              if curr == null || value < curr→value then
                   // pas trouvé
7                  return;
8              if curr → value == value then
                   // trouvé
9                  do
10                     n = curr → next;
11                 while CAS(&cur→next, n, n | 1) != n;
12             if mark(curr→mext) then
                   /* curr doit être supprimé                                    */
13                 if CAS(pred, curr, pointer(curr→next)) != cur then
14                     goto restart;
                       // On est allé trop loin dans la liste
15                 else
16                     continue;
17             pred = &curr→next;
```

Ajout :

```
1  void add(coloredPointer* plist, int value)
2      restart :
3          coloredPointer* pred = plist;
4          while !found do
5              Node curr = pointer(*pred);
6              if curr == null || value < curr → value then
7                  if CAS(pred, curr, new Node(curr, value)) != curr then
8                      goto restart;
9                  else
10                     return;
11             if mark(curr→mext) then
                   /* curr doit être supprimé                                    */
12                 if CAS(pred, curr, pointer(curr→next)) != cur then
13                     goto restart;
                       // On est allé trop loin dans la liste
14                 else
15                     continue;
16             pred = &curr→next;
```

L'algorithme est obstruction-free et lock-free mais pas wait-free.

Implémentation

```
1 struct node
2 |  struct node* next;
3 |  int value;
4 struct colorPtr
5 |  uintptr_t ptr :63// indique que l'on prends 63 bit
6 |  uintptr_t color :1;
```

On se force à utiliser des structures sur 64 bits car le CAS utilise des structures 64 bits.

Pour l'implémentation, procéder par étape : d'abord le parcours, puis chaque cas un à un.

5 Mémoire Transactionnelle

5.1 Pourquoi ?

Les algorithmes lock-free sont très difficiles à implémenter car théoriquement très difficile. Une solution intermédiaire consiste à dégrader quelque peu les performances (souvent plus rapide que le lock, mais légèrement plus lent que le lock-free), mais gagner une simplicité de programmation énorme.

Rappel : selon la loi d'Amdahl, le facteur prédominant est (en HPC) les quelques pourcents de code non parallélisable. Pour réduire au maximum la séquentialisation, on peut :

— Utiliser des lock plus nombreux (verrou à grain fin)

— Utiliser des compare-and-swap

Statistiquement, la probabilité de conflit est assez faible, on va donc essayer de supprimer les verrous. En effet, les verrous fins sont :

— Difficile à maintenir (*très très important en HPC*)

— Difficilement débogable

— Difficilement prouvable

— Difficile à composer (quasi impossible)

— Effet de bord important : protocole d'accès à une variable implicite (pour accéder à telle variable je dois faire telle chose avant

En outre, le verrouillage fin reste très pessimiste. L'idée des mémoires transactionnelles est de faire du verrouillage optimiste. On veut garder *l'atomicité* des sections critiques ($\rightarrow$ pas d'intermédiaires vu par les autres threads). On va donc laisser les threads s'exécuter en parallèles, mais revenir à un état antérieur en cas de problème.

On s'attend alors à :

— simplifier le code : pas besoin de connaître l'ordre des verrous

— non-deadlock, non-famine, composabilité assurée par la plate-forme

— on ne bloque un coeur que si c'est nécessaire (i.e. car conflit)

Dans la pratique, cela est faux. Parfois on gagne, parfois on perd par rapport à un verrouillage à grain fin, mais on reste plus performant qu'un unique verrou (et moins qu'un lock-free).

Definition 2 (Bloc atomique)**.** *Un bloc atomique est exécuté de façon atomique. Cela signifie qu'il semble s'exécuter instantanément aux yeux des autres threads.*

Dans ce cas, la programmation devient bien plus simple ; la composition de block atomique est bien plus simple. Le mot-clef `retry` permet de mettre le thread en attente et de retenter la section transactionnelle lorsqu'une des variables lue avant le retry est modifiée. On peut alors utiliser `orElse` qui permet d'exécuter une autre section en cas d'avortement de la transaction.

5.2 Mise en oeuvre

Il existe deux types de mémoire transactionnelle :

1. Mémoire transactionnelle retardée (*deferred-update*) :
   — Écriture dans une copie locale, propagée en mémoire centrale au commit
   — Dédouble les écritures mémoires (→ inutile lorsqu'il y a peu de confilts)
   — Cohérence facile à maintenir
2. Mémoire transactionnelle immédiate (*direct-update*) :
   — Écriture directement dans la mémoire centrale, annulation si la transaction avorte
   — Très peu de travail au commit
   — Cohérence plus difficile à maintenir (doit garder les valeurs originales si la transaction avorte).

Pour la détection de conflits : un conflit arrive lorsque deux threads travaillent en lecture/écriture sur la même variable. Dans le cas d'un conflit lecteur/écrivain, il faut annuler une des deux transactions (une bonne heuristique est de laisser commit le premier arrivé et d'avorter les autre). Dans le cas d'un conflit écrivain/écrivain, il est également nécessaire d'avorter une des deux transactions. Attention, il n'y a pas de conflit lecteur/lecteur.

Deux systèmes de détection de conflit sont possible :
— Détection au plus tôt (*eager)* : dès que le conflit est observé, on annule la transaction
— Détection au plus tard (*lazy* : on regarde les conflits lors du commit

Trois manières différentes sont possibles pour construire une mémoire transactionnelle :
— Mémoire transactionnelle matérielle (HTM) : on garde les données dans le cache sans propager les écritures. Cela est donc bien plus rapide, mais ne convient pas pour les grandes/longues transactions. En effet, les transaction sont limitées au L1 ($\sim$ 64 ko) ; les appels systèmes (interruptions matérielles) et switch de contextes avortent.
— Mémoire transactionnelle logicielle (STM) : On met une barrière en lecture/écriture inséré par le compilateur (ou à la main). C'est plus lent, mais on peut ainsi gérer des transactions de tailles quelconques.
— Mémoire transactionnelle hybride (HyTM) : On reste dans de la mémoire matérielle tant que le cache suffit (en cas de conflit à cause de la limite de taille), on passe en logiciel sinon.

Mise en œuvre logicielle

On peut utiliser une STM délayée au plus tard avec un verrou au commit.
Principe :
— Chaque case mémoire possède un compteur
— Le compteur correspond à la dernière date d'accès (*timestamp*)

Il y a cependant un soucis : la *transaction zombie*. On peut voir une transaction qui n'a pas été effectuée [4]. Si l'on lit une valeur qui a été modifiée entre temps, on peut lire une mauvaise valeur entrainant *par structure* un arrêt du programme (par exemple `if (x != NULL) { x->f() }` et `x` modifié à `NULL` entre temps). Le problème reste présent si l'on avorte avec la seule lecture du compteur à cause d'invariants du programmes (des garanties sémantiques de l'état mémoire inconnues du CPU). On peut imaginer modifier la valeur de deux valeurs `x` et `y` par un thread, et qu'un second thread lise l'ancienne valeur de `x` et la nouvelle de `y`, entrainant une division par 0 (ainsi la transaction n'arrive pas jusqu'au commit).

On rajoute donc une horloge globale absolue partagée, elle donne le numéro de la transaction. Au début de la transaction, on copie l'horloge globale dans l'horloge locale. A chaque lecture, on vérifie que le compteur est strictement plus petit que l'horloge (sinon on annule), et on l'ajout au readset. A chaque écriture, on l'ajoute simplement au writeSet. Lors des commits, si un compteur RS ou WS *distant* est plus grand que l'horloge *locale*, on avorte. Pour toute variable, on met à jour son timestamp au compteur global et sa valeur ; puis on incrémente de timestamp global.

4. C'est peut-être le problème ayant entrainé la désactivation du support HTM sur les CPU Intel Hasswell