Chapitre 1

Introduction

Pourquoi faire du multicoeurs :
- Limite thermique
- Limite des accès mémoires
- Limite de l'intégration (mettre $\approx 10^{12}$ mots sur une surface de $0.3^2 mm^2$)

La solution est le CMP, une architecture MIMD (Multiple Instruction Stream, Multiple Data stream), il s'agit d'un ensemble de thread qui exécute un graphe de précédence.

1.1 Loi d'Amdhal

Le gain de vitesse (speed-up) pour une architecture à n coeurs est donné par $S(n) = \frac{T_1}{T_n}$ où T_n est le temps pour exécuter le problème avec n coeurs. Idéalement, il faudrait avoir $S(n) = n$: aller n fois plus vite. Mais c'est rarement le cas !

Tout programme contient une partie séquentielle *seq* et une partie *par* pouvant être exécutée en parallèle. Soit p la partie séquentielle du problème : $p = \frac{seq}{seq+par}$.

$$S(n) = \frac{T_1}{T_1 p + \frac{(1-p)T_1}{n}} = \frac{1}{p + \frac{1-p}{n}}$$

Exemple : 60% concurrent et 40% séquentielle : $S(10) = \frac{1}{0.4+\frac{0.6}{10}} = 2.17$, $S(10) = \frac{1}{0.99+\frac{0.01}{10}} = 9.17$.

Conclusion : le petit % séquentielle influence fortement le gain en vitesse fourni par un CMP.

1.2 Efficacité

L'efficacité $E(n)$ est l'utilisation moyenne des n coeurs pour exécuter l'application : $E(n) = \frac{S(n)}{n}$. $E(n) = 1$ implique que $S(n) = n$. Généralement une augmentation de $S(n)$ (en donnant des coeurs à l'application) se réalise au détriment de l'efficacité.

1.3 Caractérisation du parallélisme

— p : fraction séquentiel de l'application
— m_{min}
— m_{max}
— p_i proportion de l'exécution avec i coeurs
— A : paralélisme moyen de l'application

$$A = \sum_{i=m_{min}}^{m_{max}} ip_i$$

A peut se définir comme étant
— égal au nombre moyen de coeurs utilisés durant l'exécution de l'application si $n \geq m_{max}$.
— égal au gain de vitesse $S(\infty) = \frac{T_1}{T_\infty} = \frac{AT_\infty}{T_\infty}$.
— le rapport entre la somme du temps d'exécution de toutes les actions du graphe de précédence de l'application et le chemin le plus long de la racine aux feuilles de ce graphe.

1.4 Relation entre $S(n)$ et A

Soit A le parallélisme moyen d'une application, $S(n)$ son gain de vitesse pour n coeurs et $E(n)$ l'efficacité des n coeurs. Alors $S(n) \geq \frac{nA}{n+A-1}$ et $E(n) \geq \frac{A}{n+A-1}$.

La programmation concurrente multicoeurs est difficile :
— Limitation de la loi d'Amdhal.
— Idéalement il faut maintenir une efficacité élevée.
— Le dimensionnement de la granularité de calcul est difficile (granularité = exécution fait entre 2 points de synchronisation).
— la localité des données freine la vitesse (conserver le maximum de données en antémémoire).
— le balancement de la charge de chaque thread n'est pas évident.
— le partage de données et la synchronisation entre les threads doivent être rapide.

Chapitre 2

Exclusion mutuelle

Un thread est une suite ordonnée d'événemments. Si a_0 et a_1 sont deux événements d'un thread, on peut les représenter schématiquement sur l'axe du temps. Un intervalle $A = (a, b)$ est l'activité (ensemble d'événements) compris entre deux événements spécifiques a et b.

Un événement a précède b si a se produit avant b : se note $a \to b$. Un intervalle $A = (a_0, a_1)$ précède $B = (b_0, b_1)$, si $a_1 \to b_0$ ou $a_1 = b_0$. Propriétés :

- irréflexive $A \to A$ est toujours faux
- antisymmétrique : si $A \to B$ alors $B \to A$ est faux
- transitive : si $A \to B \vee B \to C$ alors $A \to C$

La précédence ne peut être définis si A et B sont dans des threads différents. Les intervalles qui représentes les sections critiques de deux programmes ne peuvent pas se chevaucher.

2.1 Algorithme de Peterson

```
volatile boolean_t flag[] = {FALSE,FALSE};
volatile unsigned victim;

cilk int Thread(unsigned i) // i = 0 ou 1
{
    unsigned j = 1 - i;
    flag[i] = TRUE;
    victim = i;
    Cilk_fence();

    while (flag[j] && victim == i);
    // section critique
    flag[i] = FALSE;

    Cilk_fence();
    return 0;
}
```

L'algorithme de Peterson garanti l'exclusion mutuelle. Cet algorithme est dépourvue d'interblocage (un interblocage surgit quand aucun thread ne peut progresser), pour qu'il en ai un dans

notre cas, il faudrait que les deux threads soient pris dans leur bloucle respective. L'algorithme de Peterson est aussi dépouvue de famine.

Pour n threads, on crée $n+1$ niveaux, quand un thread est en dehors de sa section critique, il est au niveau 0. Si un thread souhaite entrer en section critique, il doit franchir les n niveaux restant. À chaque niveaux i il ne peut y avoir que $n-i+1$ threads au maximum.

```
unsigned n;          // nombre de threads
unsigned level[n];   // niveau courant du thread i (ini à 0)
unsigned victim[n]; // victime au niveau L

cilk int Thread(unsigned i) // i = 0..n-1
{
    unsigned k, L;
    for (L = 1; L < n; L += 1)
    {
        level[i] = L;
        victim[L] = i;
        Cilk_fence();

        for (k = 0; k < n; )
        {
            if (k != i && level[k] >= L && victim[L] == i)
                k = 0;
            else
                k += 1;
        }
    }
    // section critique
    level[i] = 0;

    Cilk_fence();
    return 0;
}
```

Affirmations :

- Quand T_i passe au du niveau L au niveau $L+1$, uniquement une des deux conditions est possible :
  - C1 : T_i précède tous les autres threads
  - C2 : T_i n'est pas le seul au niveau L
- S'il y a plus d'un thread au niveau L, il doit y avoir au moins 1 threads dans tous les niveaux 1 à $L-1$
- Il y a au plus $n-L+1$ threads au niveau L

2.2 Algorithme de la boulangerie

Idée : chaque thread choisit un numéro en entrant qui reflète son ordre de passage (comme à la poste).

L'algorithme ne s'appuie sur aucun dispositif centralisé et chaque thread choisit son propre

numéro en fonction de ceux pris par les autres. En cas d'égalité, il faut départager les threads :

$$n_i << n_j \text{ si } n_i < n_j \vee (n_i = n_j \wedge i < j)$$

avec n_i et n_j sont des numéros tirés respectivement par les threads i et j.

L'algorothme dispose de deux tableaux : `flag[i]` (état du thread i) et `label[i]` (numéro tiré par thread i). Avec 0 initilement dans toutes les cases de tous les threads.

```
unsigned threads;         // nombre de threads
boolean_t flag[threads]; // état courant du thread i
int64_t label[threads];  // numéro pris par le thread i

cilk int Thread(unsigned i) // i = 0..threads-1
{
    unsigned j, k;
    int64_t max;
    flag[i] = TRUE;

    for (max = label[0], k = 1; k < threads; k += 1)
        if (label[k] > max) max = label[k];
            label[i] = max + 1;

    Cilk_fence();

    for (k = 0; k < threads; )
        if (flag[k] && label[i] >> label[k])
            k = 0;
        else
            k += 1;

    // SC
    flag[i] = FALSE;

    Cilk_fence();
    return 0;
}
```

L'algorithme préserve l'exclusion mutuelle (voir slides 20 - 21). Il n'y a pas d'interblocage : `label[A]` » `label[B]` pour A et `label[B]` » `label[A]` pour B, donc impossible. L'algorithme garantit un traitement FIFO. C'est un algorithme simple et équitable mais pas pratique : il faut lire n variables différentes. Si n est grand, la pénalité est grande (loi d'Amdahl).

		writer (W)	
		single (S)	multi (M)
reader (R)	single (S)	SRSW	SRMW
	multi (M)	MRSW	MRMW

Théorème 1: Variables partagées

Au moins n MRSW variables partagées sont nécessaire pour résoudre le problème de l'exclusion mutuelle avec n threads

2.3 Verrou TAS

Certaines architectures offrent une instruction atomique mettant un mot à vrai et retournant sa valeur précédente :

```
bool TestAndSet(bool* adr)
{
    bool val = *adr;
    *adr = true;
    return val;
}
```

Avec cela on peut faire un verrou (verrou TAS) :

```
bool lock = false; // lock

void TAS_AcquireLock(bool* lock)
{
    while(TestAndSet(lock));
}

void TAS_RealeaseLock(bool* lock)
{
    *lock = false;
}
```

Propriétés :

- Nécessite 1 booléen par verrou (indépendant du nombre de threads)
- Famine possible en théorie mais improbable en pratique
- Pas FIFO
- Si l'architecture a des antémémoires cohérentes, chaque appel à `TestAndSet` crée une invalidation des antémémoires ce qui augemente le trafic sur les bus.

2.4 Verrou TATAS

```
bool lock = false; // verrou
void TATAS_AcquireLock(bool *lock)
{
    while (true)
    {
        while (*lock);
        if (!TestAndSet(lock))
            return;
    }
}

void TATAS_ReleaseLock(bool *lock)
{
    *lock = false;
}
```

TATAS est logiquement équivalent à TAS mais diffère en performance. Environ 25% plus rapide sur un Core 2 Duo avec 2 threads.

2.5 Verrou TATAS avec attente

```
void TATASBackoffAcquireLock(bool *lock)
{
    int delay;

    while (*lock);
        if (!TestAndSet(lock))
            return;

    delay = random(MIN_DELAY,MAX_DELAY);

    while (true)
    {
        sleep(delay);
        while (*lock);
            if (!TestAndSet(lock))
                return;
        delay = min(2*delay,MAX_WAIT);
    }
}
```

Si le `TestAndSet()` de TATAS ne réussit pas, c'est qu'il y a d'autre threads qui sollicitent le verrou *lock*. On attend donc un temps aléatoire avant de réessayer. Le gain en peformance augmente avec le nombre de coeurs. Portabilité et redimensionnement des délais difficile.

2.6 Verrou ticket à attente proportionnelle

```
typedef struct
{
    unsigned nextTicket;
    unsigned nowServing;
} LOCK;

LOCK lock = {0U,0U};

void TicketAcquireLock(LOCK *lock)
{
    unsigned myTicket = FetchAndIncrement(lock->nextTicket);
    while (true)
        if (lock->nowServing == myTicket)
            return;
        else
            sleep(myTicket - lock->nowServing);
}

void TicketReleaseLock(LOCK *lock)
{
    lock->nowServing += 1;
}
```

L'idée revient à prendre un numéro et à attendre son tour, l'implémentation utilise l'instruction atomique `FetchAndIncrement` :

```
int FetchAndIncrement(int *num)
{
    int pred = *num;     // fetch
    *num += 1;           // increment
    return pred;
}
```

2.7 Verrou Anderson

L'idée est de réaliser une file FIFO où chaque coeur boucle sur une variable qui lui est propre. L'implémentation utilise deux instructions atomiques : `AtomicAdd` et `FetchAndIncrement` :

```
int AtomicAdd(int *num, int inc)
{
    return *num += inc;
}
```

Algorithme :

Algorithm 1: Verrou Anderson

Data: Un tableau de booléen : *slots*, avec la première case à true et les autres à false
Data: Un *int* : *nextSlot*, la prochaine case de libre
T_1 acquiert le verrou par un `FetchAndIncrement` sur `slot[nextSlot]`, ce qui retourne true et incrémente `nextSlot`;
T_2 veut acquérir le verrou, il fait un FAI mais obtient false. `nextSlot` est incrémenté quand même;
T_2 attend car sont slot est à false;
T_1 rend le verrou est mettant son slot + 1 à true;
T_2 voit true dans son slot et commence sa SC;

```
typedef struct
{
    bool *slots;
    int nextSlot;
} LOCK;

int numThreads;

LOCK *AndersonCreateLock(void)
{
    LOCK *lock;
    int i;
    if ((lock = (LOCK *)malloc(sizeof(LOCK))) != NULL)
        if (lock->slots = (bool *)malloc(numThreads *sizeof(bool)))
        {
            lock->slots[0] = true;
            for (i = 1; i < numThreads; i += 1)
            lock->slots[i] = false;
            lock->nextSlot = numThreads;
        }
        else
        {
            free(lock); lock = NULL;
        }
    return lock;
}

int AndersonAcquireLock(LOCK *lock)
{
    int mySlot = FetchAndIncrement(&lock->nextSlot);

    if (mySlot % numThreads == 0)
        AtomicAdd(&lock->nextSlot,-numThreads);
    mySlot %= numThreads;

    while (!lock->slots[mySlot]);

    lock->slots[mySlot] = false;
    return mySlot;
}

void AndersonReleaseLock(LOCK *lock, int mySlot)
{
    lock->slots[(mySlot+1) % numThreads] = true;
}
```

Ce verrou nécessite un tableau par verrou et 1 mot pou représenter le verrou : pour V verrous et T threads, il faut $V(T+1)$ mots. Le nombre de threads de l'application doit être statique.

2.8 Verrou Graunke et Thakkar

Même propriétés que le verrou d'Anderson mais avec une instruction atomique en moins, lergèrement plus performant. Utilise l'instruction atomique `FetchAndStore`.

```
int FetchAndStore(int *adr, int val)
{
    int ret = *adr;  // fetch
    *adr = val;      // store
    return ret;
}
```

Algorithm 2: Verrou Graunke et Thakkar

Data: Un tableau de booléen : *slots*, initialisé à true
Data: Un pointeur sur bool : *tail*, ontient le dernier thread avec le verrou
Data: Un bool *whatIsLocked* initilisé à false
T_1 essaie d'acquérir le verrou : FetchAndStore sur un tuple `(tail,whatIsLocked)` et récupère `(before,spin)`;
T_1 boucle sur `spin == *before` ; `// attente active`
T_1 rentre en section critique;
T_2 essaie d'acquérir le verrou;
T_2 boucle sur `spin == *before` ; `// attente active`
T_1 met `slots[id] =!slots[id]` T_2 aquiert le verrou et rentre en section critique;

C'est algorithme est possible car l'implémentation se fie sur le fait que les adresses sont alignées sur des adresses paires. La variable `tail` peut alors être fusionnée avec `whatIsLocked` : le dernier bit de l'adresse prend alors la valeur de `whatIsLocked`.

```
typedef struct
{
    bool *slots;
    bool *tail;
} LOCK;

int numThreads;
LOCK *GraunkeThakkarCreateLock(void)
{
    LOCK *lock;
    int i;
    if ((lock = (LOCK *)malloc(sizeof(LOCK))) != NULL)
        if ((lock->slots = (bool *)malloc(numThreads *sizeof(bool))) != NULL)
        {
            for (i = 0; i < numThreads; i += 1)
                lock->slots[i] = 0x1;
            lock->tail = &lock->slots[0];
        }
        else
        {
            free(lock); lock = NULL;
        }
    return lock;
}

void GraunkeThakkarAcquireLock(LOCK *lock, int threadID)
{
    bool *before, *mySlot, spin;
    mySlot = &lock->slots[threadID] | lock->slots[threadID];
    before = (bool *)FetchAndStore(lock->tail,mySlot);

    spin = before & 0x1;
    before ^= spin;
    while (spin == *before);
}

void GraunkeThakkarReleaseLock(LOCK *lock, int threadID)
{
    lock->slots[threadID] ^= 0x1;
}
```

2.9 Verrou MCS

L'idée est de former une list chaînée représentant l'ordre des requêtes où chaque thread boucle sur une variable différente qu'ilmet à disposition. Cela utilse l'instruction `CompareAndSwap` :

```
bool CompareAndSwap(int *adr, int expectedVal, int newVal)
{
    if (*adr == expectedVal)
    {
        *adr = newVal;
        return true;
    }
    else
        return false;
}
```

Algorithm 3: Verrou MCS

Data: Un id `id`
Data: Un noeud avec un booléen `locked` et un pointeur sur le suivant `next`
Data: Un noeud global `lock`
T_1 tente d'acquérir le verrou, il fait un `FetchAndStore` sur `lock` et stocke son propre noeud;
T_1 rentre en section critique car `lock` vaut `NULL`;
T_2 tente d'acquérir le verrou, il fait un `FetchAndStore` sur `lock` et stocke son propre noeud;
T_2 boucle sur son propre noeud car `lock` n'est pas `NULL`;
T_1 relache le verrou et effectuant un `CompareAndSwap` sur `(lock,myNode,NULL)`;

```
typedef struct QNODE
{
    struct QNODE *next;
    bool locked;
} QNODE;

int numThreads;
QNODE *lock = NULL;
QNODE qnodes[numThreads];

void MCSAcquireLock(QNODE **lock, int threadId)
{
    QNODE *predecessor, *myNode = &qnodes[threadId];
    myNode->next = NULL;
    predecessor = FetchAndStore(lock,myNode);
    if (predecessor != NULL)
    {
        myNode->locked = true; predecessor->next = myNode;
        while (myNode->locked);
    }
}

void MCSReleaseLock(QNODE **lock, int threadId)
{
    QNODE *myNode = &qnodes[threadId];
    if (myNode->next == NULL)
    {
        if (CompareAndSwap(lock,myNode,NULL))
            return;
        while (myNode->next == NULL);
    }
    myNode->next->locked = false;
}
```

2.10 Verrou CLH

Utilise l'instruction atomique `FetchAndStore`, utilise 2 mots par thread et 3 mots par verrou. Pour V verrous et T threads, nous avons donc $3V + 2T$ mots. Un peu plus rapide que MCS à cause des lectures en moins.

Algorithm 4: Verrou CLH

Data: Un noeud par thread avec un pointeur sur le prochain noeud `next` et un booléen `locked`
Data: Un noeud global `lock` qui pointe vers un autre noeud à `true`
T_1 tente d'acquérir le verrou, il fait un `CAS` sur le noeud pointé par `lock`;
T_1 entre en section critique si ce qu'il récupère vaut n'est pas lock;
T_2 tente d'acquérir le verrou, il fait un `CAS` sur le noeud pointé par `lock`;
T_2 boucle sur son lock car il n'est pas libre;
T_1 libère le verrou en mettant son `next` à `true`;

Chapitre 3

Objets concurrents

En programmation concurrente, lorsque l'on fait appel à un objet et qu'on utilise des méthodes sur celui-ci, les instructions dans la même méthodes peuvent se chevaucher mais aussi pour l'ensemble des méthodes de l'objet. Pour prouver un objet, il faut donc prendre en compte toutes les interactions possibles, aussi quand on rajoute des méthodes.

Sur un file FIFO avec verrou, chaque modification de l'objet se fait par exclusion mutuelle :
- pas d'action qui se chevauche
- même comportement qu'en séquentiel

L'idée est que pour un comportement concurrent, un équivalent séquentiel. Chaque action réalisée sur un objet doit :
- prendre effet
- paraître instantanée entre son invocation et son retour

un tel objet est dit *atomique*. Définitions :
- L'invocation d'une méthode est définie par 2 événements : son invocation et sa réponse.
- La durée d'une méthode est un intervalle débutant par son invocation et terminant par sa réponse.
- Une invocation et une réponse sont dites couplées si la réponse correspond à l'invocation.
- L'invocation d'une méthode est dite pendante si sa réponse n'a pas encore eu lieu.
- Un historique est une suite d'invocation et de réponse.
- Un historique est complet si toutes ses invocations sont couplées à leur réponse, sinon il est partiel.

Notation :

Inocation : `<thread><objet>.<méthode>(<arguments>)`

Réponse : `<thread><objet>:<résultat>`

Un historique est dit séqentiel quand son premier événement est une invocation que toutes les invocations et réponses sont couplées. Un historique qui n'est pas séquentiel est dit concurrent. Pas tous les historiques ont un sens, seuls les historiques filtrés par threads donnant des sous historiques séquentiels sont permis.

Deux historiques sont dit *équivalent* si pour tous les threads A de H, $H|A = G|A$.

Un historique H est légal si pour tous les objets x dans H,$H|x$ satisfait à la spécification séquentille de x. Un historique séquentiel est simple à vérifier car l'historique définit un ordre total.

Comme les threads ne réalisent pas que des historiques séquentiels, il faut transformer un historique concurrent en un historique séquentiel équivalent.

3.1 Précédence

Une opération sur un objet précède une autre si la réponse de la première précède l'invocation de la seconde, une opération sur un objet chevauche une autre si la réponse de la première ne précède pas l'invocation de la seconde.

3.2 Atomicité

Un historique H est atomique si H peut être étendu à un historique G tel que G est équivalent à un historique séquentiel S. Remarques :

- La transformation de H à G permet de tenir compte de tous les appels pendants.
- L'historique S est une atomisation de H.
- L'historique H peut avoir plusieurs atomisations S.

Méthodes :

- Identifier les points d'atomicités (mettre les traits bleu au endroit logique) ;
- Traiter les opérations pendantes (si point d'atomicité atteint, on écrit le retour de la méthode à la fin de l'historique, sinon on la supprime) ;
- Trouver un historique séquentiel équivalent.