PROGRAMMATION A BASE DE THREADS

Marc Pérache marc.perache@cea.fr

Arthur Loussert

RAPPEL: EXCLUSION MUTUELLE (MUTEX)

Rappel: Anatomie d'un mutex

- Structure d'un mutex ainsi que d'un slot pour gérer la liste des threads en attente

```
typedef struct slot_s {
  thread_t *thread;
  struct slot_s *next;
} slot_t;

typedef struct {
  /* Compteur de threads dans ou en attente de section critique */
  volatile int nb_threads;
  /* Liste des thread bloqués */
  volatile slot_t *list_first;
  volatile slot_t *list_last;
  /* Spinlock pour verrouiller les accès aux données du mutex */
  spinlock_t lock;
} mutex_t;
```

Rappel: Anatomie d'un mutex

- Fonction de verrouillage : lock

```
void mutex_lock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        spinunlock (&(m→lock));
    } else {
        slot.thread = thread_self ();
        enqueue (&slot, m);
        thread_self ()→status = blocked;
        register_spinunlock (&(m→lock));
        yield ();
    }
}
```

Rappel: Anatomie d'un mutex

- Fonction de déverrouillage : unlock

```
void mutex_unlock (mutex_t * m)
{
    slot_t *slot;
    spinlock (&(m→lock));
    if (m→list_first != NULL) {
        slot = dequeue (m);
        wake (slot→thread);
    } else {
        m→nb_thread = 0;
    }
    spinunlock (&(m→lock));
}
```

Rappel: Anatomie d'un mutex

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        spinunlock (&(m→lock));
    return 0;
    }
    spinunlock (&(m→lock));
    return 1;
}
```

Rappel: Anatomie d'un mutex

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m)
{
     slot_t slot;
     int res = 1;
     spinlock (&(m→lock));
     if (m→nb_thread == 0) {
          m→nb_thread = 1;
          res = 0;
          goto fin;
     }
fin:
     spinunlock (&(m→lock));
     return res;
}
```

Rappel: Anatomie d'un mutex récursif

- Fonction de verrouillage : lock

```
void mutex_lock (mutex_t * m) {
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
    m→nb_thread = 1;
    m→owner = thread_self ();
    spinunlock (&(m→lock));
    } else {
        if (m→owner == thread_self ()) {
            m→step++;
            spinunlock (&(m→lock));
        } else {
            slot.thread = thread_self ();
            enqueue (&slot, m);
            thread_self ()→status = blocked;
            register_spinunlock (&(m→lock));
            yield ();
        }
    }
}
```

Rappel: Anatomie d'un mutex récursif

- Fonction de verrouillage : unlock

```
void mutex_unlock (mutex_t * m) {
    slot_t *slot;
    spinlock (&(m→lock));
    if (m→step == 1) {
        m→step--;
        if (m→list_first != NULL) {
            slot    = dequeue (m);
            wake (slot→thread);
        } else {
            m→nb_thread = 0;
        }
    } else {
        m→step--;
    }
    spinunlock (&(m→lock));
}
```

Rappel: Anatomie d'un mutex récursif

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m){
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        m→owner = thread_self ();
        spinunlock (&(m→lock));
        return 0;
    } else {
        if (m→owner == thread_self ()) {
            m→step++;
            spinunlock (&(m→lock));
            return 0;
        }
    }
    spinunlock (&(m→lock));
    return 1;
}
```

RAPPEL: SÉMAPHORES

Sémaphore - Définition

- Définition :
  - les **sémaphores** sont des compteur pour des ressources partagées par plusieurs threads.
- Opérations sur les sémaphores
  - Incrémenter le compteur
  - Décrémenter le compter
  - Si l'on essaye de décrémenter un compteur déjà à zéro, alors le thread est bloqué
- Pour utiliser les sémaphores, il faut utiliser l'include `semaphore.h`

Sémaphore - Fonctions

- Initialisation d'un sémaphore
  - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - Sémaphore non nommé
  - L'argument `value` spécifie la valeur initiale du sémaphore.
  - L'argument `pshared` indique si ce sémaphore sera partagé entre les threads d'un processus ou entre processus.
- Destruction d'un sémaphore
  - `int sem_destroy(sem_t *sem);`

Sémaphore - Fonctions

- Décrémentation (verrouillage) un sémaphore
  - `int sem_wait(sem_t *sem);`
  - Si la valeur du sémaphore est plus grande que 0, la décrémentation s'effectue et la fonction revient immédiatement.
  - Si le sémaphore vaut zéro, l'appel bloquera jusqu'à ce que soit il devienne disponible d'effectuer la décrémentation (c'est-à-dire la valeur du sémaphore n'est plus nulle).

Sémaphore - Fonctions

- Test de verrouillage
  - `int sem_trywait(sem_t *sem);`
  - Si la décrémentation ne peut pas être effectuée immédiatement, l'appel renvoie une erreur
  - La variable errno vaut EAGAIN (l'appel n'est alors pas bloquant)
- Incrémentation (déverrouillage) d'un sémaphore
  - `int sem_post(sem_t *sem);`
  - Si, à la suite de cet incrément, la valeur du sémaphore devient supérieure à zéro, un autre processus ou thread bloqué dans un appel sem_wait sera réveillé et procédera au verrouillage du sémaphore.

RAPPEL: CONDITIONS

Conditions

- Définition :
  - Une **condition** est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition (un *prédicat*) sur des données partagées soit vérifiée.

- Opérations fondamentales sur les conditions :
  - Signaler la condition (quand le prédicat devient vrai)
  - Attendre la condition en suspendant l'exécution du thread jusqu'à ce qu'un autre thread signale la condition

- Une variable condition doit toujours être associée à un **mutex**, pour éviter une condition concurrente où un thread se prépare à attendre une condition et un autre thread signale la condition juste avant que le premier n'attende réellement.

Conditions

- Initialisation d'une variable condition
  - `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
  - Utilisation des attributs par défaut si `cond_attr` vaut `NULL`.
- Initialisation statique
  - Utilisation de la constante `PTHREAD_COND_INITIALIZER`.
- Relancer un des threads attendant une variable condition
  - `int pthread_cond_signal(pthread_cond_t *cond);`
  - S'il n'existe aucun thread répondant à ce critère, rien ne se produit.
  - Si plusieurs threads attendent sur cond, seul l'un d'entre eux sera relancé, mais il est impossible de savoir lequel.

Conditions

- Relancer tous les threads attendant sur une variable condition
  - `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - Rien ne se passe s'il n'y a aucun thread attendant sur cond.

- Déverrouiller atomiquement le mutex et attendre qu'une variable condition soit signalée
  - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - L'exécution du thread est suspendue et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée.
  - Le mutex doit être verrouillé par le thread appelant à l'entrée de `pthread_cond_wait`().
  - Avant de rendre la main au thread appelant, `pthread_cond_wait`() reverrouille mutex.

Conditions

- Le déverrouillage du mutex et la suspension de l'exécution sur la variable condition sont liés atomiquement.
  - Si tous les threads verrouillent le mutex avant de signaler la condition, il est garanti que la condition ne peut être signalée (et donc ignorée) entre le moment où un thread verrouille le mutex et le moment où il attend sur la variable condition.
- Destruction d'une variable condition
  - `int pthread_cond_destroy(pthread_cond_t *cond);`
  - Librération des ressources qu'elle possède.
  - Aucun thread ne doit attendre sur la condition à l'entrée de `pthread_cond_destroy`().

Exercice

- Comment écrire des sémaphores avec mutexes et conditions ?

PROBLÈME DES PHILOSOPHES

Inspiré du cours de Michaël Rao

Problème classique : 5 philosophes

- 5 philosophes sont autour d'une table ronde
- Il y a une baguette entre chaque philosophe (i.e. : une baguette pour deux philosophes)
- Un plat de sushis pour tout le monde est au centre
- La vie d'un philosophe se résume à deux actions : penser et manger
- Pour manger, il doit prendre les 2 baguettes (à sa gauche et à sa droite), puis il peut commencer à manger
- Quand il a ni, il repose les baguettes et peut commencer à penser
- Chaque baguette : une ressource (un mutex)
- But :
  - Tout le monde doit pouvoir manger (pas de famine)
  - Limiter les attentes

Problème classique : 5 philosophes

```
mutex baguette[5];

void *philosophe(void *a){
   int i=(long long) a;
   while(1) {
      printf("%d pense\n",i);
      lock(&baguette[i]);
      lock(&baguette[(i+1)%5]);
      printf("%d mange\n",i);
      unlock(&baguette[i]);
      unlock(&baguette[(i+1)%5]);
   }
}
```

Correct ?

Problème classique : 5 philosophes

```
mutex baguette[5];

void *philosophe(void *a){
   int i=(long long) a;
   while(1) {
      printf("%d pense\n",i);
      lock(&baguette[i]);
      lock(&baguette[(i+1)%5]);
      printf("%d mange\n",i);
      unlock(&baguette[i]);
      unlock(&baguette[(i+1)%5]);
   }
}
```

Correct ?

Non : interblocage. Si tout le monde a la baguette à gauche, tout le monde est bloqué

5 philosophes : solution ?

```
mutex baguette[5], general;

void *philosophe(void *a)
{
   int i=(long long) a; while(1)
   {
      printf("%d pense\n", i);
      lock(&general);
      lock(&baguette[i]);
      lock(&baguette[(i+1)%5]);
      printf("%d mange\n", i);
      unlock(&baguette[i]);
      unlock(&baguette[(i+1)%5]); unlock
      (&general);
   }
}
```

Correct ?

5 philosophes : solution ?

```
mutex baguette[5], general;

void *philosophe(void *a)
{
   int i=(long long) a; while(1)
   {
      printf("%d pense\n", i);
      lock(&general);
      lock(&baguette[i]);
      lock(&baguette[(i+1)%5]); printf(
      "%d mange\n", i);
      unlock(&baguette[i]);
      unlock(&baguette[(i+1)%5]); unlock
      (&general);
   }
}
```

Correct... mais un seul philosophe mange à la fois. Les mutex baguette[i] ne servent à rien → une seule section critique

5 philosophes : solution ?

```
mutex baguette[5], general;

void *philosophe(void *a)
{
   int i=(long long) a; while(1)
   {
      printf("%d pense\n", i);
      lock(&general);
      lock(&baguette[i]);
      lock(&baguette[(i+1)%5]);
      unlock(&general);
      printf("%d mange\n", i);
      unlock(&baguette[i]);
      unlock(&baguette[(i+1)%5]);
   }
}
```

Mieux... mais un philosophe doit des fois attendre inutilement pour manger.

5 philosophes : solution ?

```
void *philosophe(void *a)
{
   int i=(long long) a; while(1)
   {
      printf("%d pense\n",i);
      while(1) {
         lock(&general);
         if(baguette[i]==1 &&baguette[(i+1)%5]==1) {
          baguette[i]=baguette[(i+1)%5]=0;
          unlock(&general);
           break;
         }
         unlock(&general);
      }
      printf("%d mange\n",i);
      baguette[i]=baguette[(i+1)%5]=1;
  }
}
```

Attente active et famine...

Parenthèse : Famine ?

- On peut distinguer deux types de famines :
  - la famine "avérée" : un thread est indéfiniment bloqué (quelle que soit le déroulement de la suite)
  - la famine "probabiliste" : il y a une probabilité non nulle qu'un thread reste bloqué longtemps.

- Si on s'autorise la famine "avérée", il existe une solution simple aux 5 philosophes sans interblocage, ni situation de compétition :
  - On désigne un philosophe qui n'a pas le droit de prendre de baguettes (donc ni de manger et penser)

Parenthèse : Famine ?

- Si on s'autorise la famine "probabiliste" (mais pas "avérée"), la solution précédente est bonne (modulo le fait qu'elle soit en attente active)

- Note :
  - La famine est, des fois, pas très grave (e.g. si les threads font le même travail).
  - Beaucoup considèrent que la famine probabiliste n'est pas un vrai problème.

- Ordre d'importance :
  - famine "probabiliste" / famine "avérée" / interblocage / situation de compétition

5 philosophes : solution ?

```
void *philosophe(void *a)
{
   int i=(long long) a; while(1)
   {
      printf("%d pense\n",i);
      if(i==0) {
        lock(&baguette[(i+1)%5]);
        lock(&baguette[i]);
      }else{
        lock(&baguette[i]);
        lock(&baguette[(i+1)%5]);
      }
      printf("%d mange\n",i);
      unlock(&baguette[i]);
      unlock(&baguette[(i+1)%5]);
   }
}
```

Correct. Mais pas très joli, et l'asymétrie introduit des biais (des philosophes attendent plus que d'autres en moyenne)

5 philosophes avec sémaphore

```
mutex baguette[5];
semaphore sem;
init(sem,4);

void *philosophe(void *a) {
   int i=(long long) a;
   while(1) {
      printf("%d pense\n",i);
      wait(sem);
      lock(baguette[i]);
      lock(baguette[(i+1)%5]);
      post(sem);
      printf("%d mange\n",i);
      unlock(baguette[i]);
      unlock(baguette[(i+1)%5]);
   }
}
```

Parfait ! pas de famine (même "probabiliste"), pas d'attente inutile.

PTHREAD AVANCÉ

Inspiré du cours de Michaël Rao

Read-write locks

- On pourrait permettre à plusieurs threads qui ne modifient pas la mémoire, de travailler (en lecture seule) sur une zone mémoire.

- Une solution : read-write locks (rwlocks)

- Deux types de sections critiques
  - les sections critiques en lecture seule (celles des lecteurs)
  - les sections critiques en lecture/écriture (celles des écrivains)

Read-write locks

- Garantie des rwlocks :
  - si un thread est dans une section critique en lecture/écriture, il n'y a aucun autre thread dans une section critique (ni en lecture seule, ni en lecture/écriture)
  - si aucun thread est dans une section critique en lecture/écriture, il n'y a pas de limite sur le nombre de threads dans une section critique en lecture seule

- Attention : on peut facilement arriver à des famines
  - préférer les lecteurs : il peut y avoir famine des écrivains
  - préférer les écrivains : il peut y avoir famine des lecteurs

Les rwlocks de pthreads

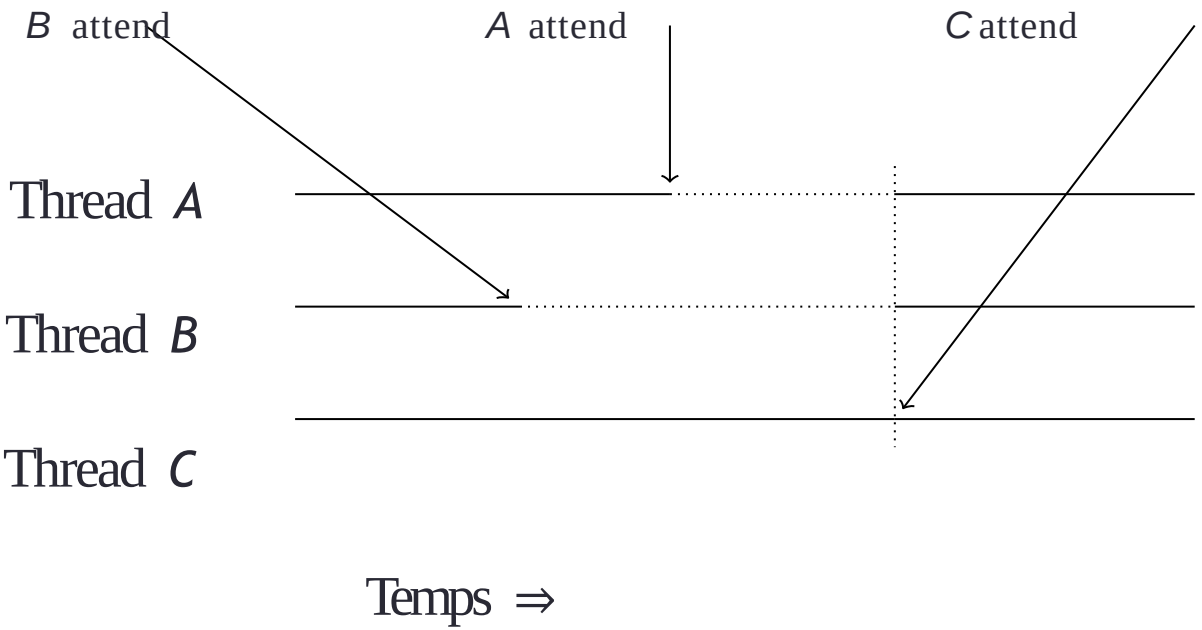
```
#include<pthread.h>
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER ;

int pthread_rwlock_init(pthread_rwlock_t *restrict
lock,const pthread_rwlockattr_t*restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t*lock);

int pthread_rwlock_rdlock(pthread_rwlock_t*lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t*lock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t*
   restrict lock,const struct timespec* restrict abstime);

int pthread_rwlock_wrlock(pthread_rwlock_t*lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t*lock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t*
   restrict lock,const struct timespec* restrict abstime);

int pthread_rwlock_unlock(pthread_rwlock_t*lock);
```

Exercice

- Comment écrire des rwlocks avec mutexes, sémaphores et/ou conditions ?

Les barrières

Les barrières permettent de synchroniser les threads



Les barrières POSIX

```
int pthread_barrier_init(pthread_barrier_t *restrict
    barrier,const pthread_barrierattr_t *restrict attr
    ,unsigned count);

int  pthread_barrier_destroy(pthread_barrier_t *barrier);


int pthread_barrier_wait(pthread_barrier_t *barrier);
```

count : nombre de threads qui doivent attendre à la barrière

Exemple :

```
pthread_barrier_t barrier;
pthread_barrier_init(&barrier,NULL,3);
```

Puis dans chaque thread (A, B et C) :

```
// section non synchronisee
pthread_barrier_wait(&barrier);
// section synchronisee
```

Exercice

- Comment écrire des barrières avec mutexes, sémaphores et/ou conditions ?

LA BIBLIOTHÈQUE MTHREAD

OPTIMISATION MULTITHREAD

Optimisation multithread

- Concurrence
- Atomicité
- Bande passante
- NUMA
- Allocation mémoire

Document de référence: *What Every Programmer Should Know About Memory*, Ulrich Drepper

Concurrence: Introduction

- Utilisation de données partagées
- Approche classique: diminution de l'empreinte mémoire de l'application
- Maximisation de la quantité de données stockées par ligne de cache
- Problème: Si plusieurs threads écrivent à une adresse mémoire d'une même ligne de cache
- La ligne de cache doit être en Exclusif dans le L1D pour chaque cœur

Concurrence: Introduction

- Les accès en écriture deviennent subitement très chers
- Si la même ligne de cache est utilisée, peut-être faut-il utiliser des synchronisations
- Ce phénomène est appelé Faux Partage (False Sharing)

Concurrence: Exemple

```
#include <error.h>
#include <pthread.h>
#include <stdlib.h>

#define N (atomic ? 10000000 : 500000000)

static int atomic;
static unsigned nthreads;
static unsigned disp;
static long **reads;

static pthread_barrier_t b;
```

Concurrence: Exemple

```
static void *
tf(void *arg)
{
  long *p = arg;

  if (atomic)
    for (int n = 0; n < N; ++n)
      __sync_add_and_fetch(p, 1);
  else
    for (int n = 0; n < N; ++n)
      {
        *p += 1;
        asm volatile("" : : "m" (*p));
      }

  return NULL;
}
```

Concurrence: Exemple

```
int
main(int argc, char *argv[])
{
  if (argc < 2)
    disp = 0;
  else
    disp = atol(argv[1]);

  if (argc < 3)
    nthreads = 2;
  else
    nthreads = atol(argv[2]) ?: 1;

  if (argc < 4)
    atomic = 1;
  else
    atomic = atol(argv[3]);

  pthread_barrier_init(&b, NULL, nthreads);
```

Concurrence: Exemple

```
void *p;
posix_memalign(&p, 64, (nthreads * disp ?: 1) * sizeof(long));
long *mem = p;

pthread_t th[nthreads];
pthread_attr_t a;
pthread_attr_init(&a);
cpu_set_t c;
for (unsigned i = 1; i < nthreads; ++i)
  {
    CPU_ZERO(&c);
    CPU_SET(i, &c);
    pthread_attr_setaffinity_np(&a, sizeof(c), &c);
    mem[i * disp] = 0;
    pthread_create(&th[i], &a, tf, &mem[i * disp]);
  }

CPU_ZERO(&c);
CPU_SET(0, &c);
pthread_setaffinity_np(pthread_self(), sizeof(c), &c);
mem[0] = 0;
tf(&mem[0]);
```

Concurrence: Exemple

```
  if ((disp == 0 && mem[0] != nthreads * N)
      || (disp != 0 && mem[0] != N))
    error(1,0,"mem[0] wrong: %ld instead of %d",
  mem[0], disp == 0 ? nthreads * N : N);

  for (unsigned i = 1; i < nthreads; ++i)
    {
      pthread_join(th[i], NULL);
      if (disp != 0 && mem[i * disp] != N)
        error(1,0,"mem[%u] wrong: %ld instead of %d", i, mem[i * disp], N);
    }

  return 0;
}
```

Concurrence: Coût

- Temps global du test
- Bleu: pas de conflit de ligne de cache
- Rouge: partage de ligne de cache

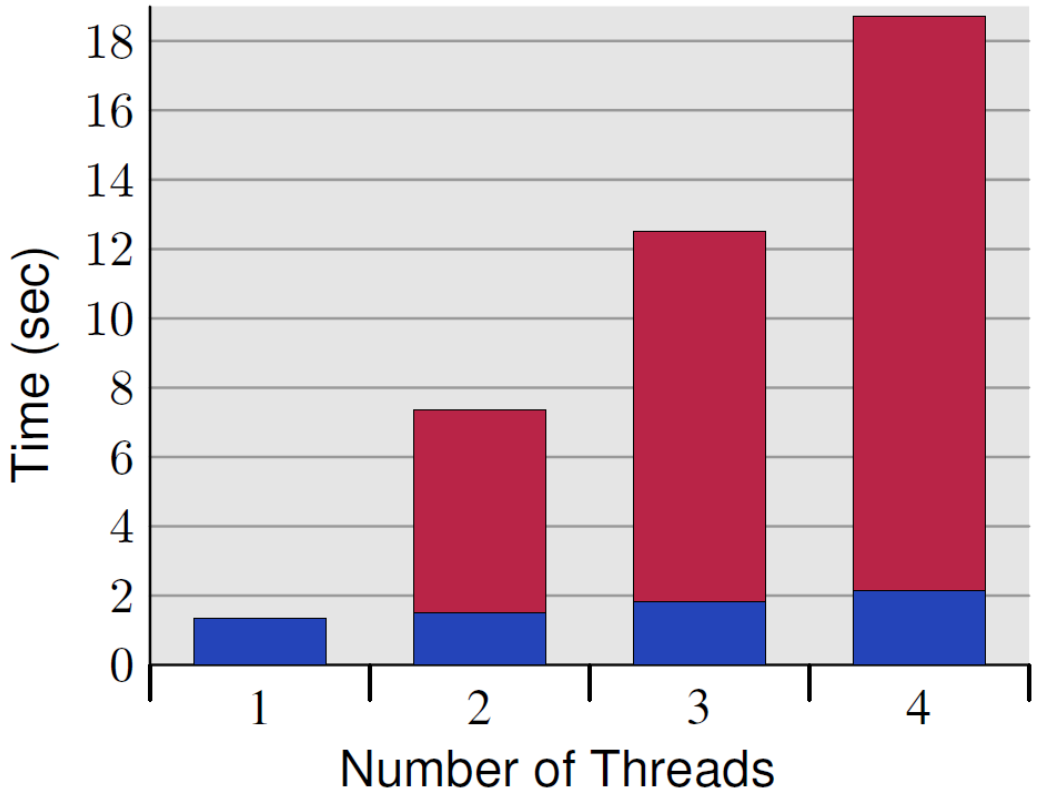


Figure 6.10: Concurrent Cache Line Access Overhead

Concurrence: Analyse

- Courbe bleue:
  - Temps d'exécution quasi stable
  - Augmentation due au bruit système et prefetching
- Courbe rouge:
  - Surcoût très important
    - 390%, 734%, 1147% respectivement

Concurrence: Analyse

- A chaque instant:
  - 1 seul cœur peut travailler à la fois
  - Concurrence sur l'acquisition de la ligne de cache
  - Tout cœur supplémentaire va seulement allonger le temps d'exécution
- Il est clair qu'il faut éviter ce cas de figure

Concurrence: Intra-socket

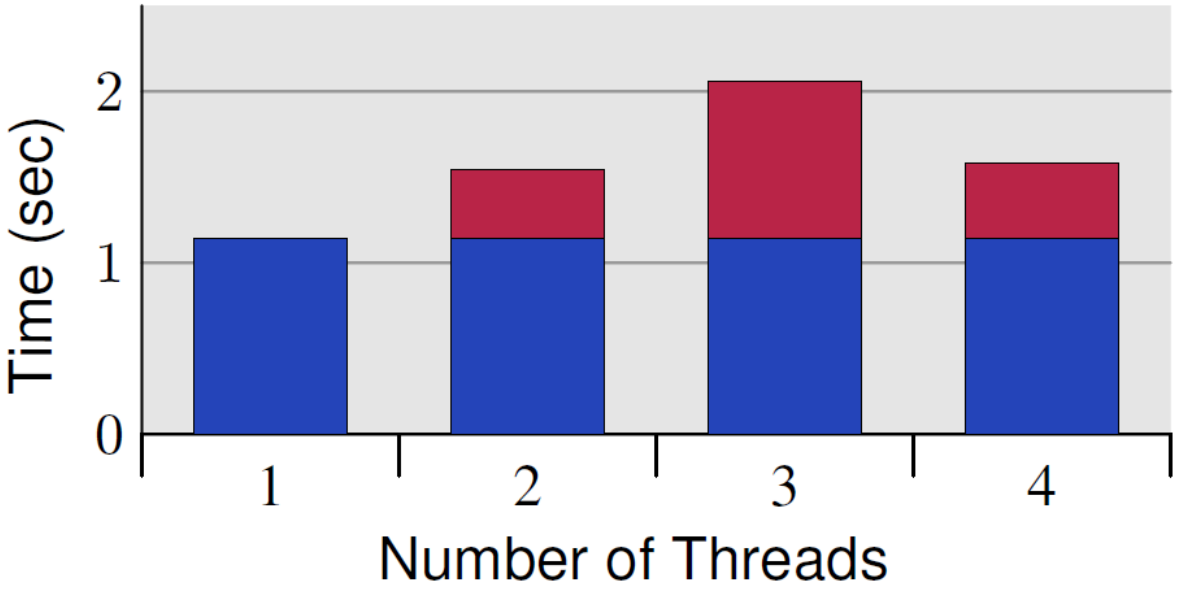


Figure 6.11: Overhead, Quad Core

Concurrence: Intra-socket

- Plus de problème visible car le cache L2 est partagé
- Cohérence de cache uniquement au niveau du L1

Concurrence: Conclusion

- Il faut tenir compte de ce problème en HPC car les architectures sont multiprocesseur
- Effet amplifié avec le NUMA
- Nombre de cœurs élevé
- Évolution des architectures

Concurrence: Comment éviter le faux partage

- Solution simple: mettre chaque variable sur une ligne de cache différente
- Problème d'augmentation de l'empreinte mémoire pas forcément acceptable
- Vers une vraie solution:
  - Identifier les variables utilisées par un seul thread
  - Identifier les variables en lecture/écriture
  - Fréquence des modifications

Concurrence: Comment éviter le faux partage

- Les variables jamais écrites ou peu souvent peuvent être considérées comme constantes et donc partagées
- Il est donc recommandé de les grouper
- Les marquer comme *const* peut aider le diagnostic
- Les marquer comme *const* déplace ces variables dans la section *readonly* du code

Concurrence: Comment éviter le faux partage

- Quelques mots clés pour les variables globales:
  - __thread: place la variable dans une section de mémoire spécifique à chaque thread
  - __attribute__((section(‘’.data.ro”))): place la variable dans la section *readonly* du code. Approche spécifique au linker GNU

Concurrence: Comment éviter le faux partage

- Cas des variable dans des structures:
  - Regrouper les champs suivant le type et le taux d'accès
  - Mettre du *padding*

```
int foo = 1;
int baz = 3;
struct {
  struct al1 {
    int bar;
    int xyzzy;
  };
  char pad[CLSIZE - sizeof(struct al1)];
} rwstruct __attribute__((aligned(CLSIZE))) =
  { { .bar = 2, .xyzzy = 4 } };
```

Modèles de programmations impactés

- MPI:
  - Non impacté
- OpenMP:
  - Problème courant
- POSIX Thread:
  - Problème courant
- Ce phénomène est souvent ignoré car difficile à diagnostiquer

Opérations atomiques

- Si de multiples threads modifient la même case mémoire de manière concurrente:
  - Pas de garanties matérielles du résultat
  - Décision délibérée pour diminuer les coûts
  - 0,001% des cas
- Néanmoins, des opérations atomiques existent

Opérations atomiques

- Bit Test:
  - Écrit un bit de manière atomique
  - Retourne la valeur initiale du bit
- Load Lock/Store Conditional (LL/SC):
  - Le load commence la transaction
  - Le store la termine
  - Le store retourne succès si aucune autre tâche n'a modifié la mémoire entre temps
  - L'opération peut ainsi être retentée

Opérations atomiques

- Compare-and-Swap (CAS):
  - Écrit une valeur de manière conditionnelle
  - Retourne la valeur précédente
  - Utile pour implémenter les spinlocks
- Arithmétique atomique:
  - Utilisable seulement sur x86 x86-64
  - Opérations arithmétiques ou logiques atomiques
  - Pas supporté par les processeurs RISC

Opérations atomiques

- Une architecture supporte LL/SC ou CAS mais pas les deux
- Les approches sont équivalentes
- Elles permettent d'implémenter les opérations arithmétiques atomiques

Opérations atomiques

- Addition atomique:

```
int curval;
int newval;
do {
  curval = var;
  newval = curval + addend;
} while (CAS(&var, curval, newval));
```

```
int curval;
int newval;
do {
  curval = LL(var);
  newval = curval + addend;
} while (SC(var, newval));
```

Opérations atomiques

- Il est important de choisir la bonne implémentation de l'incrément atomique sur x86

Exchange add	Add fetch	CAS
0,23s	0,21s	0,73s

Opérations atomiques

- Pourquoi le CAS est-il plus cher
  - Il y a deux opérations mémoire
  - L'opération CAS est plus compliquée que nécessaire
  - Il est nécessaire d'avoir une boucle en cas d'opérations concurrentes
- Il est possible avec CAS d'implémenter toutes les opérations atomiques
- Mais implémentation non optimale

Opérations atomiques

- Utiles pour l'implémentation des listes « lock-free »
- Non portable
- Il existe des opérations atomiques en OpenMP
- Il existe une bibliothèque : Open Portable Atomics
  - https://trac.mcs.anl.gov/projects/openpa/
  - Utilisé dans MPICH,OpenMPI

Algorithme lock-free

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:     loop                                                    # Keep trying until Dequeue is done
D2:         head = Q->Head                                      # Read Head
D3:         tail = Q->Tail                                      # Read Tail
D4:         next = head->next                                   # Read Head.ptr->next
D5:         if head == Q->Head                                  # Are head, tail, and next consistent?
D6:             if head.ptr == tail.ptr                         # Is queue empty or Tail falling behind?
D7:                 if next.ptr == NULL                         # Is queue empty?
D8:                     return FALSE                            # Queue is empty, couldn't dequeue
D9:                 endif
D10:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)   # Tail is falling behind. Try to advance it
D11:            else                                            # No need to deal with Tail
                    # Read value before CAS, otherwise another dequeue might free the next node
D12:                *pvalue = next.ptr->value
D13:                if CAS(&Q->Head, head, <next.ptr, head.count+1>)   # Try to swing Head to the next node
D14:                    break                                   # Dequeue is done. Exit loop
D15:                endif
D16:            endif
D17:        endif
D18:    endloop
D19:    free(head.ptr)                                          # It is safe now to free the old dummy node
D20:    return TRUE                                             # Queue was not empty, dequeue succeeded
```

Algorithme lock-free

```
enqueue(Q: pointer to queue_t, value: data type)
E1:     node = new_node()                                   # Allocate a new node from the free list
E2:     node->value = value                                 # Copy enqueued value into node
E3:     node->next.ptr = NULL                               # Set next pointer of node to NULL
E4:     loop                                                # Keep trying until Enqueue is done
E5:         tail = Q->Tail                                  # Read Tail.ptr and Tail.count together
E6:         next = tail.ptr->next                           # Read next ptr and count fields together
E7:         if tail == Q->Tail                              # Are tail and next consistent?
E8:             if next.ptr == NULL                         # Was Tail pointing to the last node?
E9:                 if CAS(&tail.ptr->next, next, <node, next.count+1>)  # Try to link node at the end of the linked list
E10:                    break                               # Enqueue is done. Exit loop
E11:                endif
E12:            else                                        # Tail was not pointing to the last node
E13:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)  # Try to swing Tail to the next node
E14:            endif
E15:        endif
E16:    endloop
E17:    CAS(&Q->Tail, tail, <node, tail.count+1>)           # Enqueue is done. Try to swing Tail to the inserted node
```

ACTIVITÉ DE R&D HPC