Programmation multi-thread: POSIX threads et Intel TBB

Module 2

INF8601 Systèmes informatiques parallèles

Michel Dagenais

École Polytechnique de Montréal

Département de génie informatique et génie logiciel

Sommaire

Programmation multi-thread: POSIX threads et Intel TBB

Programmation multi-fil

- Mémoire partagée, tous les fils d'exécution sont dans un même processus et ont accès aux mêmes données.
- Diviser le travail pour utiliser plusieurs coeurs;
- Poursuivre l'utilisation du CPU lors de bloquages d'E/S;
- Simplifier le traitement asynchrone;
- Gestionnaire qui alimente un bassin de travailleurs (thread pool);
- Pipeline avec un fil d'exécution par station;
- Groupe de pairs, division hiérarchique du travail.

Programmation multi-thread: POSIX threads et Intel TBB

La norme POSIX threads

- Unix était initialement fourni avec le code source et plusieurs systèmes d'exploitation compatibles, dérivés ou indépendants, ont été produits.
- Les variantes entre ces systèmes causaient des problèmes de compatibilité, d'où la norme ANSI/IEEE POSIX: Portable Operating System Interface (UniX-like).
- Lorsque les systèmes multi-processeurs sont devenus plus répandus, plusieurs variantes de Unix ont ajouté des fils d'exécution et une interface de programmation pour les mettre en oeuvre.
- Le processus de normalisation POSIX a développé la norme 1003.1 pour les fils d'exécution (pthreads).

ANSI/IEEE POSIX 1003.1: PThreads

- pthread_create (thread,attr,start_routine,arg);
- pthread_exit (status);
- pthread_cancel (thread);
- pthread_attr_init (attr);
- pthread_attr_destroy (attr);
- pthread_join (threadid,status);
- pthread_detach (threadid);
- pthread_attr_setstacksize (attr, stacksize);

PThreads Mutex

- pthread_mutex_init (mutex,attr);
- pthread_mutex_destroy (mutex);
- pthread_mutexattr_init (attr);
- pthread_mutexattr_destroy (attr);
- pthread_mutex_lock (mutex);
- pthread_mutex_trylock (mutex);
- pthread_mutex_unlock (mutex) ;
- Attention à l'ordre de verrouillage!

PThreads Conditions

- pthread_cond_init (condition,attr);
- pthread_cond_destroy (condition);
- pthread_condattr_init (attr);
- pthread_condattr_destroy (attr);
- pthread_cond_wait (condition,mutex);
- pthread_cond_signal (condition);
- pthread_cond_broadcast (condition);

Exemple pthread

```
// exemple_pthreads.c
#define _REENTRANT
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void afficher(int n, char lettre) {
  int i,j;
  for (j=1; j<n; j++) {
    for (i=1; i < 10000000; i++);
    printf("\%c",lettre); fflush(stdout);
  }
}
void *threadA(void *inutilise) {
  afficher(100,'A');
  printf("\n Fin du thread A\n"); fflush(stdout);
  pthread_exit(NULL);
}
```

Exemple pthread (suite)

```
void *threadC(void *inutilise) {
  afficher(150,'C');
  printf("\n Fin du thread C\n"); fflush(stdout);
  pthread_exit(NULL);
}
void *threadB(void *inutilise) {
  pthread_t thC;
  pthread_create(&thC, NULL, threadC, NULL);
  afficher(100,'B');
  printf("\n Le thread B attend la fin du thread C\n");
  pthread_join(thC,NULL);
  printf("\n Fin du thread B\n"); fflush(stdout);
  pthread_exit(NULL);
}
```

Exemple pthread (suite)

```
int main() {
  int i;
  pthread_t thA, thB;

  printf("Creation du thread A");
  pthread_create(&thA, NULL, threadA, NULL);
  pthread_create(&thB, NULL, threadB, NULL);
  sleep(1);
  //attendre la fin des threads
  printf("Le thread principal attend que les autres se terminent
  pthread_join(thA,NULL);
  pthread_join(thB,NULL);
  exit(0);
}
```

Exemple pthread (suite)

```
bash-3.2\$ ./exemple_threads
Creation du thread AACBACBACBACBACBACBACBACBABCABCABCABCABCAB
BCABCABCABCABCABCACBACBACBACB
Le thread principal attend que les autres se terminent
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBA
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBA
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBA
Fin du thread A
CB
Le thread B attend la fin du thread C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Fin du thread C
Fin du thread B
bash-3.2\$
```

Exemple pthread (suite)

```
bash-3.2\$ ./exemple_threads
Creation du thread A
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBA
ACBACBACB
Le thread principal attend que les autres se terminent
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBA
CABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABC
CABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCA
Fin du thread A
B
Le thread B attend la fin du thread C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Fin du thread C
Fin du thread B
bash-3.2\$
```

Programmation multi-thread: POSIX threads et Intel TBB

Introduction

- TBB travaille à un plus haut niveau, en langage C++, et permet de décrire l'algorithme parallèle plutôt que d'assigner les fils d'exécution; souvent plus efficace.
- Indépendant de la plate-forme;
- Basé sur les templates;
- Unité de travail (task) plutôt que travailleur (thread);
- Librairie C++, namespace: tbb;
- Compatible avec PThreads, OpenMP, MPI...

Structure de TBB

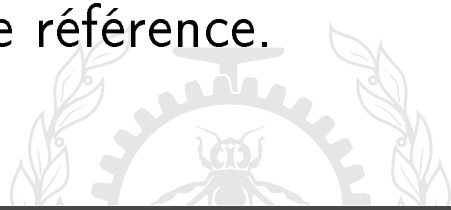
- Algorithmes parallèles;
- Ordonnancement des tâches;
- Structures de données concurrentes;
- Primitives de synchronisation;
- Mesures de temps;
- Allocateur de mémoire;

TBB

- namespace tbb; task_scheduler_init init;
- parallel_for
- parallel_reduce
- parallel_scan
- parallel_do
- pipeline
- parallel_sort

Parallel_for

- Arguments: intervalle 1D ou 2D, objet avec constructeur par copie et opérateur () pour appliquer l'opération;
- parallel_for(blocked_range<size_t>(0,n), MyClosure(a));
- L'objet initial est pris par une tâche pour [0, n/2[et sert à créer une copie passée à une autre tâche pour [n/2, n[.
- La décomposition se poursuit en arbre jusqu'à ce que l'intervalle soit assez petit (simple/grainsize, auto/nb_thread*k, or affinity partitioner);
- A chaque feuille, l'opérateur () est appliqué pour l'intervalle requis;
- Eviter de copier les données, simplement copier une référence.



Concept de partition

- Le partitionneur agit en créant une nouvelle partition qui vole une partie du travail de la partition existante.

```
struct TrivialIntegerRange {
  int lower, upper;
  bool empty() const { return lower==upper; }
  bool is_divisible() const { return upper>lower+1; }
  TrivialIntegerRange( TrivialIntegerRange& r, split ) {
    int m = (r.lower+r.upper)/2;
    lower = m;
    upper = r.upper;
    r.upper = m;
  }
};
```

Exemple de parallel_for

```
struct Average {
  const float* input; float* output;
  void operator()( const blocked_range<int>& range ) const {
    for( int i=range.begin(); i!=range.end(); ++i )
      output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.f);
  }
};

// Entrée [0..n] and Sortie [1..n-1].
void ParallelAverage( float* output, const float* input, siz
  Average avg;
  avg.input = input; avg.output = output;
  parallel_for( blocked_range<int>( 1, n ), avg );
}
```

Parallel reduce

- Arguments: intervalle, objet avec constructeur par division, opérateur () pour appliquer l'opération et méthode join pour la réduction;
- parallel_reduce(blocked_range<size_t>(0,n), MySum(a));
- Décomposition récursive jusqu'à ce que l'intervalle soit assez petit;
- A chaque feuille, l'opérateur () est appliqué pour l'intervalle requis;
- La recomposition cumule (reduce) les résultats obtenus dans chaque feuille (e.g. somme totale, minimum...).

Exemple de parallel_reduce

```
struct Sum {
  float value;
  Sum() : value(0) {}
  Sum(Sum& s, split) { value = 0; }
  void operator()(const blocked_range<float*>& r) {
    float temp = value;
    for(float* a=r.begin(); a!=r.end(); ++a) temp += *a;
    value = temp;
  }
  void join(Sum& rhs) {value += rhs.value;}
};
float ParallelSum(float array[], size_t n) {
  Sum total;
  parallel_reduce(blocked_range<float*>(array, array+n),total);
  return total.value;
}
```

Parallel scan

- Arguments: intervalle, objet avec constructeur par division, opérateur () pour scan initial ou final afin de calculer la réduction ou l'utiliser, méthode reverse_join pour propager la réduction et assign pour l'initialiser;
- parallel_scan(blocked_range<int>(0,n), MyScan(a, b));
- Décomposition récursive jusqu'à ce que l'intervalle soit assez petit;
- A chaque feuille, l'opérateur () est appliqué pour l'intervalle requis, final pour le premier, initial pour les suivants;
- La réduction des intervalles précédents fournit la valeur requise aux suivants pour faire leur passe finale. Le dernier intervalle peut se contenter de la passe finale;
- Utile lorsque plusieurs processeurs sont disponibles.

Exemple de parallel_scan

```
class Body {
  T sum; T* const y; const T* const x;
  Body( T y_[], const T x_[] ) : sum(0), x(x_), y(y_) {}
  T get_sum() const {return sum;}
  template<typename Tag>void operator()(const blocked_range<int>& r, Tag) {
    T temp = sum;
    for( int i=r.begin(); i<r.end(); ++i ) {
      temp = temp + x[i];
      if(Tag::is_final_scan()) y[i] = temp;
    }
    sum = temp;
  }
  Body(Body& b, split) : x(b.x), y(b.y), sum(0) {}
  void reverse_join(Body& a) { sum = a.sum + sum; }
  void assign( Body& b ) {sum = b.sum; }
};
float DoParallelScan(T y[], const T x[], int n) {
  Body body(y,x);
  parallel_scan(blocked_range<int>(0,n), body);
  return body.get_sum();
}
```

Parallel do

- Arguments: premier et dernier éléments (itérateur C++), objet avec constructeur et opérateur ();
- parallel_do(first, last, MyClosure);
- Possibilité d'ajouter des éléments dynamiquement;
- Le parcours de la liste est séquentiel et donc limite la mise à l'échelle.

Exemple de parallel_do

```
class Body {
  Body() {};
  typedef Cell* argument_type;
  void operator()(Cell* c,
      tbb::parallel_do_feeder<Cell*>& feeder) const {
    c->update();
    c->ref_count = ArityOfOp[c->op];
    for( size_t k=0; k<c->successor.size(); ++k ) {
      Cell* successor = c->successor[k];
      if(0 == --(successor->ref_count)) {
        feeder.add( successor );
      }
    }
  }
};

void ParallelPreorderTraversal(const std::vector<Cell*>& root_set) {
  tbb::parallel_do(root_set.begin(), root_set.end(),Body());
}
```

Parallel pipeline

- Créer un pipeline, ajouter en séquence les filtres sériels (serial_in_order, serial_out_of_order) ou parallèles qui le composent; exécuter le pipeline;
- pipeline.add_filter(filter& f);
- pipeline.run(size_t max_number_of_live_token);
- Le type de sortie d'un filtre doit correspondre à l'entrée du suivant, pas d'entrée pour le premier ni de sortie pour le dernier;
- TBB crée plusieurs tâches pour les filtres parallèles et ordonnance au besoin leurs sorties.

Exemple de parallel_pipeline

```
float RootMeanSquare( float* first, float* last ) {
  float sum=0;
  parallel_pipeline(16,
      make_filter<void,float*>(filter::serial,
          [&](flow_control& fc)-> float*{
            if( first<last ) return first++;
            else { fc.stop(); return NULL; }
          }) &
      make_filter<float*,float>(filter::parallel,
          [](float* p){return (*p)*(*p);}) &
      make_filter<float,void>(filter::serial,
          [&](float x) {sum+=x;})
  );
  return sqrt(sum);
}
```

Parallel sort

- Les deux arguments sont des itérateurs avec accès direct, et fonctions de comparaison et d'échange;
- parallel_sort(begin, end);
- Certaines opérations demeurent séquentielles.

Exemple de parallel_sort

```
#include "tbb/parallel_sort.h"
#include <math.h>

using namespace tbb;

const int N = 100000;
float a[N];float b[N];

void SortExample() {
  for( int i = 0; i < N; i++ ) {
    a[i] = sin((double)i);
    b[i] = cos((double)i);
  }

  parallel_sort(a, a + N);
  parallel_sort(b, b + N, std::greater<float>());
}
```

Tâches

- Tâche à réaliser: section de données et fonction à appliquer à chaque élément de donnée;
- Objet qui hérite de class task;
- Gestionnaire de tâches fourni par la librairie:
  - création automatique de fils d'exécution en proportion du nombre de coeurs;
  - chaque fil prend des tâches lorsque libre;
  - équilibrage de la charge entre les fils;

Ordonnancement de tâches

- Les algorithmes parallèles divisent le problème en tâches qui sont alors ordonnancées entre les fils/processeurs.
- Il est aussi possible de spécifier directement les tâches à faire exécuter efficacement en parallèle.
- Chaque fil d'exécution exécute une tâche à la fois, sans préemption.
- Lorsqu'il a fini, il prend la prochaine tâche créée par le même fil, ou à défaut celle d'un autre fil.
- Possibilité d'assigner des priorités aux tâches.

Exemple de tâches

```
#include "tbb/task_group.h"
using namespace tbb;

int Fib(int n) {
  if( n<2 ) { return n; }
  else {
    int x, y;
    task_group g;
    g.run([&]{ x=Fib(n-1); }); // spawn a task
    g.run([&]{ y=Fib(n-2); }); // spawn another task
    g.wait();
    return x+y;
  }
}
```

Exceptions

- L'ordonnancement de tâches de manière asynchrone sur plusieurs fils brise la propagation des exceptions normalement fournies par le C++.
- Les exceptions TBB peuvent être propagées d'un fil à l'autre pour remonter au fil de la tâche parent qui a créé les autres tâches.

Structures de données

- concurrent_hash_map<Key,T,HashCompare>
- concurrent_queue<T>
- concurrent_vector<T>
- Structures qui permettent un accès et ajout concurrent ainsi que l'utilisation d'intervalles (hash_map et vector).
- Les queues permettent de séparer des tâches parallèles mais un pipeline explicite est souvent mieux optimisé.

Exclusion mutuelle

- Les mutex sont utilisés pour créer un verrou (lecture ou écriture) qui supporte acquire and release;
  - mutex
  - recursive_mutex
  - spin_mutex
  - queuing_mutex
  - spin_rw_mutex
  - queuing_rw_mutex
- Attention à l'ordre de verrouillage.

Opérations atomiques

- Opérations atomiques;
- atomic<T>: fetch_and_increment, compare_and_swap, operator++, operator+=,...
- Plus rapide que les verrous, pas d'interblocage.

Horloges

- Classe tick_count pour avoir le temps réel écoulé.
- tick_count t0 = tick_count::now();
- Chaque coeur a son horloge, la fréquence peut changer dynamiquement, tick_count compense pour ces effets et convertit en secondes à la demande;
- Temps écoulé et non temps CPU.

Allocation de mémoire

- scalable_allocator<T>
- Allocation parallèle efficace, e.g. bloc de mémoire libre par processeur ou fil d'exécution;
- cache_aligned_allocator<T>
- Allocation alignée sur le début d'un nouveau bloc de cache; évite que l'accès concurrent à deux objets distincts mais proches ne cause de problème de contention pour un même bloc de cache.

Programmation multi-thread: POSIX threads et Intel TBB

Conclusion

- TBB est plus moderne et mieux intégré à C++ que des solutions comme OpenMP.
- Son utilisation est moins répandue que OpenMP.
- La plupart des opérations se font avec une organisation arborescente très efficace.
- Pthread permet un contrôle précis à bas niveau des fils d'exécution et son utilisation est très répandue.