Paralelní a distribuované výpočty (B4B36PDV)

Branislav Bošanský, Michal Jakob

bosansky@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Dnešní přednáška

Motivace

Dnešní přednáška

Nástroje

Dnešní přednáška

Vlákna

Vlákna – úvod, spuštění

V C++11 (a dalších)

- V rámci C++ exportovány hlavičkou <thread>
  - std::thread
    - první argument je funkce, kterou bude vlákno vykonávat
    - ostatní argumenty jsou argumenty dané funkce

```
#include <iostream>
#include <thread>
#include <vector>

const int thread_count = 10;
void Hello(long my_rank);

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;
    for (int thread=0; thread < thread_count; thread++) {
        threads.push_back(std::thread(Hello, thread));
    }
    std::cout << "Hello from the main thread\n";
    for (int thread=0; thread < thread_count; thread++) {
        threads[thread].join();
    }

    return 0;
}
```

```
void Hello(long my_rank) {
    std::cout << "Hello from thread " << my_rank << " of "
<< thread_count << std::endl;
}
```

Vlákna - ukončení

- V rámci C++ exportovány hlavičkou <thread>
  - std::thread
    - hlavní vlákno musí zavolat buď metodu **join**
    - případně **detach**
      - měli bychom kontrolovat, jestli dané vlákno ještě existuje
      - if (threads[thread].joinable()) ...

Přístup ke sdílené paměti

Synchronizace vláken

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyvářím histogram zbytků po dělení čísel:

```
void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {
            std::vector<int> local(PARTS);
            for (int i=thread; i<SIZE; i += thread_count) {
                 local[vector[i] % PARTS]++;
            }

            for (int i=0; i<PARTS; i++) {
                 histogram[i] += local[i];
            }
}
```

Přístup ke sdílené paměti

Synchronizace vláken

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyvářím histogram zbytků po dělení čísel:

```
void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {
          std::vector<int> local(PARTS);
          for (int i=thread; i<SIZE; i += thread_count) {
               local[vector[i] % PARTS]++;
          }

          for (int i=0; i<PARTS; i++) {
               histogram[i] += local[i];
          }
}
```

- std::thread předá referenci pomocí std::ref(histogram)

Přístup ke sdílené paměti

Synchronizace vláken - zámky

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyvářím histogram zbytků po dělení čísel:
- Mutex – první řešení

```
std::mutex hist_mutex;

void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {
          std::vector<int> local(PARTS);
          for (int i=thread; i<SIZE; i += thread_count) {
               local[vector[i] % PARTS]++;
          }

          hist_mutex.lock();
          for (int i=0; i<PARTS; i++) {
               histogram[i] += local[i];
          }
          hist_mutex.unlock();
}
```

Přístup ke sdílené paměti

Synchronizace vláken - zámky

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyvářím histogram zbytků po dělení čísel:
- Mutex – druhé řešení

```
std::vector<std::mutex> hist_part_mutex(PARTS);

void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {
            std::vector<int> local(PARTS);
            for (int i=thread; i<SIZE; i += thread_count) {
                local[vector[i] % PARTS]++;
            }

            for (int i=0; i<PARTS; i++) {
                hist_part_mutex[i].lock();
                histogram[i] += local[i];
                hist_part_mutex[i].unlock();
            }
}
```

Přístup k více proměnným

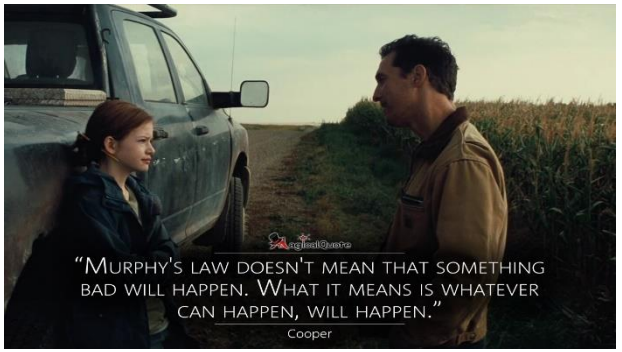
- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
  - např. chci mezivýsledek z operací, které se provádějí
  - zamknu 1., zamknu 2.

Přístup k více proměnným

- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
  - např. chci mezivýsledek z operací, které se provádějí
  - zamknu 1., zamknu 2.
    - možný deadlock!

Přístup k více proměnným

- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
  - např. chci mezivýsledek z operací, které se provádějí
  - zamknu 1., zamknu 2.
    - možný deadlock!



- musíme zamknout oba zámky současně

```
void thread_operation(){
    std::lock(mutex1,mutex2);
     ...
    complicated_task();
     ...
    mutex1.unlock();
    mutex2.unlock();
}
```

Deadlock

- Deadlocky mohou vzniknout pokud:
  1. Každý zámek může vlastnit maximálně jedno vlákno
  2. Vlákno aktuálně vlastní (má zamčený) alespoň jeden zámek a požaduje zamknout alespoň jeden další
  3. Není možné odebrat vlastnictví zámku
  4. Existuje cyklická závislost mezi vlákny

Deadlock vznikne velice snadno

Deadlock

- Deadlocky mohou vzniknout pokud:
  1. Každý zámek může vlastnit maximálně jedno vlákno
  2. Vlákno aktuálně vlastní (má zamčený) alespoň jeden zámek a požaduje zamknout alespoň jeden další
  3. Není možné odebrat vlastnictví zámku
  4. Existuje cyklická závislost mezi vlákny

Deadlock vznikne velice snadno

https://goo.gl/a6BEMb

Odemykání zámků

- Zamknuté zámky je dobré odemykat
- A co v případě výjimky?
  - Musíme zajistit odemknutí při všech možných ukončeních

```
void operation(){
    mutex.lock();
    try {
            ...
        complicated_task();
            ...
    } catch (std::string e){
        mutex.unlock();
        throw e;
    }
    mutex.unlock();
}
```

Automatická správa zámků

- Lock_guard mutex – RAII správa zámků (Resource acquisition is initialization)
  - Konstruktor automaticky volá lock() na zámku, destruktor zámek odemyká

```
void operation(){
    std::lock_guard<std::mutex> guard(mutex);
    try {
            ...
        complicated_task();
            ...
    } catch (std::string e){
        throw e;
    }
}
```

  - A co když chceme mít v lock_guard 2 zámky?

Automatická správa zámků

Vícero proměnných

- Lock_guard mutex
  - Konstruktor automaticky volá lock() na zámku, destruktor zámek odemyká
  - A co když chceme mít v lock_guard 2 zámky?

```
std::mutex m1;
std::mutex m2;

void f(int id) {
    std::lock(m1, m2);
    std::lock_guard<std::mutex> lock1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> lock2(m2, std::adopt_lock);
    std::cout << "Thread " << id << " says hi." << std::endl;
//  we do not need to unlock
}

int main(int argc, char* argv[]) {
    std::thread t1(f, 1);
    std::thread t2(f, 2);

    t1.join();
    t2.join();
}
```

Časově omezené čekání

- Pokud nechceme, aby se vlákno při pokusu o zamknutí zámku zablokovalo, můžeme využít časově omezených metod
  - časové zámky (timed_mutex)

```
std::timed_mutex m;
const int THREADS = 10;
const std::chrono::milliseconds timeout(100);
const std::chrono::milliseconds timeout2(20);

void f(int id) {
    if (m.try_lock_for(timeout)) {
        std::cout << "Thread " << id << " is computing stuff." << std::endl;
        std::this_thread::sleep_for(timeout2);
        m.unlock();
    } else {
        std::cout << "Thread " << id << " is skipping." << std::endl;
    }
}

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;

    for (int i=0; i<THREADS; i++)
        threads.push_back(std::thread(f, i));

    for (int i=0; i<THREADS; i++)
        threads[i].join();
}
```

Opakované zamykání

- Co když už máme naimplementovaných několik thread-safe metod, které bychom chtěli zavolat z jiné thread-safe metody?
  - Mějme operace nad prvky matice add, divide

```
void add_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::mutex> l(mutexes[row][column]);
    matrix[row][column] += value;
}

void divide_by_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::mutex> l(mutexes[row][column]);
    matrix[row][column] = matrix[row][column] / value;
}
```

  - Když bychom měli jinou metody, která tyto metody volá, zámky již budou zamknuté …

Opakované zamykání

- Co když už máme naimplementovaných několik thread-safe metod, které bychom chtěli zavolat z jiné thread-safe metody?
  - Mějme operace nad prvky matice add, divide

```
void add_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::mutex> l(mutexes[row][column]);
    matrix[row][column] += value;
}

void divide_by_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::mutex> l(mutexes[row][column]);
    matrix[row][column] = matrix[row][column] / value;
}
```

  - Když bychom měli jinou metody, která tyto metody volá, zámky již budou zamknuté …
  - Můžeme použít **recursive_mutex**

Recursive Lock

- recurisve_lock

```
void add_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::recursive_mutex> l(mutexes[row][column]);
    matrix[row][column] += value;
}

void divide_by_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::recursive_mutex> l(mutexes[row][column]);
    matrix[row][column] = matrix[row][column] / value;
}

//there is a one dimension padding in the matrix so that we do not need to check boundaries
void average_with_neighbours(std::vector<std::vector<int>>& matrix, int row, int column) {
    std::lock(mutexes[row][column-1],mutexes[row][column],mutexes[row][column+1],mutexes[row-
1][column],mutexes[row+1][column]);
    std::cout << "thread is averaging " << row << "," << column << " old_value: " << matrix[row][column];
    int v = (matrix[row][column-1] + matrix[row][column+1] + matrix[row-1][column] + matrix[row+1][column])/8;
    divide_by_number(matrix,row,column,2);
    add_number(matrix,row,column,v);
    std::cout << " new_value: " << matrix[row][column] << std::endl;
    for (int i=-1;i<=1;i++)
        for (int j=-1; j<=1; j++) {
            if (i*j != 0) continue;
            mutexes[row+i][column+j].unlock();
        }
}
```

Atomické proměnné

- Zámky znamenají práci navíc
- Někdy není nutné je použít
  - **Atomické proměnné** umožňují provedení atomických operací, limitují možné optimalizace kompilátoru
  - umožňují základní operace bez zámků – typicky rychlejší
  - **atomic<type>** (např. atomic<int> (=atomic_int), …)
- Vraťme se k histogramu

```
void hist_atomic(std::vector<int>& vector, int thread, std::vector<std::atomic_int>& histogram) {
    std::vector<int> local(PARTS);
    for (int i=thread; i<SIZE; i += thread_count) {
        local[vector[i] % PARTS]++;
    }

    for (int i=0; i<PARTS; i++) {
        histogram[i] += local[i];
    }
}
```

- při současném přístupu k vícero proměnným musíme použít zámky

Zpracování výsledků z jiných vláken

- Při inicializaci přes std::thread, vlákna nic nevracejí
  - dílčí výsledky jsou předávané přes sdílené proměnné
- Co když chceme vytvořit několik vláken pro pomocné úkoly a jejich výsledky zpracovat?
  - struktura future a metoda std::async

```
#include <thread>
#include <future>
#include <iostream>

int foo() {
    std::cout << "I'm a thread" << std::endl;
    return 42;
}

int main(){
    auto future = std::async(std::launch::async, foo);
    std::cout << future.get();
    return 0;
}
```

- future.get() blokuje aktivní vlákno a čeká na výsledek
- lze volat pouze jednou
- future.wait() čeká, ale nezkonzumuje výsledek

Zpracování výsledků z jiných vláken (2)

- Destruktor std::future vytvořené pomocí std::async je blokující

```
std::async(std::launch::async, foo);
std::async(std::launch::async, foo2);
```

  - funkce foo2 se spustí pouze po skončení vlákna s metodou foo
  - pokud v metodě zavoláme

```
auto future = std::async(std::launch::async, foo);
```

    - při ukončení metody se bude čekat na ukončení vláken

Zpracování výsledků z jiných vláken (2)

- Destruktor std::future vytvořené pomocí std::async je blokující

```
std::async(std::launch::async, foo);
std::async(std::launch::async, foo2);
```

  - funkce foo2 se spustí pouze po skončení vlákna s metodou foo
  - pokud v metodě zavoláme

```
auto future = std::async(std::launch::async, foo);
```

    - při ukončení metody se bude čekat na ukončení vláken

- Futures také umožňují časově-omezené čekání na výsledek
  - Má smysl pokud hlavní vlákno aktivně pracuje a průběžně kontroluje dostupnost dílčích výsledků z asynchronně puštěných vláken

Zpracování výsledků z jiných vláken (3)

- Futures také umožňují časově-omezené čekání na výsledek

```
#include <thread>
#include <future>
#include <iostream>
#include <chrono>

int function(int duration) {
    std::this_thread::sleep_for(std::chrono::seconds(duration));
    return duration*4-2;
}

int main(){
    auto f1 = std::async(std::launch::async, function, 5);
    auto f2 = std::async(std::launch::async, function, 3);

    auto timeout = std::chrono::nanoseconds(10);
    while(f1.valid() || f2.valid()){
        if(f1.valid() && f1.wait_for(timeout) == std::future_status::ready){
            std::cout << "Task1 is done with result " << f1.get() << std::endl;
        }
        if(f2.valid() && f2.wait_for(timeout) == std::future_status::ready){
            std::cout << "Task2 is done with result " << f2.get() << std::endl;
        }
        std::cout << "Work in the main thread." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }

    return 0;
}
```

Podmínkové proměnné

- Podmínkové proměnné (Condition variables) slouží ke komunikaci mezi vlákny
  - např. vlákno chce předat zprávu „dílčí výsledek je již připraven “
  - typický příklad – fronta úkolů ke zpracování (Producer-Consumer)
  - 1 (nebo víc) vlákno generuje úkoly
  - další vlákna je zpracovávají
    - zpracovávající vlákna musí dostat notifikaci o tom, že další úkol je připraven ke zpracování
    - podmíněná proměnná navázaná na zámek fronty úkolů
  - nechť má fronta úkolů omezenou velikost (např. aby nám nepřetekla paměť)
  - čekající vlákno je notifikováno metodou notify_one() (případně notify_all())

Podmínkové proměnné

```
int front = 0;
int rear = 0;
int count = 0;

std::vector<std::pair<int,int>> buffer;
std::mutex lock;
std::condition_variable not_full;
std::condition_variable not_empty;

void add_task(int row, int column){
            std::unique_lock<std::mutex> l(lock);
            not_full.wait(l, [this](){return count != MAXPOOL; });

            buffer[rear] = std::pair<int,int>(row,column);
            rear = (rear + 1) % MAXPOOL;
            count++;
            l.unlock();
            not_empty.notify_one();
}

std::tuple<int,int,int> execute_task(){
            std::unique_lock<std::mutex> l(lock);

            not_empty.wait(l, [this](){return count != 0; });

            std::pair<int,int> square = buffer[front];
            front = (front + 1) % MAXPOOL;
            count--;

            l.unlock();
            not_full.notify_one();

            int result = average_with_neighbours(*matrix, square.first, square.second);
            return std::tuple<int,int,int>(result,square.first,square.second);
}
```

Budoucnost

V C++17 (a dalších)

- Další změny a podpora paralelismu v C++17 (finální standard schválen v prosinci 2017)
  - …
  - Parallel versions of STL algorithms
  - …