Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček
jakub.marecek@fel.cvut.cz

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

Dnes částečně s využitím materiálů doc. Rychty
https://cw.fel.cvut.cz/old/_media/courses/a7b36pjc/prednasky/pjc-11-vlakna.pdf

Předchozí přednáška

- Paralelní část
  - Paralelní programování jednoduchých algoritmů
  - Vliv různých způsobů paralelizace na rychlost výpočtu
- Distribuovaná část
  - Problémy v distribuovaných systémech (shoda, konzistence dat)
  - Navržení robustných řešení

- CourseWare
  - https://cw.fel.cvut.cz/wiki/courses/b4b36pdv/start
- Nebudeme měnit rozvrh cvičení.

Dnešní přednáška

Vlákna v C++11 (a dalších)

- Principy: synchronizace, vlákna, atomické proměnné, zámky, RAII.
  Viz také učebnice.
- Praktické použití v C++: hlavičky <thread> a <future>. Viz také materiály doc. Rychty
  https://cw.fel.cvut.cz/old/_media/courses/a7b36pjc/prednasky/pjc-11-vlakna.pdf
  a výborná doporučení:
  https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html
- Finesy moderního C++. Zatím špatně dokumentované.

Bajka

V učebnici je zejména první kapitola velmi snadno čitelná

1.2 A Fable

Instead of treating coordination problems (such as mutual exclusion) as programming exercises, we prefer to think of concurrent coordination problems as if they were physics problems. We now present a sequence of fables, illustrating some of the basic problems. Like most authors of fables, we retell stories mostly invented by others (see the Chapter Notes at the end of this chapter).

Alice and Bob are neighbors, and they share a yard. Alice owns a cat and Bob owns a dog. Both pets like to run around in the yard, but (naturally) they do not get along. After some unfortunate experiences, Alice and Bob agree that they should coordinate to make sure that both pets are never in the yard at the same time. Of course, we rule out trivial solutions that do not allow any animals into an empty yard.

How should they do it? Alice and Bob need to agree on mutually compatible procedures for deciding what to do. We call such an agreement a *coordination protocol* (or just a *protocol*, for short).

The yard is large, so Alice cannot simply look out of the window to check whether Bob's dog is present. She could perhaps walk over to Bob's house and knock on the door, but that takes a long time, and what if it rains? Alice might lean out the window and shout "Hey Bob! Can I let the cat out?" The problem is that Bob might not hear her. He could be watching TV, visiting his girlfriend, or out shopping for dog food. They could try to coordinate by cell phone, but the same difficulties arise if Bob is in the shower, driving through a tunnel, or recharging his phone's batteries.

Alice has a clever idea. She sets up one or more empty beer cans on Bob's windowsill (Fig. 1.4), ties a string around each one, and runs the string back to her house. Bob does the same. When she wants to send a signal to Bob, she yanks the string to knock over one of the cans. When Bob notices a can has been knocked over, he resets the can.

Up-ending beer cans by remote control may seem like a creative solution, but it is still deeply flawed. The problem is that Alice can place only a limited number of cans on Bob's windowsill, and sooner or later, she is going to run out of cans to knock over. Granted, Bob resets a can as soon as he notices it has been knocked over, but what if he goes to Cancún for Spring Break? As long as Alice relies on Bob to reset the beer cans, sooner or later, she might run out.

So Alice and Bob try a different approach. Each one sets up a flag pole, easily visible to the other. When Alice wants to release her cat, she does the following:

1. She raises her flag.
2. When Bob's flag is lowered, she unleashes her cat.
3. When her cat comes back, she lowers her flag.

Bob's behavior is a little more complicated.

1. He raises his flag.
2. While Alice's flag is raised
   a) Bob lowers his flag
   b) Bob waits until Alice's flag is lowered
   c) Bob raises his flag
3. As soon as his flag is raised and hers is down, he unleashes his dog.
4. When his dog comes back, he lowers his flag.

Přístup k více proměnným

Riziko souběhu, riziko uváznutí, …

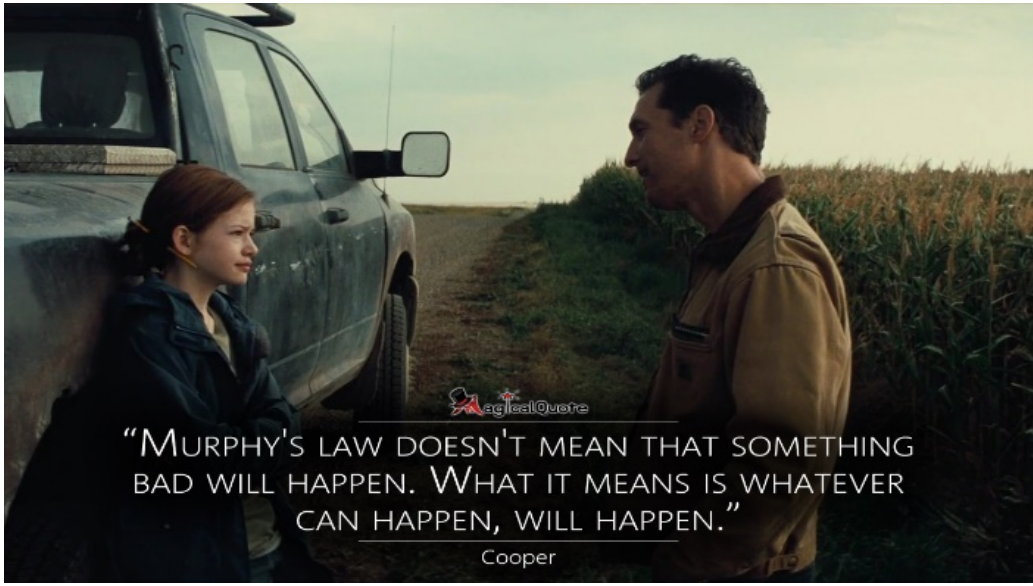
- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
  - např. chci mezivýsledek z operací, které se provádějí
  - zamknu 1., zamknu 2.

Přístup k více proměnným

Riziko souběhu, riziko uváznutí, …

- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
  - např. chci mezivýsledek z operací, které se provádějí
  - zamknu 1., zamknu 2.
    - Možné uváznutí!
  - musíme zamknout oba zámky současně

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



Uváznutí (Deadlock)

- Uváznutí (deadlock) může nastat 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

Uváznutí nastane velice snadno

Vlákna

V C++11 (a dalších)

- **std::thread** exportována hlavičkou <thread>
- první argument konstruktoru 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) {
    std::cout << "Hello from thread " << my_rank << " of " << thread_count << std::endl;
}
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";
    for (int thread = 0; thread < thread_count; thread++) {
        threads[thread].join();
    }
    return 0;
}
```

Vlákna

- **std::thread** exportována hlavičkou <thread>
- Vlákno začně běžet bezprostředně po zavolání konstruktoru
- Objekt není možné kopírovat (není **CopyConstructible, CopyAssignable**),
- Pro *ukončení* hlavní vlákno musí zavolat buď metodu **join,** případně **detach**, je ale potřeba kontrolovat, jestli dané vlákno ještě existuje pomocí metody **joinable**
  - if (threads[thread].joinable()) …

- pokud dojde vyvolání výjimky, celá aplikace může být ukončena přes **std::terminate,** je tedy třeba **std::promise**

Vlákna

- **std::thread** exportována hlavičkou <thread>
- Metoda **join** čeká na skončení vlákna, a následně zajistí návratovou hodnotu **joinable** false
- Metoda **detach** vlákno odpojí od hlavního vlákna a zajistí návratovou hodnotu **joinable** false
- V případě volání na vláknu, které není joinable, vyhodí výjimku **std::system_error**

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

using namespace std::this_thread;

void A() {
  std::cout << "a";
  sleep_for(std::chrono::seconds(5));
  std::cout << "A";
}

void B() {
  std::cout << "b";
  sleep_for(std::chrono::seconds(1));
  std::cout << "B";
}

void C() {
  std::cout << "c";
  std::thread t(A);
  t.detach();
  std::thread u(B);
  u.join();
  std::cout << "C";
}

int main() {
  C();
  std::thread t(B);
  t.join();
  A();
}
```

Přístup ke sdílené paměti

Synchronizace vláken a riziko souběhu

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k jinému než očekávanému výsledku

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

int i = 2;

void decrement() { i = i - 1; }

void print() {
    int j;
    do { j = i; } while (j == 0);
    std::cout << j << std::endl;
}

int main() {
    std::thread first(decrement);
    std::thread second(decrement);
    std::thread third(print);
    first.join();
    second.join();
    third.join();
    return 0;
}
```

Proč je tento příklad problematický?

Přístup ke sdílené paměti

Synchronizace vláken a riziko souběhu

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k jinému než očekávanému výsledku

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

int i = 2;

void decrement() { i = i - 1; }

void print() {
    int j;
    do { j = i; } while (j == 0);
    std::cout << j << std::endl;
}

int main() {
    std::thread first(decrement);
    std::thread second(decrement);
    std::thread third(print);
    first.join();
    second.join();
    third.join();
    return 0;
}
```

Možná:
Obě vlákna načtou 2,
pak obě zapíší 1

Přístup ke sdílené paměti

Synchronizace vláken a riziko souběhu

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k jinému než očekávanému výsledku

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

int i = 2;

void decrement() { i = i - 1; }

void print() {
    int j;
    do { j = i; } while (j == 0);
    std::cout << j << std::endl;
}

int main() {
    std::thread first(decrement);
    std::thread second(decrement);
    std::thread third(print);
    first.join();
    second.join();
    third.join();
    return 0;
}
```

Možná:
Obě vlákna korektně odečtou, ale pak uváznou.

Atomické proměnné

- **Atomické proměnné** umožňují provedení atomických operací,
  - umožňují základní operace bez zámků – typicky rychlejší
  - **atomic<type>** (např. atomic<int> (=atomic_int), …)
  - limitují možné optimalizace kompilátoru

```
#include <iostream>
#include <atomic>
#include <thread>

std::atomic<int> i(0);

void set(int j) { i.store(j, std::memory_order_relaxed); }

void print() {
   int j;
   do { j = i.load(std::memory_order_relaxed); }
   while (j == 0);
   std::cout << j << std::endl;
}

int main() {
  std::thread first(print);
  std::thread second(set, 1); first.join();
  second.join();
  return 0;
}
```

Atomické proměnné

- **Atomické proměnné** umožňují provedení atomických operací,
  - umožňují základní operace bez zámků – typicky rychlejší
  - **atomic<type>** (např. atomic<int> (=atomic_int), …)
  - limitují možné optimalizace kompilátoru

```
#include <iostream>
#include <atomic>
#include <thread>

std::atomic<int> i(0);

void set(int j) { i.store(j, std::memory_order_relaxed); }

void print() {
   int j;
   do { j = i.load(std::memory_order_relaxed); }
   while (j == 0);
   std::cout << j << std::endl;
}

int main() {
  std::thread first(print);
  std::thread second(set, 1); first.join();
  second.join();
  return 0;
}
```

Novice: What is a good prefix for global variables?
Expert: //

Přístup ke sdílené paměti

Synchronizace vláken - zámky

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích jinému než očekávanému výsledku
- Při současném přístupu k vícero proměnným musíme použít zámky
- 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();
}
```

Bjarne Stroustrup and Herb Sutter píší, že tohle je špatný nápad:
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-raii

Přístup ke sdílené paměti

Synchronizace vláken - zámky

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k 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();
          }
}
```

Bjarne Stroustrup and Herb Sutter píší, že tohle je špatný nápad:
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-raii

Automatická správa zámků

Vícero proměnných

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

```
std::mutex m1;

void f(int id) {
    std::lock(m1);
    std::lock_guard<std::mutex> lock1(m1, 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();
}
```

Automatická správa zámků

Vícero proměnných

- Lock_guard mutex
  - RAII správa zámků (Resource acquisition is initialization)
  - 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

- recursive_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();
        }
}
```

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 (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);
}
```

Ladění

V C++11 (a dalších)

- Metoda thread::hardware_concurrency vrací počet hardwarových threadů
- shared_ptr není vhodné, pokud člověk do detailu nerozumí šablonovým funkcím **atomic** https://en.cppreference.com/w/cpp/memory/shared_ptr/atomic
- Výjimky nejsou vhodné.
- Existují kvalitní nástroje, např. https://clang.llvm.org/docs/ThreadSanitizer.html https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

Vlákna

V C++17 (a dalších)

- Podpora paralelismu ve standardní šablonové knihovně STL v C++17
  - Hlavička **execution** definuje objekty
    std::execution::seq
    std::execution::par
    std::execution::par_unseq
    které jde předat jako první parametr standardním algoritmům.
  - std::vector<int> my_data;
    std::sort(std::execution::par,my_data.begin(),my_data.end());
  - Stále není snadné používat výjimky.

Vlákna

V C++17 (a dalších)

- Podpora paralelismu ve standardní šablonové knihovně STL v C++17
- Pěkný příklad z https://www.osti.gov/servlets/purl/1644821
  používá také lambda výrazy, auto typování, atp.

```
void matrix_multiply(
        const MyMatrix<double>& a,
        const MyMatrix<double>& b,
        MyMatrix<std::atomic<double>>& c
) {
    auto start = boost:make_counting_iterator(0);
    auto end = boost:make_counting_iterator(a.rows() * b.cols() * a.cols());
    auto i_stride = b.cols() * a.cols();
    auto j_stride = a.cols();
    std::for_each(
            std::par_unseq, start, end,
            [&](int idxs) {
                auto i = idxs / i_stride;
                auto j = (idxs % i_stride) / j_stride;
                auto k = idxs % j_stride;
                c(i, j) += a(i, k) * b(k, j);
            }
    ) ;
}
```

Vlákna

V C++20

- jthread ("joining thread")
- automaticky se spustí ("join") v destruktoru.

```
#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::jthread(Hello, thread));
   }

   std::cout << "Hello from the main thread\n";

   return 0;
}

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

Vlákna

V C++20

- jthread ("joining thread")
- Automaticky se čeká na dokončení ("join") v destruktoru.
- RAII správa (Resource acquisition is initialization)
- Je možné jej požádat o ukončení (pomocí "std::stop_token") ale není možné to vynutit)

Vlákna

V C++

Shrnutí:

- C++11: náročné
- C++17: některé algoritmy snadné
- C++20: obecně méně náročné, ale zatím nepodporované

- Pochopit logiku C++ plně vyžaduje další studium, např.: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-concurrency
- Existují kvalitní nástroje, např. https://clang.llvm.org/docs/ThreadSanitizer.html https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
- Na zkoušce budeme pracovat s `<thread>` jen v teoretické části.