

IB109 Návrh a implementace paralelních systémů

Pokročilá rozhraní pro implementaci paralelních aplikací

Jiří Barnat

Nevýhody POSIX Threads a Lock-free přístupu

- Na příliš nízké úrovni
- Vhodné pro systémové programátory
- „Příliš složitý přístup na řešení jednoduchých věcí.“

Co bychom chtěli

- Paralelní konstrukce na úrovni programovacího jazyka
- Prostředek vhodný pro aplikační programátory
- Snadné vyjádření běžně používaných paralelních konstrukcí

OpenMP

Myšlenka

- Programátor specifikuje co chce, ne jak se to má udělat.
- Náznak deklarativního přístupu v imperativním programování.

Realizace

- Programátor informuje překladač o zamýšlené paralelizaci uvedením **značek ve zdrojovém kódu** a označením bloků.
- Při překladu překladač sám doplní nízkoúrovňovou realizaci paralelizace.

OpenMP nabízí

- Pragma direktivy překladače
 - #pragma omp direktiva [seznam klauzulí]
- Knihovní funkce
- Proměnné prostředí

Překlad kódu

- Překladač podporující standard OpenMP
 - při překladu pomocí GCC je nutná volba `-fopenmp`
 - `g++ -fopenmp myapp.c`
- Podporováno nejpoužívanějšími překladači (i Visual C++)
- Možno přeložit do sekvenčního kódu

WWW

- <http://www.openmp.org>

Direktiva parallel – příklad v C++

```
1  #include <omp.h>
2  main ()
3  {
4      int nthreads, tid;
5      #pragma omp parallel private(tid)
6      {
7          tid = omp_get_thread_num();
8          printf("Hello World from thread = %d\n", tid);
9          if (tid == 0)
10         {
11             nthreads = omp_get_num_threads();
12             printf("Number of threads = %d\n", nthreads);
13         }
14     }
15 }
```

Použití

- Strukturovaný blok, tj. `{...}`, následující za touto direktivou se provede paralelně.
- Mimo paralelní bloky se kód vykonává sekvenčně.
- Vlákno, které narazí na tuto direktivu se stává hlavním vláknem (master) a má identifikaci vlákna rovnou 0.

Podmíněné spuštění

- Klauzule: `if` (výraz typu `bool`)
- Vyhodnotí-li se výraz na `false` direktiva `parallel` se ignoruje a následující blok je proveden pouze v jedné kopii.

Stupeň paralelismu

- Počet vláken.
- Přednastavený počet specifikován proměnnou prostředí.
- Klauzule: `num_threads` (výraz typu `int`)

Klauzule: private (seznam proměnných)

- Vyjmenované proměnné se zduplikují a stanou se lokální proměnné v každém vlákně.

Klauzule: firstprivate (seznam proměnných)

- Viz private s tím, že všechny kopie proměnných jsou inicializované hodnotou originální kopie.

Klauzule: shared (seznam proměnných)

- Vyjmenované proměnné budou explicitně existovat pouze v jedné kopii.
- Přístup ke sdíleným proměnným nutno serializovat.

Klauzule: default ([shared|none])

- shared: všechny proměnné jsou sdílené, pokud není uvedeno jinak.
- none: vynucuje explicitní uvedení každé proměnné v klauzuli private nebo v klauzuli shared.

Klauzule: reduction (operátor: seznam proměnných)

- Při ukončení paralelního bloku jsou vyjmenované privátní proměnné zkombinovaný pomocí uvedeného operátoru.
- Kopie uvedených proměnných, které jsou platné po ukončení paralelního bloku, jsou naplněny výslednou hodnotou.
- Proměnné musejí být skalárního typu (nesmí být pole, struktury, atp.).
- Použitelné operátory: +, *, -, &, |, ^, && a ||

Použití

- Iterace následujícího for-cyklu budou provedeny paralelně
- Musí být použito v rámci bloku za direktivou `parallel` (jinak proběhne sekvenčně).
- Možný zkrácený zápis: `#pragma omp parallel for`

Klauzule: `private`, `firstprivate`, `reduction`

- Stejně jako pro direktivu `parallel`.

Klauzule: `lastprivate`

- Hodnota privátní proměnné ve vláknu zpracovávající poslední iteraci for cyklu je uložena do kopie proměnné platné po skončení cyklu.

Klauzule: ordered

- Bloky označené direktivou `ordered` v těle paralelně prováděného cyklu jsou provedeny v tom pořadí, v jakém by byly provedeny sekvenčním programem.
- Klauzule `ordered` je povinná, pokud tělo cyklu obsahuje `ordered` bloky.

Klauzule: `nowait`

- Jednotlivá vlákna se nesynchronizují po provedení cyklu.

Klauzule: `schedule` (typ plánování[, velikost])

- Určuje jak budou iterace rozděleny/mapovány mezi vlákna.
- Implicitní plánování je závislé na implementaci.

static

- Iterace cyklu rozděleny do bloků o specifikované velikosti.
- Bloky staticky namapovány na vlákna (round-robin).
- Pokud není uvedena velikost, iterace rozděleny mezi vlákna rovnoměrně (pokud je to možné).

dynamic

- Bloky iterací cyklu v počtu specifikovaném parametrem velikost přidělovány vláknům na žádost, tj. v okamžiku, kdy vlákno dokončilo předchozí práci.
- Výchozí velikost bloku je 1.

guided

- Bloky iterací mají velikost proporcionální k počtu nezpracovaných iterací poděleným počtem vláken.
- Specifikována velikost k , udává minimální velikost bloku (výchozí hodnota 1).
- Příklad:
 - $k = 7$, 200 volných iterací, 8 vláken
 - Velikosti bloků: $200/8=25$, $175/8=21$, ..., $63/8 = 7$, ...

runtime

- Typ plánování určen až za běhu proměnnou `OMP_SCHEDULE`.

Použití

- Strukturované bloky, každý označený direktivou `section`, mohou být v rámci bloku označeným direktivou `sections` provedeny paralelně.
- Možný zkrácený zápis `#pragma omp parallel sections`
- Umožňuje definovat různý kód pro různá vlákna.

Klauzule: `private`, `firstprivate`, `reduction`, `nowait`

- Stejně jako v předchozích případech

Klauzule: `lastprivate`

- Hodnoty privátních proměnných v poslední sekci (dle zápisu kódu) budou platné po skončení bloku `sections`.

Direktiva sections – příklad

```
1  #include <omp.h>
2  main ()
3  {
4      #pragma omp parallel sections
5      {
6          #pragma omp section
7          {
8              printf("Thread A.");
9          }
10         #pragma omp section
11         {
12             printf("Thread B.");
13         }
14     }
15 }
```

Nevnořený paralelismus

- Direktiva `parallel` určuje vznik oblasti paralelního provádění.
- Direktivy `for` a `sections` určují jak bude práce mapována na vlákna vzniklé dle rodičovské direktivy `parallel`.

Vnořený paralelismus

- Při nutnosti paralelismu v rámci paralelního bloku, je třeba znovu uvést direktivu `parallel`.
- Vnořování je podmíněné nastavením proměnné prostředí `OMP_NESTED` (hodnoty `TRUE`, `FALSE`).
- Typické použití: vnořené `for`-cykly
- Obecně je vnořování direktiv v OpenMP poměrně komplikované, nad rámec tohoto tutoriálu.

Bariéra

- Místo, které je dovoleno překročit, až když k němu dorazí všechna ostatní vlákna.
- Direktiva bez klauzulí, tj. `#pragma omp barrier`.
- Vztahuje se ke strukturálně nejbližší direktivě `parallel`.
- Musí být voláno všemi vlákny v odpovídajícím bloku direktivy `parallel`.

Poznámka ke kódování

- Direktivy překladače nejsou součástí jazyka.
- Je možné, že v rámci překladu bude vyhodnocen blok, ve kterém je umístěna direktiva bariéry, jako neproveditelný blok a odpovídající kód nebude ve výsledném spustitelném souboru vůbec přítomen.
- Direktivu `barrier`, je nutné umístit v bloku, který se bezpodmínečně provede (zodpovědnost programátora).

Direktiva `single`

- V kontextu paralelně prováděného bloku je následující strukturní blok proveden pouze jedním vláknem, přičemž není určeno kterým.

Klauzule: `private`, `firstprivate`

Klauzule: `nowait`

- Pokud není uvedena, tak na konci strukturního bloku označeného direktivou `single` je provedena bariéra.

Direktiva `master`

- Speciální případ direktivy `single`.
- Tím vláknem, které provede strážení blok, bude hlavní (`master`) vlákno.

Direktiva `critical`

- Následující strukturovaný blok je chápán jako kritická sekce a může být prováděn maximálně jedním vláknem v daném čase.
- Kritická sekce může být pojmenována, souběžně je možné provádět kód v kritických sekcích s jiným názvem.
- Pokud není uvedeno jinak, použije se implicitní jméno.
- `#pragma omp critical [(name)]`

Direktiva `atomic`

- Nahrazuje kritickou sekci nad jednoduchými modifikacemi (update) proměnných v paměti.
- Atomicita se aplikuje na jeden následující výraz.
- Obecně výraz musí být jednoduchý (jeden *load* a *store*).
- Neatomizovatelný výraz: `x = y = 0;`

Problém (nestálé proměnné)

- Modifikace sdílených proměnných v jednom vlákně může zůstat skryta ostatním vláknům.

Řešení

- Explicitní direktiva pro kopírování hodnoty proměnné z registru do paměti a zpět.
- `#pragma omp flush [(seznam)]`

Použití

- Po zápisu do sdílené proměnné.
- Před čtením obsahu sdílené proměnné.
- Implicitní v místech bariéry a konce bloků (pokud nejsou bloky v režimu `nowait`).

Problém (thread-private data)

- Při statickém mapování na vlákna je drahé při opakovaném vzniku a zániku vláken vytvářet kopie privátních proměnných.
- Občas chceme privátní globální proměnné.

Řešení

- Perzistentní privátní proměnné (přetrvají zánik vlákna).
- Při znovuvytvoření vlákna, se proměnné znovupoužijí.
- `#pragma omp threadprivate` (seznam)

Omezení

- Nesmí se použít dynamické plánování vláken.
- Počet vláken v paralelních blocích musí být shodný.

Direktiva copyin

- Jako `threadprivate`, ale s inicializací.
- Viz `private` versus `firstprivate`.

```
void omp_set_num_threads (int num_threads)
```

- Specifikuje kolik vláken se vytvoří při příštím použití direktivy `parallel`.
- Musí být použito před samotnou konstrukcí `parallel`.
- Je přebito klauzulí `num_threads`, pokud je přítomna.
- Musí být povoleno dynamické modifikování procesů (`OMP_DYNAMIC`, `omp_set_dynamic()`).

```
int omp_get_num_threads ()
```

- Vrací počet vláken v týmu strukturálně nejbližší direktivy `parallel`, pokud neexistuje, vrací 1.

`int omp_get_max_threads ()`

- Vrací maximální počet vláken v týmu.

`int omp_get_thread_num ()`

- Vrací unikátní identifikátor vlákna v rámci týmu.

`int omp_get_num_procs ()`

- Vrací počet dostupných procesorů, které mohou v daném okamžiku participovat na vykonávání paralelního kódu.

`int omp_in_parallel ()`

- Vrací nenula pokud je voláno v rozsahu paralelního bloku.

```
void omp_set_dynamic (int dynamic_threads)  
int omp_get_dynamic()
```

- Nastavuje a vrací, zda je programátorovi umožněno dynamicky měnit počet vláken vytvořených při dosažení direktivy `parallel`.
- Nenulová hodnota `dynamic_threads` značí povoleno.

```
void omp_set_nested (int nested)  
int omp_get_nested()
```

- Nastavuje a vrací, zda je povolen vnořený paralelismus.
- Pokud není povoleno, vnořené paralelní bloky jsou serializovány.


```
void omp_init_lock (omp_lock_t *lock)
void omp_destroy_lock (omp_lock_t *lock)
void omp_set_lock (omp_lock_t *lock)
void omp_unset_lock (omp_lock_t *lock)
int  omp_test_lock (omp_lock_t *lock)
```

```
void omp_init_nest_lock (omp_nest_lock_t *lock)
void omp_destroy_nest_lock (omp_nest_lock_t *lock)
void omp_set_nest_lock (omp_nest_lock_t *lock)
void omp_unset_nest_lock (omp_nest_lock_t *lock)
int  omp_test_nest_lock (omp_nest_lock_t *lock)
```

- Inicializuje, ničí, blokuje a čeká, odemyká a testuje –
- – normální a rekurzivní mutex.

OMP_NUM_THREADS

- Specifikuje defaultní počet vláken, který se vytvoří při použití direktivy `parallel`.

OMP_DYNAMIC

- Hodnota `TRUE`, umožňuje za běhu měnit dynamicky počet vláken.

OMP_NESTED

- Povoluje hodnotou `TRUE` vnořený paralelismus.
- Hodnotou `FALSE` specifikuje, že vnořené paralelní konstrukce budou serializovány.

OMP_SCHEDULE

- Udává defaultní nastavení mapování iterací cyklu na vlákna.
- Příklady hodnot: `"static,4"`, `dynamic`, `guided`.

Intel's Thread Building Blocks (TBB)

Co je Intel TBB

- TBB je C++ knihovna pro vytváření vícevláknových aplikací.
- Založená na principu zvaném **Generic Programming**.
- Vyvinuto synergickým spojením Pragma direktiv (OpenMP), standardní knihovny šablon (STL, STAPL) a programovacích jazyků podporující práci s vlákny (Threaded-C, Cilk).

Generic Programming

- Vytváření aplikací specializací existujících předpřipravených obecných konstrukcí, objektů a algoritmů.
- Lze nalézt v objektově orientovaných jazycích (C++, JAVA).
- V C++ jsou obecnou konstrukcí šablony (templates).
 - `Queue<Int>`
 - `Queue<Queue<Char>>`

Vlastnosti Intel TBB

- Knihovna, implementovaná s využitím standardního C++.
- Nepožaduje podporu speciálního jazyka či překladače.
- Podporuje vnořený paralelismus, potažmo je možné stavět složitější paralelní systémy z menších paralelních komponent.
- Cílem použití je nechat programátora specifikovat úlohy k paralelnímu provedení, nikoliv ho nutit popisovat, co a jak dělají jednotlivá vlákna.

Home Page

- <http://www.threadingbuildingblocks.org/>

TBB poskytuje šablony pro

- Paralelizaci iterací jednotlivých cyklů – datový paralelismus.
- Definici vlastních paralelně přístupovaných datových struktur.
- Využití nízkoúrovňových HW primitiv.
- Zamykání přístupů do kritické sekce v různých podobách.
- Snadnou definici paralelních souběžných úloh.
- Škálovatelnou alokaci paměti.

IB109

- Pouze demonstrace použití TBB.
- Kompletní použití TBB je nad rámec tohoto kurzu.

Paralelní for-cyklus

- Je dána množina nezávislých indexů, tzv. rozsah (range).
- Pro každý index z množiny je provedeno tělo cyklu.

Paralelní for-cyklus v TBB

- Šablona, která má dva parametry – Rozsah a tělo cyklu.
- Šablona zajistí vykonání těla cyklu pro všechny indexy ve specifikovaném rozsahu.
- Rozsah je dělen na pod-rozsahy. Paralelismu dosaženo souběžným vykonáváním těla cyklu nad jednotlivými pod-rozsahy.

Příklad – paralelní for

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

const int n=1000;
float input[n];
float output[n];

struct Average {
    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);
    }
};

Average avg;

parallel_for( blocked_range<int>( 1, n ), avg );
```


Koncept dělení

- Instance některých tříd je nutné za běhu (rekurzivně) dělit.
- Zavádí se nový typ konstruktoru, dělicí konstruktor:
$$X::X(X\& x, split)$$
- Dělicí konstruktor rozdělí instanci třídy X na dvě části, které dohromady dávají původní objekt. Jedna část je přiřazena do x , druhá část je přiřazena do nově vzniklé instance.
- Schopnost dělit-se musí mít zejména rozsahy, ale také třídy, jejichž instance běží paralelně a přitom nějakým způsobem interagují, např. třídy realizující paralelní redukci.

split

- Speciální třída definovaná za účelem odlišení dělicího konstrukturu od kopírovacího konstrukturu.

Požadavky na třídu realizující rozsah

- Kopírovací konstruktor

```
R::R (const R&)
```

- Dělicí konstruktor

```
R::R (const R&, split)
```

- Destruktor

```
R::~~R ()
```

- Test na prázdnotu rozsahu

```
bool R::empty() const
```

- Test na schopnost dalšího rozdělení

```
bool R::is_divisible() const
```

Předdefinované šablony rozsahů

- Jednodimenzionální: `blocked_range`
- Dvoudimenzionální: `blocked_range2d`

blocked_range

- `template<typename Value> class blocked_range;`
- Reprezentuje nadále dělitelný otevřený interval $[i, j)$.

Požadavky na třídu Value specializující blocked_range

- Kopírovací konstruktor
`Value::Value (const Value&)`
- Destruktor
`Value::~~Value ()`
- Operátor porovnání
`bool Value::operator<(const Value& i, const Value& j)`
- Počet objektů v daném rozsahu (operátor `-`)
`size_t Value::operator-(const Value& i, const Value& j)`
- k -tý následný objekt po i (operátor `+`)
`Value Value::operator+(const Value& i)`

Použití blocked_range<Value>

- Nejdůležitější metodou je konstruktor.
- Konstruktor specifikuje interval rozsahu a velikost největšího dále nedělitelného sub-intervalu:
- `blocked_range(Value begin, Value end [, size_t grainsize])`

Typická specializace

- `blocked_range<int>`
- Příklad: `blocked_range<int>(5, 17, 2)`
- Příklad: `blocked_range<int>(0, 11)`

```
parallel_for<Range,Body>
```

- `template<typename Range, typename Body>`
`void parallel_for(const Range& range, const Body& body);`

Požadavky na třídu realizující tělo cyklu

- Kopírovací konstruktor
`Body::Body (const Body&)`
- Destruktor
`Body::~~Body ()`
- Aplikátor těla cyklu na daný rozsah – operátor ()
`void Body::operator()(Range& range) const`

```
parallel_reduce<Range, Body>
```

- `template<typename Range, typename Body>`
`void parallel_reduce(const Range& range, const Body& body);`

Požadavky na třídu realizující tělo redukce

- Dělicí konstruktor
`Body::Body (const Body&, split)`
- Destruktor
`Body::~Body ()`
- Funkce realizující redukci nad daným rozsahem – operátor ()
`void Body::operator()(Range& range)`
- Funkce realizující redukci hodnot z různých rozsahů
`void Body::join(Body& to_be_joined)`

Třída Partitioner

- Paralelní konstrukce mají třetí volitelný parametr, který specifikuje strategii dělení rozsahu.
- `parallel_for<Range,Body,Partitioner>`

Předdefinované strategie

- `simple_partitioner`
 - Rekurzivně dělí rozsah až na dále nedělitelné intervaly.
 - Při použití `blocked_range` je volba `grainsize` klíčová pro vyvážení potenciálu a režie paralelizace.
- `auto_partitioner`
 - Automatické dělení, které zohledňuje zatížení vláken.
 - Při použití `blocked_range` volí rozsahy větší, než je `grainsize` a tyto dělí pouze do té doby, než je dosaženo rozumného vyvážení zátěže. Volba minimální velikosti `grainsize` nezpůsobí nadbytečnou režii spojenou s paralelizací.

`concurrent_queue`

- `template<typename T> concurrent_queue`
- Fronta, ke které může souběžně přistupovat více vláken.
- Velikost fronty je dána počtem operací vložení bez počtu operací výběru. Záporná hodnota značí čekající operace výběru.
- Definuje sekvenční iterátory, nedoporučuje se je používat.

`concurrent_vector`

- `template<typename T> concurrent_vector`
- Zvětšovatelné pole prvků, ke kterému je možné souběžně přistupovat z více vláken a provádět souběžně zvětšování pole a přístup k již uloženým prvkům.
- Nad vektorem lze definovat rozsah a provádět skrze něj paralelně operace s prvky uloženými v poli.

concurrent_hash_map

- `template<typename Key, typename T, typename HashCompare>`
`class concurrent_hash_map;`
- Mapa, ve které je možné paralelně hledat, mazat a vkládat.

Požadavky na třídu HashCompare

- Kopírovací konstruktor
`HashCompare::HashCompare (const HashCompare&)`
- Destruktor
`HashCompare::~HashCompare ()`
- Test na ekvivalenci objektů
`bool HashCompare::equal(const Key& i, const Key& j) const`
- Výpočet hodnoty hešovací funkce
`size_t HashCompare::hash(const Key& k)`

Objekty pro přístup k datům v `concurrent_hash_map`

- Přístup k párům Klíč-Hodnota je skrze přístupovací třídy.
- `accessor` – pro přístup v režimu read/write
- `const_accessor` – pro přístup pouze v režimu read
- Použití přístupovacích objektů umožňuje korektní paralelní přístup ke sdíleným datům.

Příklad použití přístupovacího objektu

- ```
typedef concurrent_hash_map<Int,Int> MyTable;
MyTable table;

MyTable::accessor a;
table.insert(a, 4);
a->second += 1;
a.release();
```

## Metody pro práci s `concurrent_hash_map`

- `bool find(const_accessor& result, const Key& key) const`
- `bool find(accessor& result, const Key& key)`
- `bool insert(const_accessor& result, const Key& key)`
- `bool erase(const Key& key)`

## Další způsoby použití

- Iterátory pro procházení mapy.
- Lze definovat rozsahy a s nimi pracovat paralelně.

# C++11

## Pozorování

- C++11 má definované příkazy pro podporu vláken.
- Není třeba používat externí knihovny jako je POSIX Thread.

## Jak je to možné

- C++11 definuje virtuální výpočetní stroj.
- Veškerá sémantika příkazů se odkazuje na tento virtuální výpočetní stroj.
- Virtuální výpočetní stroj je paralelní, příkazy související s podporou vláken mohou být součástí jazyka.
- Přenos sémantiky z virtuálního výpočetního stroje na reálný HW je na zodpovědnosti překladače.

# Příklad – Vlákna a mutexy v C++11

```
#include <thread>
#include <mutex>
std::mutex mylock;

void func(int& a)
{
 mylock.lock();
 a++;
 mylock.unlock();
}

int main()
{
 int a = 42;
 std::thread t1(func, std::ref(a));
 std::thread t2(func, std::ref(a));
 t1.join();
 t2.join();
 std::cout << a << std::endl;
 return 0;
}
```

## Potencionální riziko uváznutí

- Jazyk s plnou podporou mechanismu výjimek.
- Vyvolání výjimky v okamžiku, kdy je vlákno v kritické sekci (uvnitř mutexu) pravděpodobně způsobí, že nebude vláknem volána metoda odemykající zámek svázaný s kritickou sekcí.

## Řešení

- Využití principu RAII a OOP.
- Zamčení mutexu realizováno vytvořením lokální instance vhodné předdefinované zamykací třídy.
- Odemykání umístěno do destruktoru této třídy.
- Destruktor je proveden v okamžiku opuštění rozsahu platnosti daného objektu.

## Třída `lock_guard`

- Obalení standardního zámku v RAII stylu.
- Mutex na pozadí nelze „předat“ jinému vláknu, nevhodné pro podmínkové proměnné.
- Příklad použití:

```
std::mutex m;
void func(int& a)
{
 std::lock_guard<std::mutex> l(m);
 a++;
}
```

## Třída `unique_lock`

- Obecnější předatelné RAII obalení mutexu.
- Doporučené pro použití s podmínkovými proměnnými.



## Podpora vláknování v C++11

- Vlákna.
- Mutexy a RAII zámky.
- Podmínkové proměnné.
- Sdílené futures (místa uložení dosud nespočítané hodnoty).

## Rozcestník

- <http://en.cppreference.com/w/cpp/thread>

## Jiné rychlé přehledy

- <http://www.codeproject.com/Articles/598695/Cplusplus-threads-locks-and-condition-variables>
- <http://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g>

## Neatomicky

- `int x,y;`

Thread 1

`x = 17;`

`y = 37;`

Thread 2

`cout << y << " ";`

`cout << x << endl;`

- Nemá definované chování.

## Správně atomicky

- `atomic<int> x, y;`

Thread 1

`x.store(17);`

`y.store(37);`

Thread 2

`cout << y.load() << " ";`

`cout << x.load() << endl;`

- Chování je definované, možné výstupy: 0 0, 0 17, 37 17.

## Paměťový model

- Implicitní chování zachovává sekvenční konzistenci (automaticky vkládá odpovídající paměťové bariéry).
- Riziko neefektivního kódu.

## Příklad 1

- `atomic<int> x, y;`

Thread 1

```
x.store(17,memory_order_relaxed);
y.store(37,memory_order_relaxed);
```

Thread 2

```
cout << y.load(memory_order_relaxed) << " ";
cout << x.load(memory_order_relaxed) << endl;
```

- Sémantika povoluje v tomto případě i výstup: 37 0.

## Paměťový model

- Implicitní chování zachovává sekvenční konzistenci (automaticky vkládá odpovídající paměťové bariéry).
- Riziko neefektivního kódu.

## Příklad 2

- `atomic<int> x, y;`

Thread 1

```
x.store(17,memory_order_release);
y.store(37,memory_order_release);
```

Thread 2

```
cout << y.load(memory_order_acquire) << " ";
cout << x.load(memory_order_acquire) << endl;
```

- Acquire nepřeuspořádá operace load, Release – store.

## Jiné přístupy

## Paralelní for cyklus

- Nejčastější a nejjednodušší metoda paralelizace.
- Datová paralelizace.

## Jak a kde lze řešit paralelní for cyklus

- <http://parallel-for.sourceforge.net/>