

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

- Paralelizace deklarativním stylem programování.
- Programátor specifikuje co chce, ne jak se to má udělat.

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>

Použití

- Strukturovaný blok, tj. `{...}`, následující za touto direktivou se provede paralelně
- Mimo tyto 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 vlákne

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 `||`

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

- 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`

- viz direktiva `parallel`

Klauzule: `lastprivate`

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

Klauzule: `ordered`

- Bloky označené jako `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
- Pro označení `ordered` bloku se použije direktiva `ordered`

Klauzule: `nowait`

- Jednotlivá vlákna se nesynchronizují po provedení cyklu

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

- Určuje jak budou jednotlivé 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`

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

Direktiva `single`

- Následující strukturní blok proveden pouze jedním vláknem (není určeno kterým)
- Vztahuje se k direktivě `parallel` (viz `sections` ve `for`)

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 strážžený 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_dynamic()
```

- 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 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>>`

Home Page

- <http://threadingbuildingblocks.org/>

Tutoriál

- <http://threadingbuildingblocks.org/uploads/81/91/Latest%20pen%20Source%20Documentation/Tutorial.pdf>

Getting started

- http://threadingbuildingblocks.org/uploads/81/91/Latest%20pen%20Source%20Documentation/Getting_Started.pdf

Referenční manuál

- <http://threadingbuildingblocks.org/uploads/81/91/Latest%20pen%20Source%20Documentation/Reference.pdf>

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 – Tělo cyklu a Rozsah.
- 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.

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 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)`

```
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.
- Je možné definovat vlastní dělicí strategie.

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.

`accessor`

- `template<typename Key, typename T, typename HashCompare>`
`class concurrent_hash_map<Key,T,HashCompare>::accessor;`
- `accessor: typedef const std::pair<const Key,T> value_type`
- `accessor: value_type& operator*() const`
- `accessor: value_type& operator->() const`

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.

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/>