

Programování se sdílenou pamětí

OpenMP synchronizace

AVS – Architektury výpočetních systémů
Týden 9, 2024/2025

Jirka Jaroš

Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2, 612 66 Brno - Královo Pole
jarosjir@fit.vutbr.cz



- **Synchronizace slouží**

- k ochraně přístupů ke sdíleným datům,
- k čekání na nějakou událost nebo
- k vynucení pořadí akcí.

- **OpenMP direktivy:**

- **Vysoká úroveň**

critical,
atomic,
barrier,
master,
ordered

}

vzájemné vyloučení

}

synchronizace událostmi

barrier, flush
jsou implicitně
součástí
jiných direktiv!

- **Nízká úroveň** (synchronizace na míru):

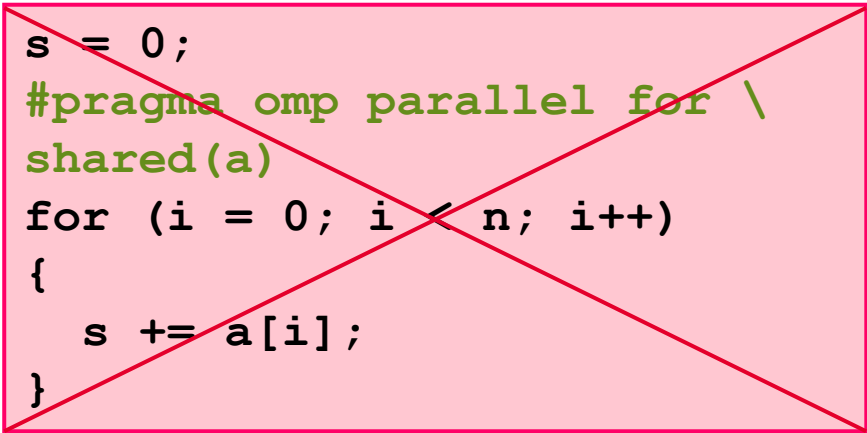
flush

locks (jednoduché a vnořené)

1. Několik vláken čte a modifikuje **disjunktní prvky** sdíleného objektu
= **bez problému**
2. Několik vláken čte a zapisuje **do téže sdílené proměnné** (datové struktury)
= **problém**

Případ 2 vyžaduje synchronizaci.

```
#pragma omp parallel for \
shared(a, b)
for (i = 0; i < n; i++)
{
    a[i] += b;
}
```



```
s = 0;
#pragma omp parallel for \
shared(a)
for (i = 0; i < n; i++)
{
    s += a[i];
}
```

THREAD 1: increment(x)

```
{  
    x++;  
}
```

THREAD 1:

```
1 lw    r2, 0(r1)  
2 addi  r2, #1  
3 sw    r2, 0(r1)
```

Není
atomická
operace

THREAD 2: increment(x)

```
{  
    x++;  
}
```

THREAD 2:

```
1 lw    r2, 0(r1)  
2 addi  r2, #1  
3 sw    r2, 0(r1)
```

- Proložení 1-1-2-2-3-3 dává výsledek $x = 1$
- Proložení 1-2-3-1-2-3 dává zamýšlený výsledek $x = 2$
- Adresa x je v registru $r1$, přičte se k ní offset 0

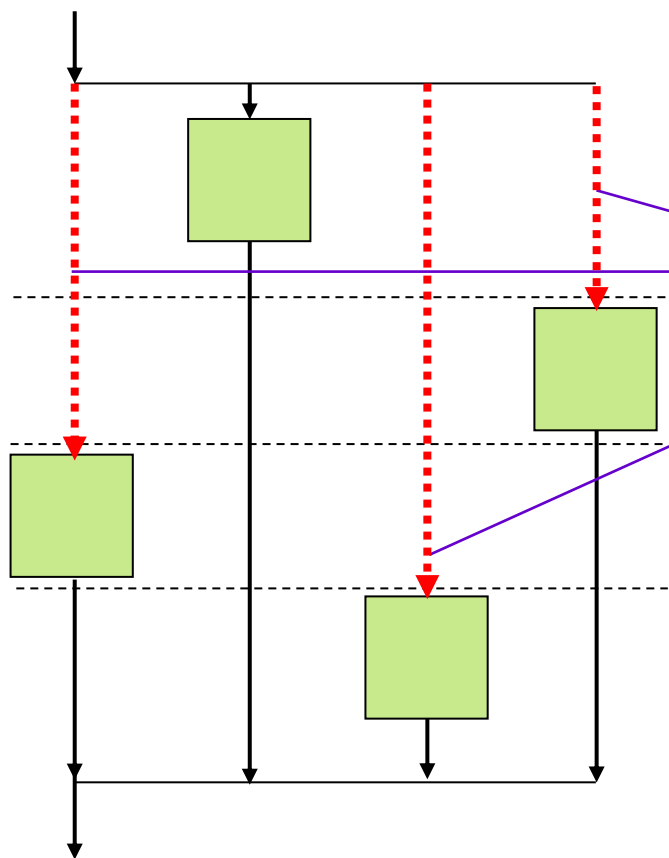
```
/*nalezení počtu nulových prvků vektoru*/  
int count =0;  
# pragma omp parallel for  
  for (i = 0; i < n; i++)  
    if (a[i] == 0) count++;
```

- **špatně**, modifikace sdílené proměnné **count** souběžně více vlákn.
- Vlákn musejí **přístupovat ke count exkluzivně**, přístup jednoho vlákna vylučuje přístup jiného!
(Zde se snadno vyřeší dovětkem **reduction(+:count)**)

OpenMP direktivy kompilátoru

VZÁJEMNÉ VYLOUČENÍ

```
#pragma omp critical[(name)]  
blok
```



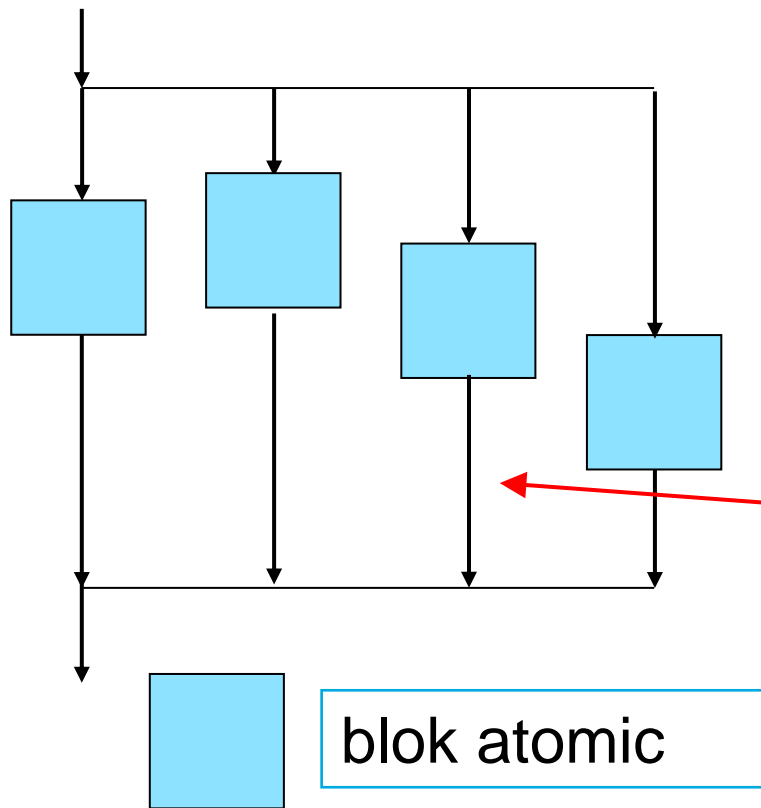
Čekání na vstup do
kritické sekce

Každé vlákno provádí kritickou sekci,
ale v určitém okamžiku pouze jediné.
Pořadí není zaručeno!



Kritická sekce

`#pragma omp atomic`



- Každé vlákno provádí blok **atomic**, např. inkrementaci prvku vektoru, **nerozdělitelně**.
- Pokud každé vlákno modifikuje
 - **jiný prvek vektoru**, mohou běžet souběžně
 - **stejnou proměnnou**, běží jedno vlákno po druhém (serializace jako u critical)

1. Vzájemné vyloučení pomocí **kritické sekce (CS)** znamená, že CS se provádí postupně jednotlivými vlákny (serializace, opak paralelizace).

Implementace: pomocí sdílené binární proměnné **zámek (lock)**.

2. Atomicita znamená nepřerušené zpracování sekvence instrukcí.

Implementace: pomocí atomických instrukcí (čtení – modifikace – záznam). Několik vláken může pracovat v atomické sekci **souběžně**, pokud

- pracují **s různými daty** a pokud
- atomické instrukce z více vláken **nejsou serializovány** architekturou (sběrnice).

```
s = 0;
#pragma omp parallel for \
    shared(a)
for (i = 0; i < n; i++)
{
    #pragma omp critical
    s += a[i];
}
```

Výsledek je správně, ale výpočet je sekveční.

Výpočet je tedy velice pomalý (serializace + režie CS)

Atomická sekce zde příliš nepomůže, jen sníží režii.

OpenMP přece podporuje redukci...

```
s = 0;
#pragma omp parallel for reduction(+:s)
for (i = 0; i < n; i++)
    s += a[i];
```

```
int first = n;
#pragma omp parallel for
for (int i = 0; i < n; i++)
{
    if (a[i] == 0 && i < first)
    {
        #pragma omp critical
        if (i < first) first = i;
    }
}
```

- Hledá se index **first** prvního nulového prvku vektoru.
- **first = n** když neexistuje.

Proč dva testy **i < first** ?

1. Abychom vstupovali do kritické sekce jen když je to nutné, co nejméně krát!
2. Aby bylo jisté, že v mezičase nenalezlo jiné vlákno nulový prvek s nižším indexem

Lépe: `#pragma omp parallel for reduction(min:first)`

- Sdílená datová struktura obsahuje seznam úloh různé velikosti a složitosti.
- Daná položka (item) může být zpracována libovolným vláknem, více položek souběžně více vlákeny.

```
int  get_next_item();
void process_item(int item);

int main()
{
    int item;
    #pragma omp parallel private(item)
    {
        item = get_next_item();
        while (item != -1)
        {
            process_item(item);
            item = get_next_item();
        }
    }
}
```

Fronta obsahuje úlohy označené indexy
0 až MAX-1

```
int get_next_item()
{
    static int head = 0;
    int new_item;
    #pragma omp critical
    {
        if (head == MAX)
            new_item = -1;
        else
            new_item = head++;
    }
    return new_item;
}
```

- **Nepojmenované CS jsou globální.**
 - V jednom okamžiku může běžet kdekoliv v programu jen jedna CS.
- **OpenMP dovoluje kritické sekce pojmenovat a tím snížit čekání vláken:**
 - ze stejně pojmenovaných sekcí může v daném okamžiku běžet jen jedna,
 - ale sekce pojmenované jinak mohou běžet souběžně!
- Vnoření **critical** stejného jména = deadlock.
- Vnoření **critical** různého jména: musí být zajištěno stejné pořadí vstupu do kritických sekcí, jinak deadlock.
- **Proto je třeba CS sekci vždy pojmenovat!**

- Direktiva chrání aktualizaci (tj. čtení – modifikaci – zápis, angl. **update**) **jednoho** paměťového místa.

```
#pragma omp atomic
```

```
x++; | ++x; | x--; | --x;
```

```
x binop = expr; | x = x binop expr;
```

- kde *x* je skalár a *binop* je:

+, *, −, /, bitwise AND, XOR, OR (&, ^, |), <<, >>

- Dovětky direktivy atomic byly přidány v OpenMP 3.1:

```
[read | write | update | capture] [seq_cst]
```

default

```
# pragma omp atomic read
```

```
v = x;
```

```
# pragma omp atomic write
```

```
x = expr;
```

Žádná část *x* se nemůže změnit dokud R/W není hotové.

- Dovětek capture umožňuje atomickou aktualizaci se záchytem (přiřazením) hodnoty před nebo po aktualizaci do privátní proměnné (fetch-and-increment).

#pragma omp atomic [capture]
příkaz nebo strukturovaný blok

```
#pragma omp atomic capture  
{  
    old_value = *p;  
    (*p)++;  
}  
// zde se dá použít old_value
```

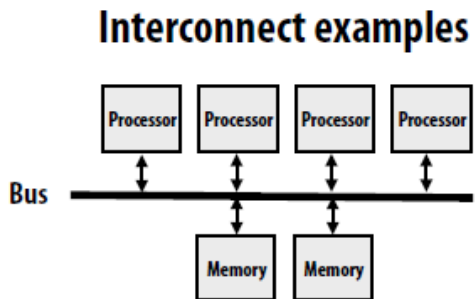
- Dovětek **seq_cst** = sequential consistency, garantuje proložené provedení operací všech vláken a současně zachování pořadí operací v kódu jednotlivých vláken. Obsahuje implicitní operaci flush (viz dále).

- Vytvoření histogramu z vektoru n čísel $a[i]$

```
#pragma omp parallel for shared(histogram, a, n)
for (i=0; i < n; i++)
{
    #pragma omp atomic
    histogram[a[i]] += 1;
}
```

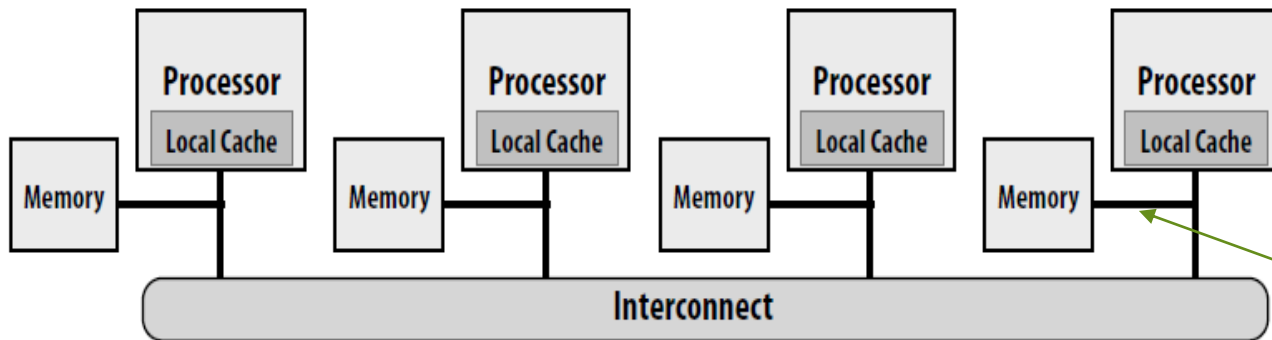
- Na rozdíl od **critical** může zpracování atomického bloku využít přídavný paralelismus. Avšak
 - současná aktualizace **různých paměťových míst** může probíhat **jen když atomické instrukce nebudou serializovány** (např. sběrnicí);
 - výkonost může klesnout kvůli falešnému **sdílení (false sharing)**, kdy je generováno mnoho výpadků (viz dále).

Kolik atomických instrukcí může běžet současně:



Atomická instrukce blokuje sběrnici – serializace

Až 4 atomické instrukce současně, pokud přistupují do různých modulů paměti



Počet kanálů?

```
#pragma omp parallel for \  
    reduction(+: s)  
for (int i = 0; i < n; i++)  
{  
    s += a[i];  
}
```

```
s = 0;  
#pragma omp parallel for  
for (int i = 0; i < n; i++)  
{  
    #pragma omp atomic  
    s += a[i];  
} /*pomalé, serializované*/
```

```
s = 0;  
#pragma omp parallel  
{  
    int mysum = 0;  
    #pragma omp for nowait  
    for (int i = 0; i < n; i++)  
    {  
        mysum += a[i];  
    }  
    #pragma omp atomic  
    s += mysum;  
} /*velmi dobré provedení*/
```

- Kritická sekce CS je časově drahá jen když o ni silně soupeří mnoho jader.

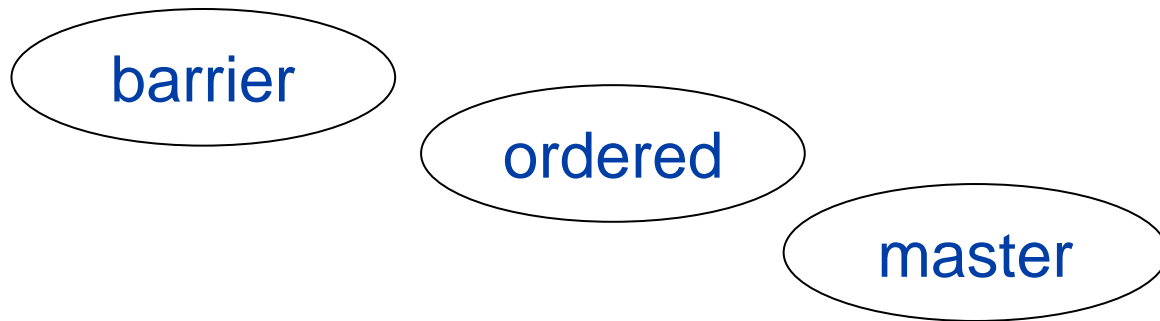
Snížení režie:

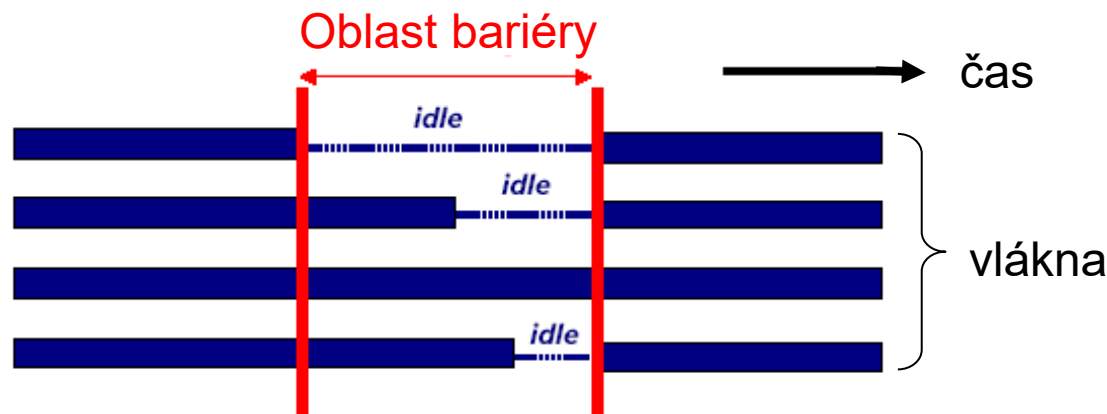
- Kde je to možné, použij direktivu **atomic místo critical**.
 - Důvod: zápisy do jedné sdílené proměnné jsou sice serializovány oběma direktivami, ale režie atomic je mnohem menší – je implementována jen atomickými instrukcemi bez spin zámků.
- Místo jedné CS je často lepší použít **mnoho zámků** a tak značně snížit soupeření o jeden každý zámek (tzv. vektorový zámek).
Např. 1 zámek na každou 1/10 intervalu hodnot histogramu.

OpenMP direktivy kompilátoru

SYNCHRONIZACE UDÁLOSTMI

- Jedno vlákno (proces) nastaví boolovskou podmínku, další vlákna (procesy) ji testují a čekají na jej nastavení.
- Sem patří bariéra, synchronizace 2 vláken pomocí boolovského návěští (flag), aj.





- Bariéra způsobí, že vlákna co přišla k bariéře čekají až k bariéře dorazí všechna vlákna týmu.
- Vícenásobné bariéry musí procházet všechna vlákna ve stejném pořadí.
- `#pragma omp barrier` má smysl pouze v rozsahu direktivy `parallel`, může být i sirotek.

- Bariéry nejsou dovoleny v **dynamickém** rozsahu **for**, **ordered**, **sections**, **single**, **master** a v sekcích **critical**.
- Implicitní bariéry jsou na konci paralelních oblastí **parallel**, paralelních smyček **for**, sekcí **sections**, a skupin tasků **taskgroup** (pokud není potlačeno dovětkem **nowait**)
- Bariéry mají značnou časovou režii. Je třeba je používat obezřetně!

- Oblast `ordered` se provádí v sekvenčním pořadí (serializace, velká režie)
- Užitečné pro sekvenční uspořádání výstupu výpočtů (debug):

```
#pragma omp for ordered schedule(dynamic)
for (i = low; i < high; i += step)
    work(i);
```

dovětek `ordered` avizuje přítomnost
direktivy `ordered` někde v paralelní oblasti

```
void work(int k)
{
    #pragma omp ordered
    printf(" %d", k);
}
```

direktiva `ordered` (sirotek)

#pragma omp master strukturovaný blok

- Označuje blok kódu **v rámci paralelní oblasti**, který je proveden jen **hlavním vláknem**.
- Ostatní vlákna blok přeskočí a nemusí ani dojít k této direktivě.
- Na rozdíl od single **neobsahuje** implicitní bariéru na konci bloku
- **Použití:**
 - pro omezení I/O operací jen na hlavní vlákno
 - pro přístup do jeho threadprivate proměnných.
 - generování tasků,...

OpenMP direktivy a funkce

SYNCHRONIZACE PROGRAMOVANÁ UŽIVATELEM

vlákno p1

(producent)

data = ... ;

flag = 1;

vlákno p2

(konzument)

while (flag==0) ;

... = data;

Takto jednoduše
komunikovat NELZE !

- Předpokládá provedení zápisů (producent) a čtení (konzument) v pořadí uvedeném v kódu.
- Moderní procesory/kompilátory **mění pořadí jak čtení, tak zápisů** (je to nezbytné pro zlepšení výkonnosti). Případně mohou být paměťové transakce zviditelněny v opačném pořadí vlivem propojovací sítě.
- Pro obnovení správného pořadí použijte direktivu flush!

- Paměťová **konzistence** se týká pořadí, ve kterém vidí **různá vlákna** aktualizace **různých** paměťových míst, tj. **kdy** jsou zápisy vidět.
 - **Zápisy** na adresu **x** a pak na **y** mohou dvě vlákna vidět **v jiném pořadí** (jedno vlákno xy, druhé yx nebo obě yx).
- Modely konzistence jsou založeny na uspořádání R, W a S (S = synchronizační instrukce **fence**).
- Základní je **sekvenční konzistence**, SC. Předpokládá, že
 - paměťové operace jsou atomické,
 - pořadí přístupů do paměti specifikované v programu každého vlákna nemění kompilátor ani HW CPU.
- Operace R, W, S jsou v multiprocesoru sekvenčně konzistentní když **globální sekvence** přístupů do paměti
 1. **zachovává pořadí** přístupů každého vlákna
 2. je **proložením kódů** všech vláken.

- **Sekvenční konzistence** je v současnosti nemoderní, je totiž překážkou modernímu hardware a optimalizujícím kompilátorům.
- Moderní procesory vykazují **relaxovanou** (uvolněnou) **paměťovou konzistenci**, kdy se čtení a zápisy mohou vzájemně předbíhat, i když je to proti intuici. Je to kvůli zlepšení výkonnosti. **OpenMP ji podporuje.**
- Modely konzistence paměti jsou často exportovány z procesorů do multiprocesorů, kdy různé CPU mohou vidět paměťové operace v jiném pořadí i vlivem propojovací sítě.
- Programátor musí použít synchronizační příkazy a primitiva (např. fence, **flush**) aby **vymezil oblasti relaxovaného chování.**

```
#pragma omp flush [ (list) ]
```

- Definuje bod, v němž má vlákno garantováno, že hodnoty
 - všech proměnných viditelných vláknu (když chybí list)
 - nebo jen proměnných v seznamu listjsou konzistentní s hlavní pamětí a tak viditelné všem vláknům.
- Kompilátor může totiž některé proměnné dočasně udržovat v registrech místo v paměti kvůli optimalizaci, HW zase v Load nebo Store buferu.
- Paměťové operace mohou být jen částečně dokončené, jejich výsledky na cestě. Takže některá vlákna mohou vidět dočasně sdílenou paměť jinak než jiná vlákna.
- Flush vynutí v místě svého výskytu shodné (konzistentní) vidění všech vláken.

Doporučení: na seznamu (list) používat jen jednu proměnnou!

- Flush se chová se jako **paměťová zábrana (*memory fence*)** – brání přesunům paměťových přístupů R a W (se všemi proměnnými nebo těmi uvedenými v seznamu list) přes **flush**:
 - všechny operace R, W před flush se dokončí;
 - nové op. R, W za flush se zahájí až po návratu z flush;
 - dvě operace flush s překrývajícími se množinami proměnných v seznamech list nemohou být přehozeny.
- **flush je implicitně přítomen mj.**
 - na vstupu nebo výstupu z **paralelních oblastí**
 - na implicitních a explicitních **bariérách**
 - na vstupu/výstupu oblastí **critical**
 - kdykoli je **zámek „set“** nebo „unset“

Jaká přeskládání jsou možná v následujícím kódu:

```
a = ...; // (1)
b = ...; // (2)
c = ...; // (3)
#pragma omp flush(c) // (4)
#pragma omp flush // (5)
. . . a . . . b . . . ; // (6)
. . . c . . . ; // (7)
```

- (1) a (2) se nesmí přesunout za (5),
- (6) se nesmí přesunout před (5),
- (4) a (5) se nemohou vyměnit, (c) a () se překrývají
- (3) se nesmí přesunout za (4),
- (7) se nesmí přesunout před (4), atd.

- OpenMP nemá synchronizační konstrukty pro tento účel
- Když je to potřeba, musí to zařídit programátor sám:

```
int main()
{
    double *A, sum;
    int flag = 0, flg_tmp;
    A = new double[N];

    #pragma omp parallel sections
    {
        #pragma omp section ← producent
        {
            fill_rand(N, A);
            #pragma omp flush
            #pragma omp atomic write
                flag = 1;

            #pragma omp flush (flag)
        } // end of producent
    }
```

```
#pragma omp section ← konzument
{
    while (true)
    {
        #pragma omp flush(flag)
        #pragma omp atomic read
            flg_tmp = flag;
        if (flg_tmp == 1)
            break;
    }
    #pragma omp flush
    sum = sum_array(N, A);
} // end of consument
} // end of parallel
delete[] A;
} // main
```


- flush není třeba, jelikož `seq_cst` jej přidává k atomickým operacím.

```
int main()
{
    double *A, sum;
    int flag = 0, flg_tmp;
    A = new double[N];

    #pragma omp parallel sections
    {
        #pragma omp section ← producent
        {
            fill_rand(N, A);
            #pragma omp atomic write seq_cst
            flag = 1;
        } // end of producent
    }
```

```
    #pragma omp section ← konzument
    {
        while (true)
        {
            #pragma omp atomic read seq_cst
            flg_tmp = flag;
            if (flg_tmp == 1)
                break;
        }
        sum = sum_array(N, A);
    } // end of consument
} // end of parallel
delete[] A;
} // main
```

- Proměnná zámku umožňuje synchronizovat vlákna
 - je typu `omp_lock_t` a má 64 bitovou adresu.

Knihovní rutiny:

- `omp_init_lock` – inicializuje (alokuje) zámek.
- `omp_set_lock` – čeká na volný zámek (blokuje) až se dočká, nastaví jej.
- `omp_test_lock` – pokusí se získat zámek, ale nečeká (neblokuje); vrátí úspěch (1), nezdar (0)
- `omp_unset_lock` – uvolní zámek
- `omp_destroy_lock` – dealokace

```
#include <omp.h>
int main()
{
    int id;
    omp_lock_t lck;           // vytvoří proměnnou zámku lck
    omp_init_lock(&lck);      // inicializace požaduje pointer na lck

    #pragma omp parallel shared(lck) private(id)
    {
        id = omp_get_thread_num();
        while (!omp_test_lock(&lck)) // 0 = nezdar, neblokuje
        {
            // pokud test_lock vrací 0
            work2(id);           // dělej něco jiného work2
        }
        work1(id);              // volný lock získán a nastaven, dělej work1(id) v CS
        omp_unset_lock(&lck);    // lock uvolněn
    }
    omp_destroy_lock(&lck);
}
```

```
omp_lock_t maxlock;          // vytvoř zámek maxlock
omp_init_lock(&maxlock);
...
#pragma omp parallel for
{
    for(i = 0; i < N; i++)
    {
        ...
        omp_set_lock(&maxlock);
        kritická sekce
        omp_unset_lock(&maxlock);
    }
    omp_destroy_lock(&maxlock);
}
```

blokuje, (čeká na
volný maxlock)

kdo získal maxlock,
ten jej uvolní

- Kdykoliv se vlákno pokouší získat exkluzivní přístup ke dvěma nebo více sdíleným prostředkům, může vzniknout deadlock.
- **Deadlock vznikne pokud platí když platí 4 podmínky:**
 1. Přístup ke každému prostředku je exkluzivní.
 2. Vláknu je dovoleno užívat jeden prostředek a zároveň žádat jiný.
 3. Žádné vlákno není ochotno se vzdát prostředku, který získalo.
 4. Existuje cyklická posloupnost vláken snažících se získat prostředky, přičemž každý prostředek je užíván jedním vláknem a žádán jiným.

Deadlocku se lze vyhnout negací jedné z těchto podmínek.

direktiva / dovětek	režie [us]
parallel	1,5
barrier	1,0
schedule static	1,0
schedule guided	6,0
schedule dynamic	50 *)
ordered	0,5
single	1,0
reduction	2,5
atomic	0,5
critical	0,5
lock/unlock	0,5

Intel Xeon quad-core,
3 GHz, kompilátor Intel

*) default chunk size = 1;
pro chunk size = 16 jen
5 μ s .

S počtem vláken roste
režie direktiv parallel
a barrier lineárně.

Pokračování příště