

Programování se sdílenou pamětí

OpenMP sekce a tasky

AVS – Architektury výpočetních systémů
Týden 8, 2024/2025

Jirka Jaroš

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




OpenMP Direktivy kompilátoru

PARALELNÍ SEKCE

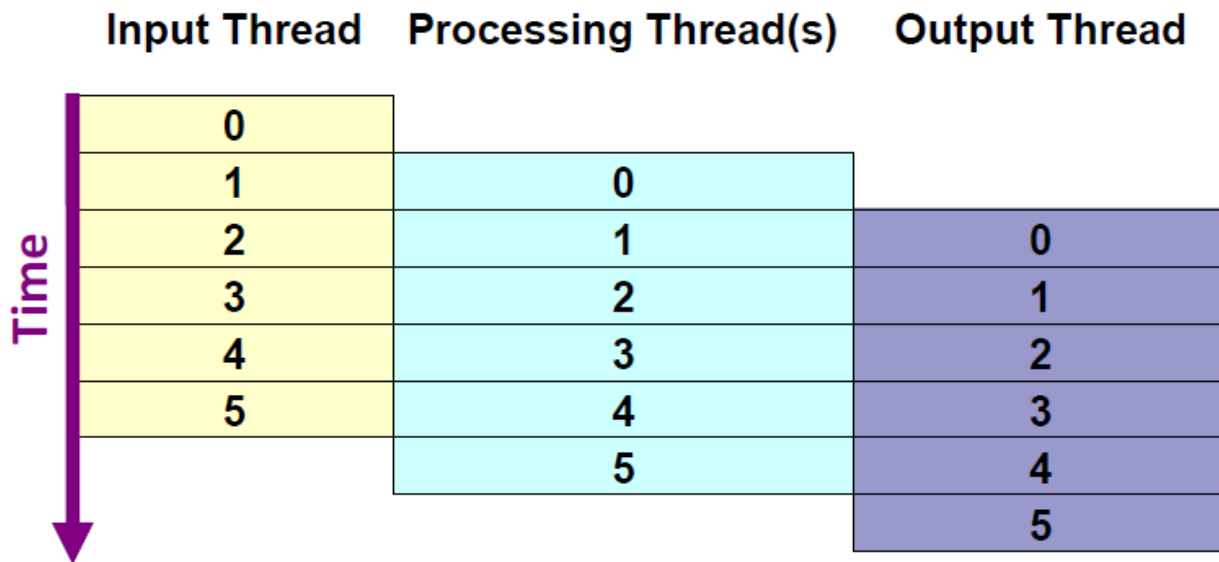
- Sekce určují úseky kódu, které mohou běžet paralelně.
- Každá sekce je provedena **jen jednou** nějakým vláknem v týmu.
- Sekcí může být více než vláken, ale pak nemůžeme mezi sekcemi komunikovat stylem producent -> konzument!
- Počet sekcí nelze měnit dynamicky za chodu programu.

```
#pragma omp sections [clause[ clause] ...]
{
    #pragma omp section
    {work1();}
    #pragma omp section
    {work2();
     work3();}
    #pragma omp section
    {work4();}
} /** implicitní bariéra pokud se nepoužije nowait **/
```



```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait
```

- Pipeline, producent/konzument, Master-Workers, offload zpracování na GPU/XeonPhi...
- Data mezi sekcemi předávána pomocí **sdílených bufferů strážených kritickými sekcemi**.
- Pomocí **nested paralelizmu** je možné přiřadit více vláken jedné sekci.



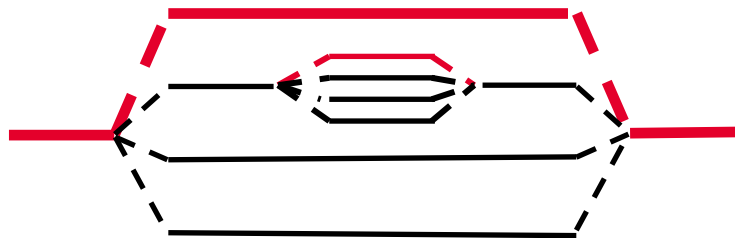
```
#pragma omp parallel sections num_threads(3)
{
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) read_input(i);
            (void) signal_read(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_read(i);
            (void) process_data(i);
            (void) signal_processed(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_processed(i);
            (void) write_output(i);
        }
    }
} /*-- End of parallel sections --*/
```

Input Thread

Processing Thread(s)

Output Thread

- Direktiva **parallel** uvnitř jiného **parallel** stanoví dynamicky nový vnořený tým vláken. U současných implementací je tým omezen na 1 vlákno (default) a úroveň zanoření na 1.
- **Toto chování lze změnit**
 - nastavením proměnné prostředí `OMP_NESTED` na `TRUE`
 - voláním funkce dyn. knihovny `omp_set_nested()` ;
 - aktuální stav lze zjistit pomocí `omp_get_nested()` ;
- **Vlákno které dorazí k direktivě parallel se stává novým master vláknem pro vnořenou část.**
 - Pokud se ve vnořené paralelní oblasti zeptám na počet vláken, získaná hodnota je pro tuto úroveň!
 - Abych zjistil kolik je celkem všech vláken na všech úrovních, musím tuto funkci zavolat všude a pak redukcí sumarizovat, případně pomocí prefix-scanu zjistit svoje globální ID



- Není to nutně hlavním vláknem (master), ale prvním vláknem, které dojde k direktivě **single**.
- Užitečné pro tisk zpráv o postupu řešení, vstup dat, **přidělování tasků**.
- **Obsahuje implicitní bariéru** pokud se nepoužije **nowait**.

```
#pragma omp single [clause[ clause] ...]  
printf ("Work done.\n");
```



```
private (list) , firstprivate (list) ,  
nowait, copyprivate (list)
```



vlákno single zpřístupní svoje privátní
proměnné ostatním vláknům v týmu (broadcast)

```
#include <omp.h>

void input_params(int, int); // Read cmdline params
void do_work(int, int);

void main(){
    int Nsize, choice;

    #pragma omp parallel private(Nsize, choice)
    {
        #pragma omp single copyprivate(Nsize, choice)
            input_params(Nsize, choice);
        // implicitní bariera
        do_work(Nsize, choice);
    }
}
```

Broadcast private
proměnných všem

#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,...

- K příkazu musí dojít všechna vlákna nebo žádné.
- Každé vlákno musí projít sérii příkazů sdílení práce ve stejném pořadí.
- Není dovoleno skočit dovnitř nebo ven z bloku spojeného s tímto příkazem (**jedině exit()**).
- Vnořování příkazů sdílení práce je nelegální.
- Příkazy sdílení práce nesmí uvnitř obsahovat bariéru.
- Mají implicitní bariéru nebo nowait na konci.
- Direktivy sdílení práce mohou být **sirotci**.

- Dovolují během **sériové části** programu automaticky přenastavit **počet vláken tak, aby byl roven počtu použitelných jader** a to dynamicky, podle aktuální zátěže systému např.
v multiprogramovém prostředí, kde běží více aplikací/uživatelů současně.
- Počet vláken se nemůže změnit v době provádění paralelní oblasti.
- Nastavení **počet vláken = počet jader**:
`omp_set_num_threads (omp_get_num_procs ())`
- Nastavení dynamických vláken pro celý program:
`setenv OMP_DYNAMIC (TRUE, FALSE)`
- Povolení/zákaz dynamických vláken za běhu:
`void omp_set_dynamic(int) (0, 1)`
- Dotaz jsou-li dynamická vlákna povolena/zakázána:
`int omp_get_dynamic(void) (0, 1)`

Name	Functionality
<code>omp_set_num_threads</code>	<i>Set number of threads</i>
<code>omp_get_num_threads</code>	<i>Return number of threads in team</i>
<code>omp_get_max_threads</code>	<i>Return maximum number of threads</i>
<code>omp_get_thread_num</code>	<i>Get thread ID</i>
<code>omp_get_num_procs</code>	<i>Return maximum number of processors</i>
<code>omp_in_parallel</code>	<i>Check whether in parallel region</i>
<code>omp_set_dynamic</code>	<i>Activate dynamic thread adjustment</i> <i>(but implementation is free to ignore this)</i>
<code>omp_get_dynamic</code>	<i>Check for dynamic thread adjustment</i>
<code>omp_set_nested</code>	<i>Activate nested parallelism</i> <i>(but implementation is free ignore this)</i>
<code>omp_get_nested</code>	<i>Check for nested parallelism</i>
<code>omp_get_wtime</code>	<i>Returns wall clock time</i>
<code>omp_get_wtick</code>	<i>Number of seconds between clock ticks</i>

OMP_NUM_THREADS	positive number
OMP_DYNAMIC	TRUE or FALSE
OMP_NESTED	TRUE or FALSE
OMP_SCHEDULE	"static,2"
OMP_PROC_BIND	TRUE, FALSE, CLOSE,... (obslužný systém ne/bude přesunovat vlákna mezi jádry)
OMP_DISPLAY_ENV	TRUE or FALSE
plus další ...	

OpenMP Direktivy kompilátoru

TASKY

- **Task** (úloha) má kód, datové prostředí (svoje data), interní řídicí proměnné a přiřazené vlákno.
- Jedno vlákno **může** generovat tasky a ty jsou pak prováděny vlákny týmu v neurčeném pořadí (nezávisle na sobě).
- Umožňuje paralelizaci smyček while, rekurzivních výpočtů, ...

```
#pragma omp task [clause[ clause] ...]  
strukturovaný blok
```



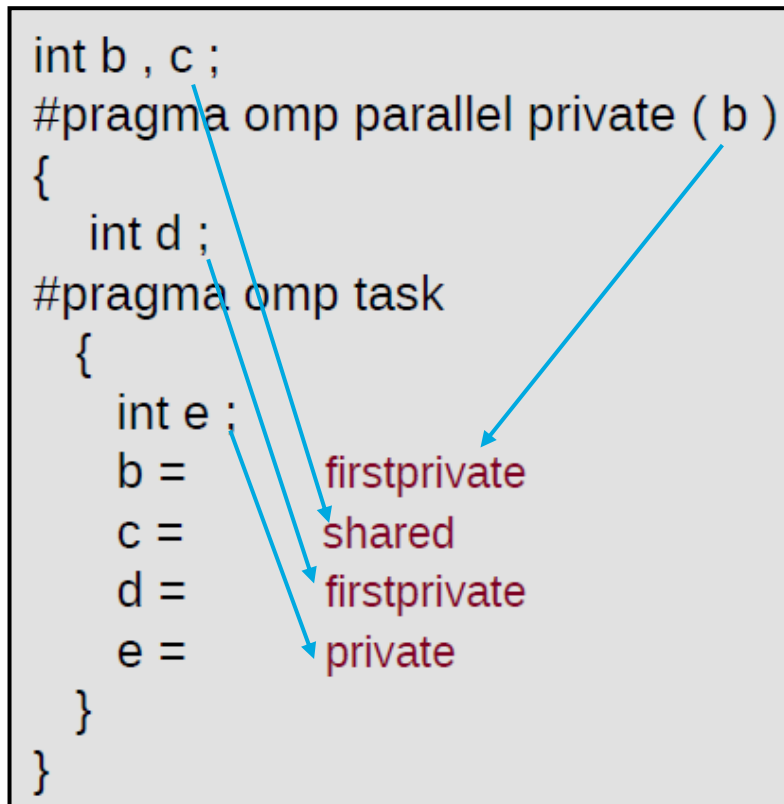
dovětky definují datové prostředí tasku:

```
default (shared | none)  
private (list)  
firstprivate (list)  
shared (list) a další... (mergeable, final, untied)
```

- Vlákno, **které dojde k direktivě task** vygeneruje novou instanci tasku, zabalí kód a data pro provedení.
- Pro každou **private** a **firstprivate** proměnnou **je alokována paměť**, hodnota firstprivate proměnné je inicializována hodnotou původní proměnné před direktivou task.
- **Nějaké vlákno v paralelní oblasti pak provede task**, obslužné prostředí zařídí okamžité nebo pozdější provedení.

- Proměnné, které jsou **privátní** na vstupu do tasku, jsou uvnitř tasku implicitně **firstprivate**.
- **Sdílené** proměnné před direktivou task zůstávají **sdílené** i uvnitř tasku.
- Chceme-li dostat nějaký výsledek z tasku ven, musíme přes sdílenou proměnnou.
- Proměnné deklarované v tasku jsou **privátní**, v tasku – sirotku **firstprivate**.

```
int b , c ;  
#pragma omp parallel private ( b )  
{  
    int d ;  
    #pragma omp task  
    {  
        int e ;  
        b =  
        c =  
        d =  
        e =  
    }  
}
```



```
#pragma omp parallel ← Zde se vytvoří vlákna.  
{  
  #pragma omp master ← Pouze master vlákno projde tuto část  
  {  
    #pragma omp task  
    fred();  
  
    #pragma omp task ← 3 nezávislé úlohy.  
    daisy();  
  
    #pragma omp task  
    billy();  
  } ← Ostatní vlákna nečekají a hned jdou  
} ← zpracovávat.  
} ← Zde je fronta čekajících tasků. V  
   tomto bodě je garantováno  
   dokončení všech tasků.
```



```
my_pointer = listhead;
```

```
#pragma omp parallel
```

```
{
```

```
    #pragma omp single nowait
```

```
{
```

```
    while (my_pointer)
```

```
    {
```

```
        #pragma omp task firstprivate(my_pointer)
```

```
        {
```

```
            do_independent_work (my_pointer);
```

```
        }
```

```
        my_pointer = my_pointer->next;
```

```
    }
```

```
    } // end of single - bariéra potlačena (nowait)
```

```
} // end of parallel region - implicitní bariéra
```

Všechny tasky dokončí zde

Jedno vlákno bude řídit smyčku while,
generovat tasky pro další vlákna
týmu

my_pointer musí být firstprivate,
aby každý task měl def. svou hodnotu

blok 1

blok 2

blok 3

- Na vláknové bariéry (implicitní nebo explicitní)
 - Všechny tasky vygenerované v jednom paralelním regionu, dokončí v tomtéž regionu.
- Na direktivě `taskwait`
 - Čeká se na dokončení všech tasků definovaných v aktuálním tasku.
 - Platí pouze pro tasky vygenerované v tomto tasku, tedy neplatí pro následníky.
- Na konci `taskgroup` regionu
 - Čeká se na všechny tasky generované z tohoto tasku, tedy i následníky.

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        fred();

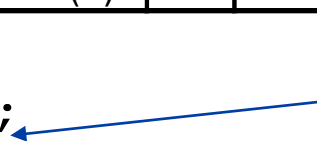
        #pragma omp task
        daisy();

        #pragma omp taskwait
        #pragma omp task
        billy();
    }
}
```

`fred()` a `daisy()` musí dokončit dříve než `billy()` začne

n	0	1	2	3	4	5	6	7	8	9	10
fib(n)	0	1	1	2	3	5	8	13	21	34	55

```
if (n < 2) return n;
```



Sekvenčně:

```
int seqfib(const int n)
{
    int x, y;
    if (n < 2) return n;

    x = seqfib(n - 1);
    y = seqfib(n - 2);
    return x + y;
}
```

Jakmile je dosaženo jisté hodnoty n , je lépe počítat fib(n) sekvenčně a režii tasků v OpenMP vynechat:

if ($n \leq 30$) return seqfib(n);

```
int main (int argc,
          char **argv)
{
    int n, result;
    n = atoi (argv[1]);
    #pragma omp parallel
    {
        #pragma omp single
        {
            result = fib(n);
        }
    }
    printf ("fib(%d)=%d\n",
           n, result);
}
```

```
int fib (int n)
{
    int x, y;
    if (n < 2 ) return n;
    if (n ≤ 30) return seqfib(n);
```

pozastav rodič.
task, až dokončí
dceřiné tasky

```
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib(n-2);
#pragma omp taskwait
    return x+y;
}
```

x+y musí být přístupné –
shared, default je firstprivate

- Závislosti mezi tasky lze specifikovat pomocí sdílených proměnných a klauzule **depend**.
- **Depend (in: var)** – task se nespustí dokud se nevykonají přechozí tasky, které se odkazují na proměnnou var v klauzuli **depend (out: var)** – závislosti na které se má čekat.

```
#pragma omp task depend (out: a)
```

```
{ ... } // writes a
```

```
#pragma omp task depend (out: b)
```

```
{ ... } // writes b
```

```
#pragma omp task depend (in: a, b)
```

```
{ ... } // reads a and b
```

- v OpenMP 4.0 lze čekat i na části polí

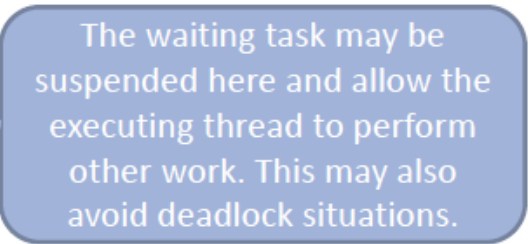
```
#pragma omp task depend (in: a[1:10]))
```

- Umožňuje suspendovat právě probíhající task a nechat běžet jiné

```
#include <omp.h>

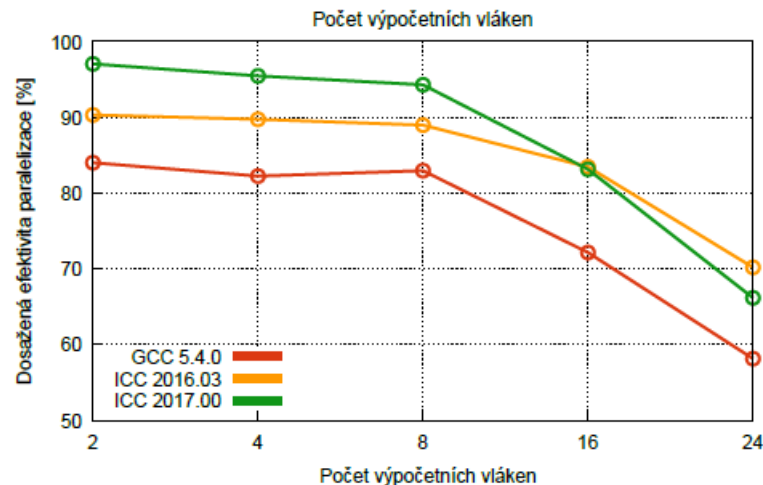
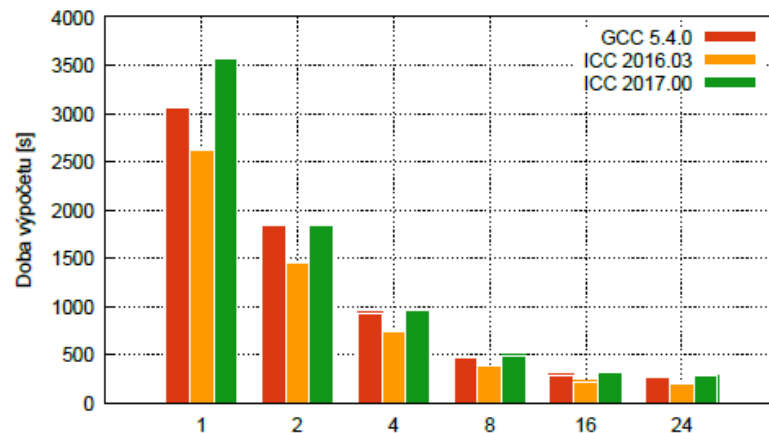
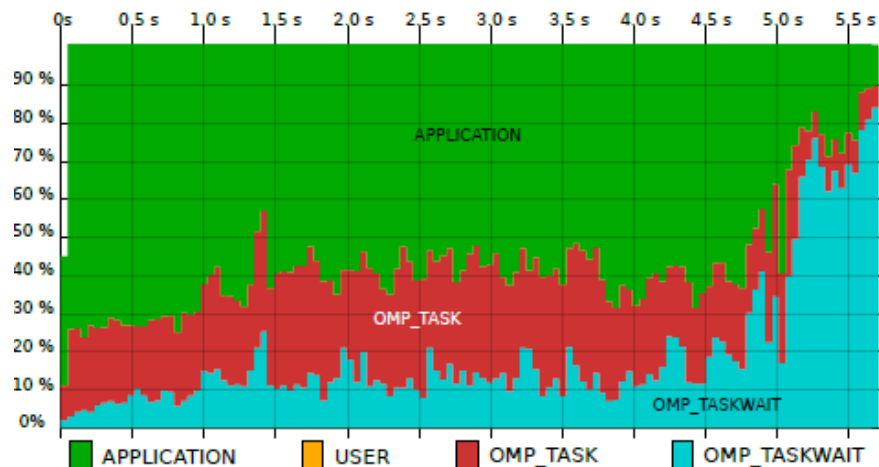
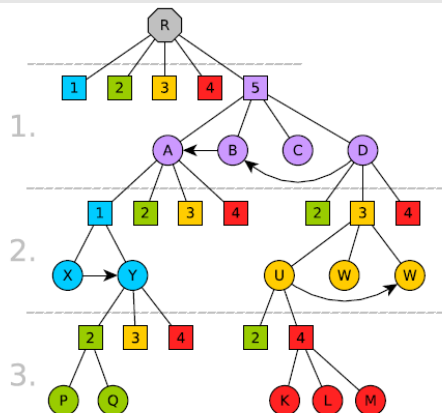
void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
    for(int i = 0; i < n; i++)
        #pragma omp task
        {
            something_useful();
            while( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```



The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations.

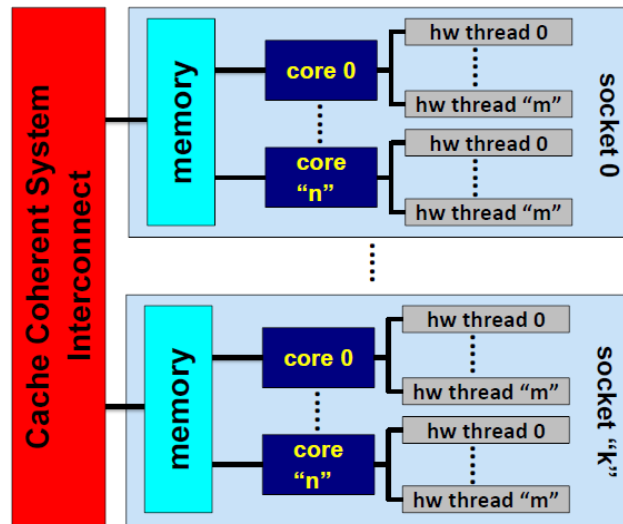
- Tasky jsou implicitně svázány (**tied**) s vláknem, které je jako první začne **vykonávat** – ne to které ho vytvoří!
- Platí následující omezení pro plánování vykonání
 - Pouze vlákno které je svázáno s taskem jej může vykonávat.
 - Task může být suspendován (přerušen) pouze v tzv. scheduling points
 - Vytvoření tasku, ukončení tasku, taskwait, barrier, taskyield.
 - Pokud není vlákno suspendováno na bariéře, může vykonávající vlákno přepnout pouze k přímému potomku všech tasků svázaných s tímto vláknem.
- **Tasky lze vytvářet i v režimu untied**
 - Task může být převzat jiným vláknem na scheduling pointu.
 - Umožňuje vyšší flexibilitu při implementaci a vyvažování zátěže.
 - **Ale:**
 - threadprivate proměnné mohou být nedefinované
 - Ptát se na ID vlákna nedává smysl
 - Pozor na kritické sekce



OpenMP Direktivy kompilátoru

NOVINKY VERZE 4.0/4.5

- Jak mapovat OpenMP vlákna na HW vlákna v systému?
- HW vlákna jsou číslována (/proc/cpuinfo)
- OS může libovolně migrovat OpenMP vlákno přes HW vlákna.
- Vlákna lze i **zamknout** v rámci tzv. `OMP_PLACES` pomocí `OMP_PROC_BIND` nebo `proc_bind` klauzule.
- Existují 3 abstraktní místa (thread, core, socket)
- Příklady:
 - `OMP_PLACES = "{0, 1, 2} , {5, 6, 7}"`
 - `OMP_PLACES = "{0-7}, {8-15}"`
 - `OMP_PLACES = threads | cores | sockets`

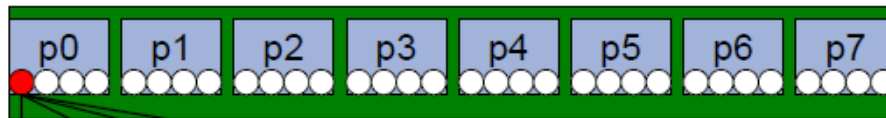


- Proměnná OMP_PROC_BIND udává jak se OpenMP vlákna distribuují přes OMP místa.
 - Master** – Každé vlákno v týmu je přiřazeno na stejné místo jako master vlákno.
 - Close** – Přiřadí vlákna z týmu na místa blízká rodičovskému vláknu (blížkost je definována pořadím míst v OMP_PLACES).
 - Využití – využít nejprve vlákna na jednom HW jádře/socketu.
 - Spread** – Round robin distribuce vláken přes všechna místa.
 - Použití všech jader/socketů v systému

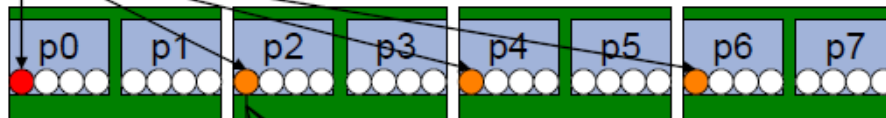
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4 = cores  
#pragma omp parallel proc_bind(spread)  
#pragma omp parallel proc_bind(close)
```

Example

→ initial



→ spread 4



→ close 4



- **Cancel** – provede přerušení OpenMP regionu

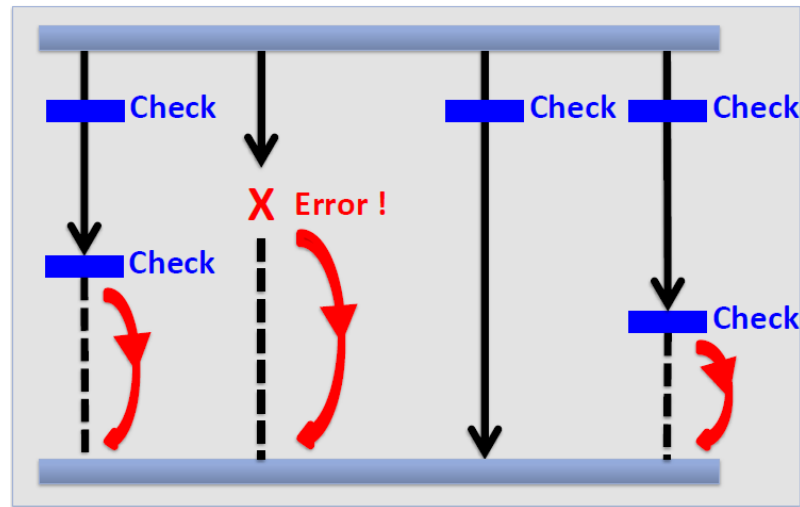
- Vlákno (úloha), které je přerušeno pokračuje ke konci přerušného regionu
- `#pragma omp cancel <construct>`
`[if-clause]`

- **Cancellation point** – Bod, kde vlákna / úloha testuje jestli má být vykonávání přerušeno

- Buď přerušení nebo pokračování
- Mezi tyto body patří implicitní bariéry, regiony mezi bariérami, cancel regiony, cancellation pointy
- `#pragma omp cancellation point <construct>`

Podporované konstrukty

- Parallel
- Sections
- For
- Taskgroup



```
bool has_zero = false;
#pragma omp parallel default(none) shared(matrix, has_zero)
{
    #pragma omp for
    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < cols; col++)
        {
            if (matrix(row, col) == 0)
            {
                #pragma omp critical
                {
                    has_zero = true;
                }
                #pragma omp cancel for
            }
        }
    }
    #pragma omp cancellation point for
}
```

Zde přeručíme výpočet

Zde se ostatní vlákna dozví, že mají ukončit výpočet

```
void search_parallel(btree *t,
element e, bool* present)
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
                search(t, e, present);
        }
    }
}
```

Taskgroup zajistí
zrušení všech
vygenerovaných tasků
v této oblasti

```
void search (btree *t, element e, bool*
present)
{
    if (t->element == e)
    {
        #pragma omp cancel taskgroup
        *present = true;
    }
    if (t->left)
    {
        #pragma omp task
        search(t->left, e, present);
    }
    if (t->right)
    {
        #pragma omp task
        search(t->right, e, present);
    }
}
```

Task se vždy dokončí celý.

Pokračování příště