

# Programování se sdílenou pamětí

## Úvod do OpenMP a smyčky for

### AVS – Architektury výpočetních systémů

#### Týden 7, 2024/2025

**Jirka Jaroš**

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



- Tři paralelní programovací modely
  - Abstrakce předkládané programátorovi
  - Ovlivňují jak programátoři přemýšlejí při psaní programů
- Tři architektury strojů
  - Abstrakce hardware pro software na nízké úrovni
  - Typicky odpovídají implementaci

## 1. Sdílený adresový prostor

(Shared Address Space, SAS)

(MIMD, SPMD)

## 2. Zasílání zpráv

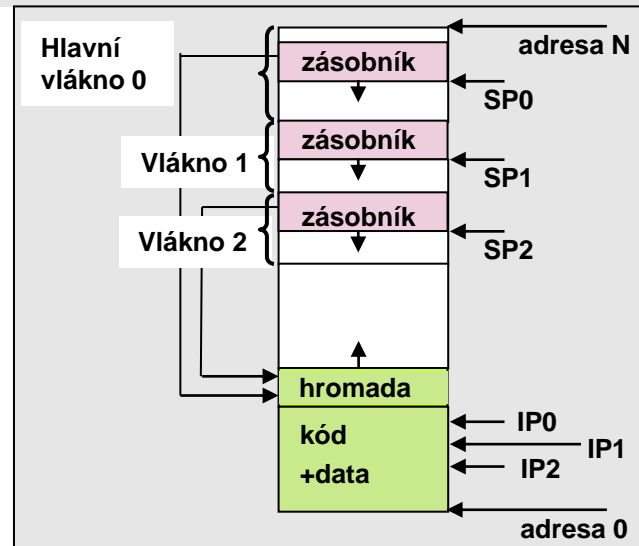
(Message Passing, MP)

(MIMD, SPMD)

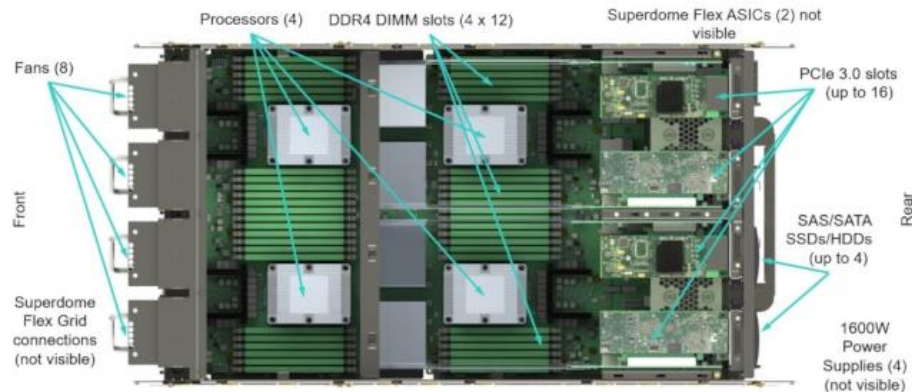
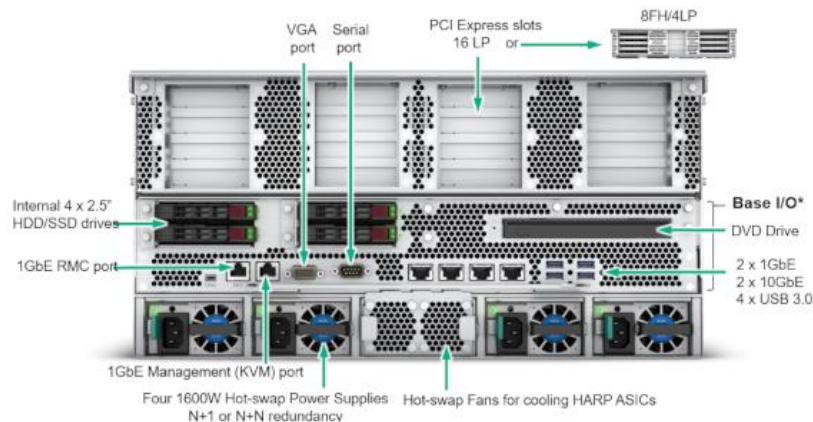
## 3. Datově-paralelní model

(SIMD, SIMT)

- Vlákna **komunikují**:
  - **čtením/zápisem do sdílených proměnných**
    - Komunikace je implicitní v paměťových operacích – snadnější programování.
    - Vlákno 1 zapíše do X, vlákno 2 čte X (vidí update).
    - To vyžaduje synchronizaci.
  - **manipulací synchronizačních primitiv**
    - např. vzájemné vyloučení použitím zámků
- Přirozené rozšíření sekvenčního programovacího modelu
- Sdílené proměnné jako velká nástěnka
  - každé vlákno může číst nebo zapisovat
  - problém se **současným přístupem R+W** ke sdíleným datům



- HPE Superdome Flex server
- 32x Intel® Xeon® Platinum, 24-core, 2.9 GHz, 205W
- 24 TB DDR4 2993MT/s of physical memory per node
- 2x 200 Gb/s IB port
- 71.2704 TFLOP/s
- <https://www.youtube.com/watch?v=a6XBOCQfcF4>



1. **Sekvenční jazyk** + **příkazy** paralelního zpracování, komunikace a synchronizace (např. Universal Parallel C)
2. **Sekvenční jazyk** + **dynamické knihovny**:
  - programování s vlákny (Pthreads, Java threads, WindowsThreads ..., Intel Threading Building Blocks)
  - nebo zasílání zpráv MPI (Message Passing Interface)
3. **Sekvenční jazyk** + **direktivy pro kompilátor** (pragma).
  - Umožňuje sekvenční i paralelní zpracování a inkrementální postup při paralelizaci
4. **OpenMP API**: kombinace direktiv a knihovních programů.

- **Vlákná jsou ovládaná a použita kernelem OS**

- Uživatel má k dispozici knihovny vláken, např. POSIX threads, Windows threads.
- nebo API (**OpenMP**, OpenACC, CUDA, OpenCL).
- OS mapuje (plánuje) SW vlákna na HW vlákna.

- **HW vlákno**

- Prostředky provádějící vlákno nezávislé na jiných HW vláknech.
- Vlákna 1 procesu mohou běžet na různých jádrech.
- **Volání systému nebo knihovných programů** musí být pro vlákna bezpečné (**thread-safe**), tj. správnost je zajištěna i při volání z více vláken současně.

- Vyvinuto konsorciem (OpenMP ABR) hlavních výrobců HW a SW pro **paralelní výpočty se sdílenou pamětí** (procesory i akcelerátory)
- Existuje API pro C/C++ a pro Fortran
  - První verze v roce 1997.
  - Současná verze 5.0 (2018).
  - Verze 5.2 ve stádiu implementace.
- **Anglické tutoriály:**
  - OpenMP 3.0:
    - <https://computing.llnl.gov/tutorials/openMP/>
  - OpenMP 4.0:
    - <http://wiki.scinethpc.ca/wiki/images/9/9b/Ds-openmp.pdf>
  - Kompletní dokumentace 5.x, poslední vývoj, user's group:
    - <http://www.openmp.org/specifications/>
    - <http://compunity.org>



- **OpenMP podporuje:**

- jemný a hrubý paralelismus (smyčky, paralelní sekce)
- datový a funkční paralelismus (SPMD, paralelní sekce)

- **OpenMP dovoluje:**

- inkrementální paralelizaci programů se sdílenou pamětí
- psát **přenosné a částečně škálovatelné** programy
- psát paralelní programy se zabudovanou **sekvenční** verzí
- ověřit **správnost** programů

- **OpenMP zjednodušuje:**

- psaní vícevláknových programů ve Fortranu, C a C++
- ve verzi 4.0 existuje přidává vektorizaci SIMD (AVX)
- od verze 4.0 podporuje programování akcelérátorů (Xeon Phi, GPU)



- Vlákna OpenMP jsou abstrakcí
  - implementace je může mapovat na vlákna kernelu OS, lehká vlákna POSIX (P-threads), Win32\* threads apod.
- OpenMP je nezávislé na stroji a OS
  - přenos správného programu jinam vyžaduje „jen“ rekompilaci
  - **OpenMP-kompatibilní** (-aware, -compliant) kompilátory existují pro všechny hlavní verze Unixu, Linuxu, Windows
  - Zapnutí OpenMP (gcc -fopenmp, intel -qopenmp)
- Implementace OpenMP pro C/C++ poskytují hlavičkový soubor soubor **omp.h** s definicemi a prototypy funkcí.

- Direktivy pro kompilátor

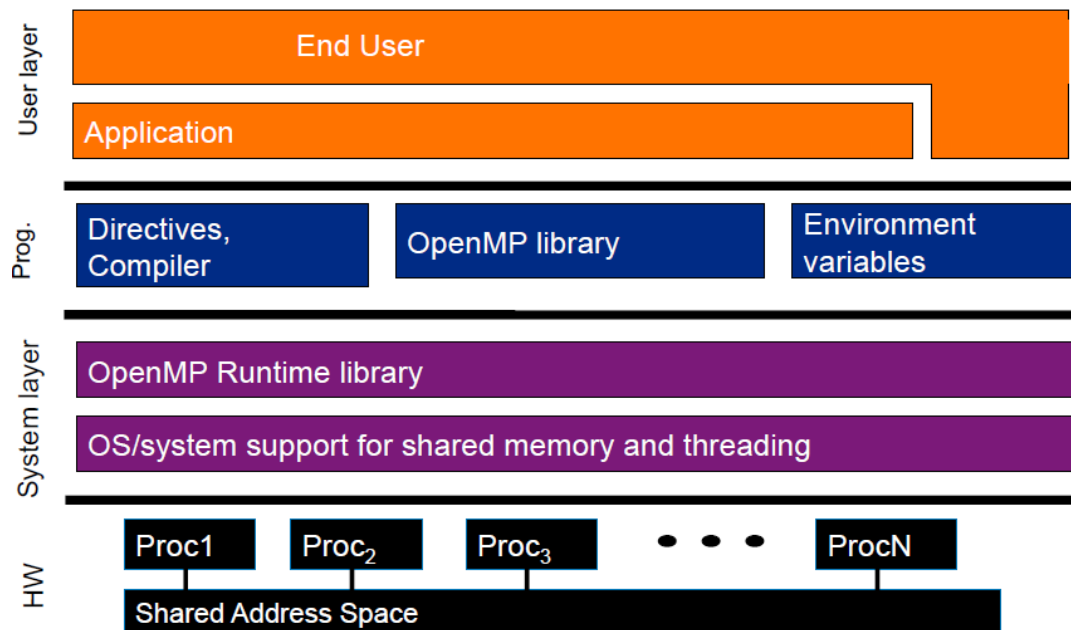
- `#pragma omp ...`
- vytvoření týmu vláken
- sdílení práce mezi vlákny
- synchronizace vláken

- Knihovny rutiny

- pro nastavení atributů vláken a dotazy na ně
  - `omp_set_num_threads(...)`,
  - `omp_get_thread_num()`,
  - `omp_get_num_threads(),...`

- Proměnné prostředí

- řízení chování paralelního programu za jeho běhu (`OMP_NUM_THREADS`, `OMP_PLACES`, `OMP_PROC_BIND`, `OMP_ENV_DISPLAY`, ...)
- <https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf>



- **Vlákn**

- IP + SP, registry, privátní zásobník.
- Na svou identitu se může dotázat.

- **Je-li více vláken než logických jader** (oversubscription)

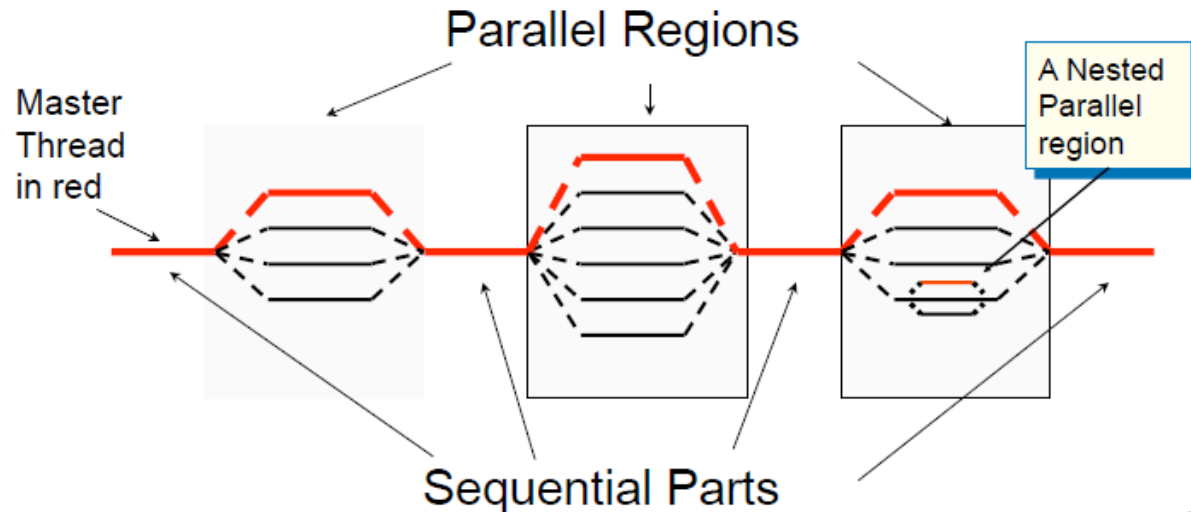
- Pak se vlákna mapují na jádra dynamicky (Web servery).
- Lépe 1 vlákno na 1 (logické) jádro, a staticky bindovat. Odpadá pak režie plánování vláken, vyplachování cache a další problémy.

- **Program OpenMP**

- Se začne vykonávat jedním (hlavním) vláknem (**master thread**).
- Jakmile dojde k **paralelnímu příkazu**, vytvoří se tým vláken (**worker threads**).
- Členové týmu provádějí příkazy paralelně a synchronizují se na **bariéře**.
- Program se může při běhu takto **větvit a zase spojit mnohokrát** (model fork–join).

- **Strukturovaný blok:**

- blok jednoho nebo { více příkazů; } s jedním vstupem nahoře a jedním výstupem dole.
- Může obsahovat `exit()` uvnitř.
- Každá OpenMP direktiva platí pouze pro následující strukturovaný blok.



Direktivy kompilátoru

# VYTVOŘENÍ TÝMU VLÁKEN

- Direktiva **parallel**

- definuje paralelní oblast (region)
- vytvoří tým vláken
- zahájí paralelní zpracování
  - strukturovaný blok provádějí **všetchna vlákna**, každé vlákno specificky podle svého ID (→ SPMD)
- hlavní (master) vlákno má číslo 0
- na konci paralelní oblasti je **automaticky bariéra**

nastavení počtu vláken

```
01  omp_set_num_threads (4);
02  double a[1000];
03  #pragma omp parallel [clause[clause]... ]
04  {
05      int id = omp_get_thread_num();
06      work(id, a);
07  }
```

Dovětky

Strukturovaný blok  
vykonávaný 4 vlákny

```
private(list)
shared(list)
default(shared|none)
reduction(operator:list)
if(logical expression)
copyin(list)
num_threads(thread_count)
```

- Proměnné deklarované před paralelní oblastí musí být v jejím **lexikálním (textovém) rozsahu** označeny private nebo shared
  - bez označení budou shared (pozor na hazardy -> vzájemné vyloučení)
- **if**: vytvoření týmu vláken lze podmínit
  - podle výsledku nějakého testu uživatele (např. počet iterací), se pracuje sekvenčně if (FALSE) nebo paralelně if (TRUE)

- **Počet je určen následujícími faktory, pořadí podle priority:**
  - Vyhodnocení dovětky `if` (jestli vůbec paralelně).
  - Nastavení dovětkem `num_threads(thread_count)` v dané paralelní oblasti.
  - Použitím knihovní funkce `omp_set_num_threads(int)` pro všechny následující paralelní oblasti.
  - Nastavením proměnné prostředí pro celý program  
`export OMP_NUM_THREADS=12`
  - Implementační implicitní hodnota = obvykle počet CPU (v tomto počtu jsou tvořena i „dynamická“ vlákna).
  - Počet vláken v týmu zůstává uvnitř paralelní oblasti konstantní. Pro další oblast jej může změnit:
    - uživatel (`void omp_set_num_threads(int)`)
    - obslužný systém (dynamická vlákna)



- Dovětek **private (var)**
  - Vytvoří novou lokální kopii proměnné *var* pro každé vlákno
  - Hodnoty privátních proměnných **nejsou** v paralelní oblasti **inicializovány** na hodnotu původní sdílené proměnné *var* (C++ volá **defaultní constructor**)
  - Pokud potřebujeme okopírovat původní hodnotu do lokálních proměnných (C++ copy constructor), použijeme direktivu **firstprivate**
  - Za paralelní oblastí nejsou privátní proměnné dostupné – obnoví se hodnota původní vnější proměnné před paralelní oblastí
  - Pokud potřebujeme přenést hodnotu privátních proměnných zpět do původní, můžeme využít buď **reduction** nebo **lastprivate**
- **Privatizovat** se mohou **jen úplné objekty**, nikoliv prvky polí nebo části datových struktur.
- **Proměnné** deklarované **uvnitř paralelní oblasti** jsou také **privátní** (bez dovětku)!

```
01 #include <omp.h>
02 #include <stdio.h>
```

```
03 void main() {
04     int tid, nt ;
```

**Direktiva paralelní oblasti,  
počet vláken implicitní**

```
05     #pragma omp parallel private(tid, nt)
```

```
06     {
```

**Kolik nás je v týmu?**

```
07         nt = omp_get_num_threads();
```

```
08         tid = omp_get_thread_num();
```

```
09     }
```

**Moje ID**

```
10         printf("Hello from thread %d \
                out of %d \n", tid, nt);
```


```
11     }
```

**Konec paralelní oblasti**

```
12 }
```

**Tento strukturovaný  
blok provádí každé  
vlákno! (SPMD)**

## možné výstupy (4 vlákna)



```
graph TD; A[možné výstupy<br/>(4 vlákna)] --> B[Hello from thread 1 out of 4<br/>Hello from thread 2 out of 4<br/>Hello from thread 0 out of 4<br/>Hello from thread 3 out of 4]; A --> C[Hello from thread 3 out of 4<br/>Hello from thread 1 out of 4<br/>Hello from thread 2 out of 4<br/>Hello from thread 0 out of 4];
```

Hello from thread 1 out of 4  
Hello from thread 2 out of 4  
Hello from thread 0 out of 4  
Hello from thread 3 out of 4

Hello from thread 3 out of 4  
Hello from thread 1 out of 4  
Hello from thread 2 out of 4  
Hello from thread 0 out of 4

Vlákna vypisují v náhodném pořadí, tak jak dospěly k funkci printf.

```
int x = 5, y = 6, z = 7;
float a[10], b[10], c[10];
#pragma omp parallel num_threads(5) \
    private(x, a) \
    firstprivate(y, b) \
    shared(z, c) \
{
    int thread_id = omp_get_thread_num();
    x++; y++; z++;

    a[thread_id] = 0;
    b[thread_id] = 1;
    c[thread_id] = 2;

    a += thread_id; *a = 5;
    b += thread_id; *b = 5;
    c += thread_id; *c = 5;
}
```

Co bude v proměnných?

Kam zapíšeš?

Kam zapíšeš?

- Program v OpenMP v sobě může mít zabudovávající sekvenční verzi:

## Kompilátor s OpenMP

```
#pragma omp
```

se bere jako direktiva OpenMP

```
#ifdef _OPENMP
```

příkazy, které se mají  
provést jen v paralelní  
verzi (např. knih. fce)

```
#endif
```

## Kompilátor bez OpenMP

```
#pragma omp
```

pragma je ignorováno

```
#ifdef _OPENMP
```

tyto příkazy se v sekvenční  
verzi vynechají

```
#endif
```

- Makro `_OPENMP` = yyyyymm = rok a měsíc schválené specifikace OpenMP.


```
#include <stdio.h>
#ifdef _OPENMP
    #include <omp.h>
#endif

void main(){
    int tid = 0, nt = 1;
    #pragma omp parallel private(tid, nt)
    {
        #ifdef _OPENMP
            nt = omp_get_num_threads();
            tid = omp_get_thread_num();
        #endif
        printf("Hello from thread %d out of %d \n", iam, nt);
    }
}
```

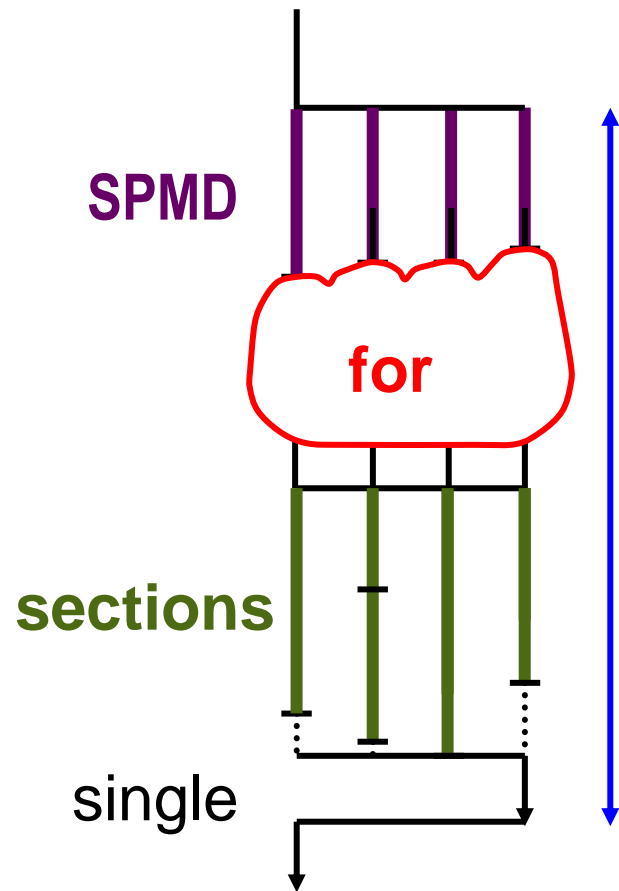
- Direktivy platí pouze v jejich statickém, tj. **lexikálním rozsahu**
  - **Statický nebo lexikální rozsah**: kód textově uzavřen mezi začátek a konec strukturovaného bloku následujícího za direktivou
  - **Dynamický rozsah**: statický rozsah rozšířený o procedury a funkce volané zevnitř statického rozsahu
  - Direktiva, která je v dynam. rozsahu jiné direktivy, ale ne v jejím statickém rozsahu, se nazývá **sirotek** (orphan).

```
#pragma omp parallel
{
    void dowork();
}
```

```
void dowork ()
{
    #pragma omp for
    for (int i=0; ... )
        ...
}
```



- Direktiva **parallel** sama o sobě představuje program **SPMD**, tj. každé vlákno provádí redundantně stejný kód.
- Jak diverzifikovat práci vláken v týmu?  
Pomocí direktiv **uvnitř** paralelní oblasti pro sdílení práce (work sharing):
  - ve smyčce **for**
  - v **sekcích**
  - 1 vláknem (single)
  - v úlohách (task)





**Direktivy kompilátoru**

# **PARALELIZACE SMYČEK**

Sekvenční kód

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP,  
jen direktiva parallel

```
#pragma omp parallel
```

```
{
```

```
    int id, i, Nthrds, istart, iend;
```

```
    id = omp_get_thread_num();
```

```
    Nthrds = omp_get_num_threads();
```

```
    istart = id * N / Nthrds;
```

```
    iend = (id+1) * N / Nthrds;
```

```
    if (id == Nthrds-1)iend = N;
```

```
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
```

```
}
```

OpenMP,  
Direktiva parallel a for


```
#pragma omp parallel
```

```
#pragma omp for
```

```
    for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

**#pragma omp for** [clause[ clause] ...]

blok



```
private(list)           reduction(operator: list)
firstprivate(list)       schedule(kind[, chunk_size])
lastprivate(list)        ordered
nowait ... na konci smyčky není implicitní bariéra
```

- počet iterací musí být znám na vstupu do smyčky
- všechny iterace se musí dokončit, blok: 1 vstup, 1 výstup
- schedule kind: **static**, **dynamic**, **guided**, **runtime**, **auto**
  - Norma OpenMP 4.5 zavádí modifikátor chunku **simd**, **monothonic**, **nonmonothonic**.  
Např. **schedule (simd:static)**
- **vnořené smyčky**:
  - paralelně se dělá jen smyčka nejbližší direktivě
  - její index je implicitně privátní

```
#pragma omp parallel for
for (int y = 0; y < 25; ++y)
{
    #pragma omp for
    for (int x=0; x < 80; ++x)
    {
        tick(x,y);
    }
}
```

vnoření paralelních  
smyček je nelegální

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < 25; ++y)
    for (int x = 0; x < 80; ++x)
    {
        tick(x,y);
    }
```

novější dovětek **collapse**  
vytvoří jednu smyčku ze  
2 vnořených a tu paralelizuje.

**Pozor:** Zavádí režii výpočtu  
indexů x a y

Tyto zápisy jsou ekvivalentní:

```
double res[MAX]; int i;  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i = 0; i < MAX; i++)  
        res[i] = huge();  
}
```

```
double res[MAX]; int i;  
#pragma omp parallel for  
    for (i = 0; i < MAX; i++)  
        res[i] = huge();
```

když paralelní oblast obsahuje  
jen `#pragma omp for`

Podobně lze zkrátit

`#pragma omp parallel sections`

```
void mxv(int n, int m, double* a, double* b[],
        double* c)
```

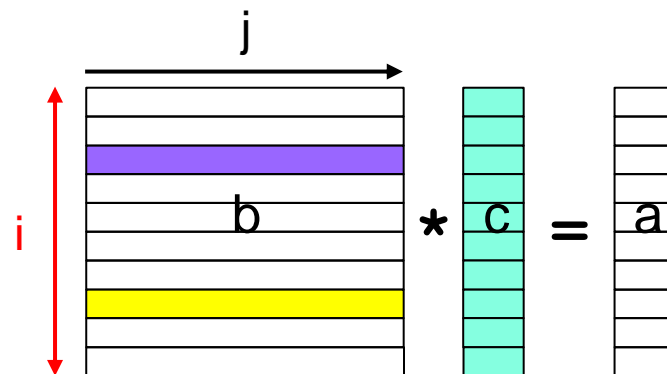
```
{
    #pragma omp parallel for default(none)\
        shared(m,n,a,b,c)
```

```
    for (int i=0; i < n; i++)
```

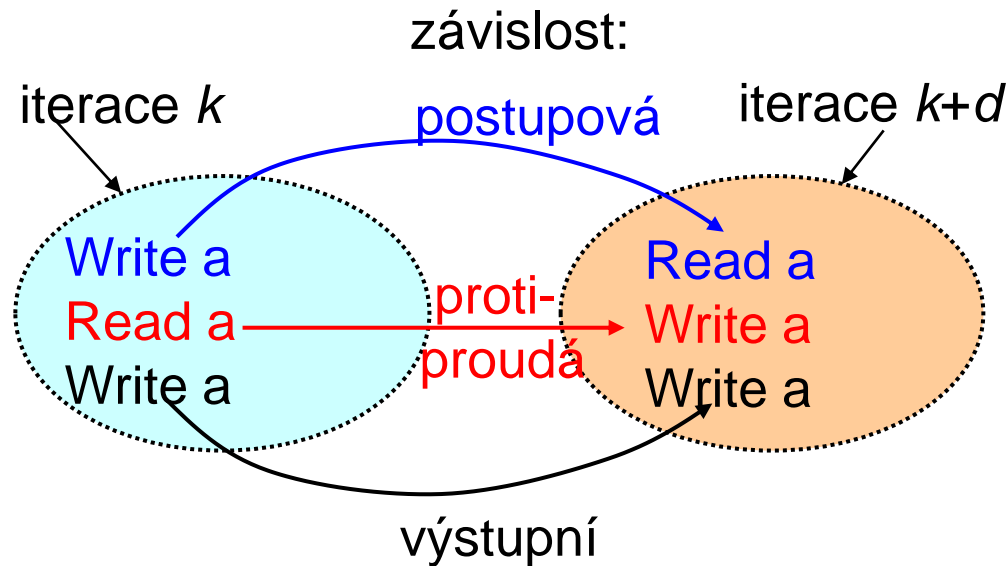
paralelní smyčka

```
    {
        double sum = 0.0;
        for (int j=0; j < m; j++)
            sum += b[i][j]*c[j];
        a[i] = sum;
    }
```

sekvenční smyčka



Bude-li  $n = 10$  řádků a 2 vlákna,  
pak thread 0 si vezme řádky  $i = \langle 0, 4 \rangle$   
a thread 1 řádky  $i = \langle 5, 9 \rangle$ .



- Neznáme pořadí provedení iterací!

- Iteraci  $k$  provádí jedno vlákno v čase  $t$ , ale iteraci  $k+d$  může provádět jiné vlákno před nebo po iteraci  $k$  (nejisté!)
- Kompilátory OpenMP **nekontrolují závislosti** mezi iteracemi.
- Když výsledek **iterace závisí na jiných iteracích**, nelze přímo paralelizovat.

Jak udělat iterace smyčky nezávislé, aby mohly být prováděny v libovolném pořadí bez závislostí?

Příklad:

```
for (i=2; i<=m; i++)  
    for (j=1; j<=n; j++)  
        a[i][j] = 2 * a[i-1][j];
```

Závislost v indexu **i** přemístíme do sekvenční smyčky, tam nevadí:

```
int i, j;  
#pragma parallel for private(i)  
for (j=1; j<=n; j++) ← paralelní, j je impl. privátní  
    for (i=2; i<=m; i++) ← sekvenční, i je privátní  
        a[i][j] = 2 * a[i-1][j];
```

Pozor ale na lokalitu dat!



- **reduction** (*op: list*) provádí redukci skalárních proměnných na seznamu *list*, operátorem *op*
- je vytvořena **privátní** kopie každé proměnné ze seznamu *list* (jedna pro každé vlákno) a inicializována.
- Na konci je **atomicky** aktualizována každým vláknem **stejnomená sdílená proměnná**.

```
double ave = 0.0, A[MAX]; int i;
```

```
#pragma omp parallel for \  
    reduction (+: ave)
```

```
for (i = 0; i < MAX; i++)
```

```
    ave += A[i];
```

```
ave = ave / MAX
```

může být  
seznam

lze specifikovat  
více redukcí  
najednou

Operátor	Poč. hodnota
+	0
*	1
-	0
min	max. možná
max	min. možná

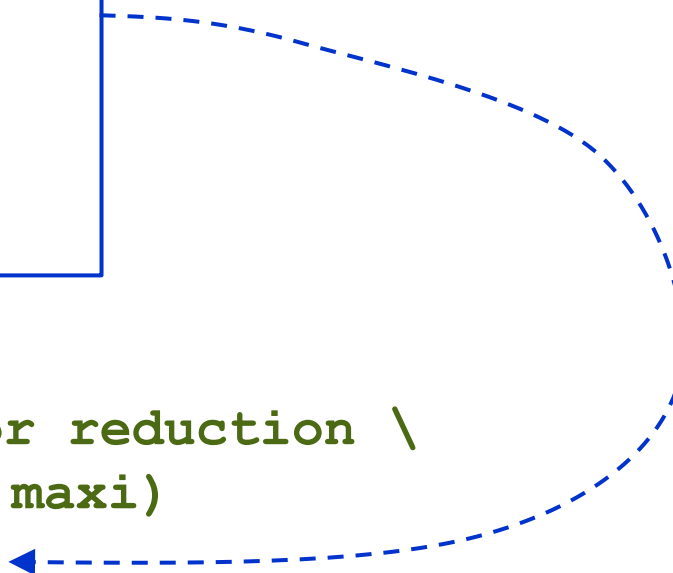
Operátor	Poč. hodnota
&	$\sim 0$
	0
^	0
&&	1
	0

## OpenMP 4.0:

- Rozšíření o redukce definované uživatelem  
**#pragma omp declare reduction ...**

```
int mini = a[0];  
int maxi = a[0];  
for (i=1; i<n; i++)  
{  
    if (a[i] < mini)  
        mini = a[i];  
    if (a[i] > maxi)  
        maxi = a[i];  
}
```

```
int mini, maxi;  
# pragma omp parallel for reduction \  
    (min:mini, max:maxi)
```



OpenMP smyčky

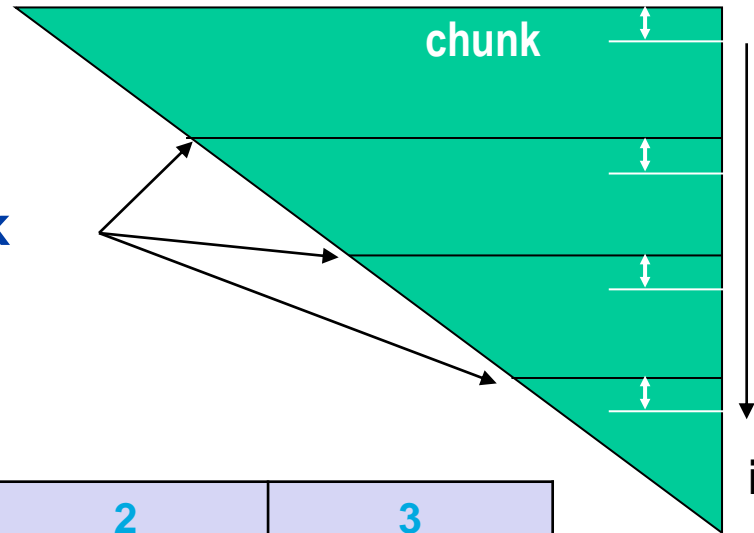
# PLÁNOVÁNÍ ITERACÍ

```
#pragma omp for schedule(static[, chunk_size])  
for (i=0; i<n; i++) {  
    the_same_work(i);  
}
```

- nejméně režie při běhu, naplánování provedeno v době kompilace.
- když není udán `chunk_size`, dostane každé vlákno cca stejnou porci iterací  $q$ .
  - Je-li  $n = t * q - r$ , pak některá vlákna dostanou méně než  $q$  ( $r$ -krát  $q - 1$  nebo 1 krát  $q - r$ )
- je-li udán `chunk_size`, jsou porce přiděleny vláknům cyklicky (prokládané plánování). Užitečné, když se práce v iteracích mění lineárně.
- Jeli udán modifikátor `SIMD`, je `chunk_size` zarovnan na velikost SIMD registru.

```
#pragma omp for schedule(static[, chunk_size])  
  for (i=0; i < n; i++) {  
    the_same_work(i);  
  }
```

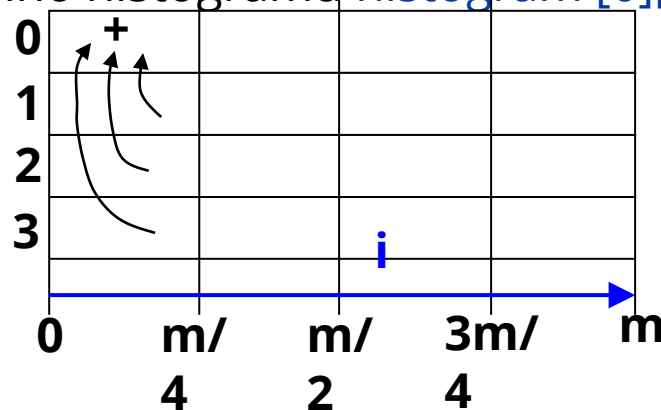
4 vlákna bez chunk



**Příklad:** 16 iterací, 4 vlákna:

Thread ID	0	1	2	3
no chunk	0–3	4–7	8–11	12–15
chunk=2	0–1 8–9	2–3 10–11	4–5 12–13	6–7 14–15

- Vektor  $a[n]$  má prvky integer v intervalu  $\langle 0, m-1 \rangle$ . Četnost výskytu různých hodnot se má znázornit histogramem  $\text{histogram}[m]$ .
  - Nej dřív každé z  $P$  vláken vytvoří vlastní histogram  $\text{histogram}[\text{myid}][m]$ ,  $\text{myid} = 0, \dots, nt-1$  ze svého segmentu vektoru  $a[n]$
  - pak každé vlákno počítá jednu část všech  $P$  histogramů do části jednoho globálního histogramu  $\text{histogram}[0][m]$ .



= histogram[0][m] ( $t_0$ )  
= histogram[1][m] ( $t_1$ )  
= histogram[2][m] ( $t_2$ )  
= histogram[3][m] ( $t_3$ )

```
int histogram [10][m];           // až 10 vláken
#pragma omp parallel shared (a, m, n)
{
    int nt, myid, i, k;
    nt = omp_get_num_threads();
    myid = omp_get_thread_num();
    #pragma omp for schedule(static) num_threads(10)
    for (i = 0; i < n; i++)        // segmenty vektoru
        histogram[myid][a[i]]+=1;
    #pragma omp for schedule(static) num_threads(2)
    for (i = 0; i < m; i++) {      // segmenty histogramů
        for (k = 1; k < nt; k++)   // vláken k > 0 do k = 0
            histogram[0][i]+=histogram[k][i];
    }
}
```



```
#pragma omp for schedule(dynamic[, chunk_size])  
for (i = 0; i < n; i++) {  
    unpredictable_amount_of_work(i);  
}
```

- iterace jsou přidělovány po blocích ( $\text{chunk\_size} \geq 1$ ), se **synchronizací (exkluzivním přístupem k indexu) při každém přidělení** → největší režie za běhu
- default  $\text{chunk\_size} = 1$ ; čím je  $\text{chunk\_size}$  větší, tím je menší synchronizační režie ale hrubší vyvážení zátěže.
- vlákno čeká na bariéře nejvýš dobu kterou trvá jinému vláknu dokončit jeho posledních  $\text{chunk\_size}$  iterací.

```
#pragma omp for schedule(guided[, chunk_size])  
  for (i = 0; i < n; i++) {  
    the_similar_work(i)  
  }
```

- menší synchronizační režie než **dynamic**;
- typická implementace přidělí prvnímu volnému vláknu porci  $q_0 = \lceil n / t \rceil$  iterací
- následující porce se přidělují podle vztahu  $q_i = \lceil q_{i-1}(1 - 1/t) \rceil$   
(minimálně chunk\_size, default chunk\_size je 1);
- pro další iterace jde vlákno, které právě dokončí předchozí porci iterací.

**Příklad:** 200 iterací. Postupně odebrané porce:

$$50 = \lceil 200 / 4 \rceil,$$

$$38 = \lceil 50(1 - 1 / 4) \rceil,$$

$$29 = \lceil 38(1 - 1 / 4) \rceil,$$

$$22 = \lceil 29(1 - 1 / 4) \rceil, \text{ vlákno skončilo 1. porci iterací,}$$

17, ... dostane 2. porci

13, .... vlákno přichází pro 2. porci,

10, ... vlákno přichází pro 2. porci,

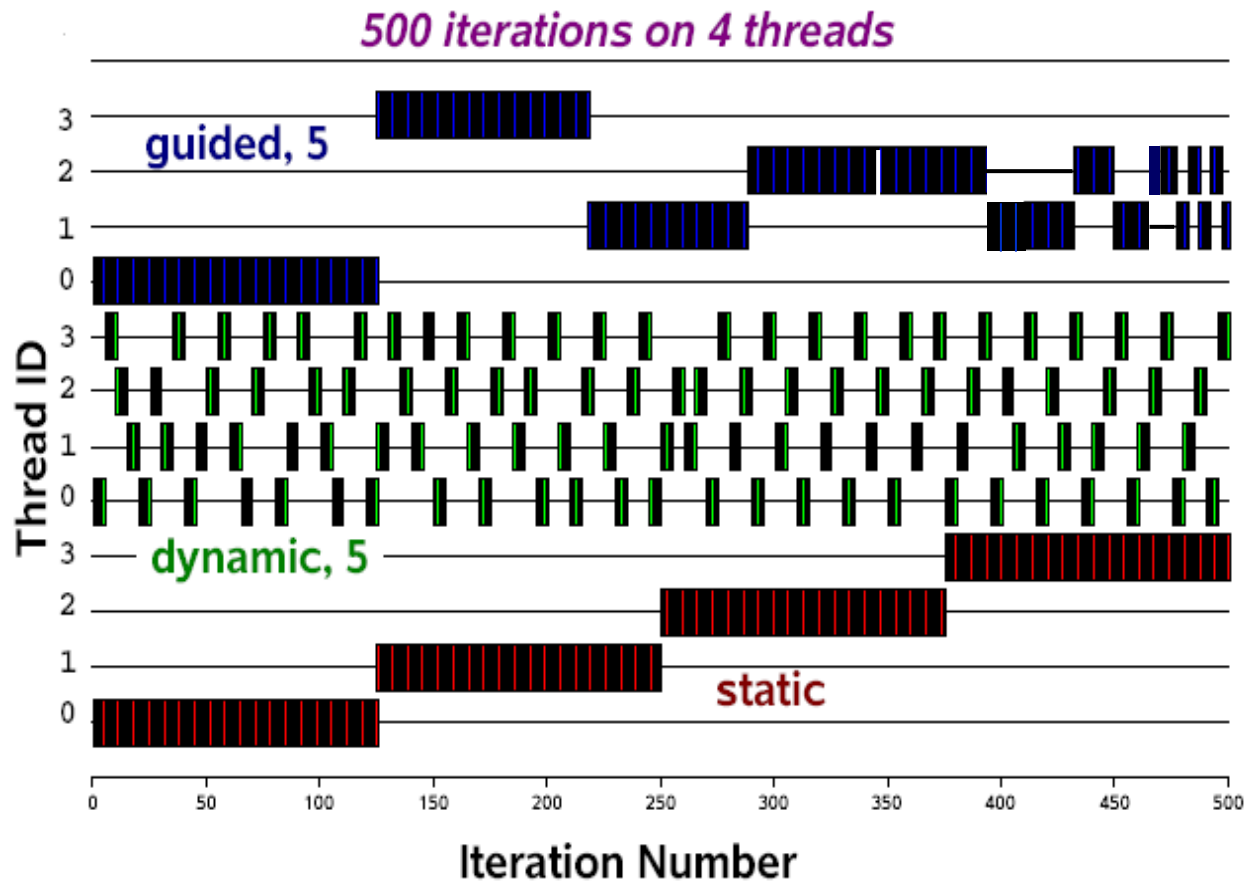
8, ... vlákno přichází pro 3. porci,

6, ... atd.

5,

2.

Celkem iterací na vlákna: 50, 50, 48, 52  
a 11 synchronizovaných přístupů  
k indexu smyčky



- Mějme 35 stejně náročných iterací 0 až 34, 3 vlákna 0 až 2. Najděte všechny dávky iterací přidělené vláknům 0–2.

	<b>vlákno 0</b>	<b>vlákno 1</b>	<b>vlákno 2</b>
static, no chunk	12	11	11
static, chunk = 5	$5 + 5 + 4$	$5 + 5$	$5 + 5$
dynamic, chunk = 7	$7 + 7$	$7 + 6$	7
guided, no chunk	12	$8 + 3$	$6 + 4 + 1$

```
#pragma omp for schedule(runtime)
```

- Způsob plánování (schedule kind) a chunk\_size mohou být zvoleny pomocí proměnné prostředí **OMP\_SCHEDULE** před provedením programu, např. v Unixu příkazem shellu

```
setenv OMP_SCHEDULE "dynamic,3"
```

To je pohodlné pro experimentování, není nutná rekompilace.

```
#pragma omp for schedule(auto)
```

- Plánování je ponecháno na obslužném (runtime) systému. Vhodné, když se obslužný systém může učit z předchozích běhů téže smyčky.

- `schedule(static)` má **nejmenší režii**
  - vhodné, když je práce v iteracích stejná (bez `chunk_size`) nebo se mění lineárně (pak je `chunk_size` nutný).
- `schedule(dynamic, chunk_size)` je vhodná pro **vyvážení zátěže**, když provedení iterací trvá různě dlouhou dobu
  - bez `chunk_size` je sync. **režie příliš velká** (implicitně `chunk_size` = 1)!
- `schedule (guided)` je vhodná pro iterace s ne příliš rozdílnou dobou provedení
  - má nižší počet porcí a tím nižší synchronizační režii než `dynamic`.

- **simd – schedule(simd:static)**

- Velikost chunku se vždy zaokrouhlí na nejbližší vyšší velikost simdlen.

```
int dot;
#pragma omp parallel for simd \
    schedule(simd:static:100) reduction(+: dot) aligned(a, b: 64)
{
    for (int i = 0; i < N; i++)
        dot += (a[i] * b[i]);
}
```

- **monotonic – schedule(monotonic:dynamic)**

- Pokud již vlákno vykonalo iterací  $i$ , musí být následující iterace větší než  $i$  (monotónně rostoucí posloupnost)

- **nonmonotonic – schedule(nonmonotonic:dynamic)**

- Iterace se mohou přidělovat v libovolném pořadí



```
#pragma omp parallel default(none) \  
    shared(n, a, b, c, d) private(i)
```

```
{  
    #pragma omp for nowait  
    for (i = 0; i < n-1; x++)  
    {  
        b[i] = (a[i] + [a[i+1]]) / 2;  
    }
```

Zrušena  
implicitní  
bariera

```
    #pragma omp for nowait  
    for (i = 0; i < n; y++)  
    {  
        d[i] = 1.0 / c[i];  
    }  
}
```

OpenMP

# IMPLICITNÍ CHOVÁNÍ PROMĚNNÝCH

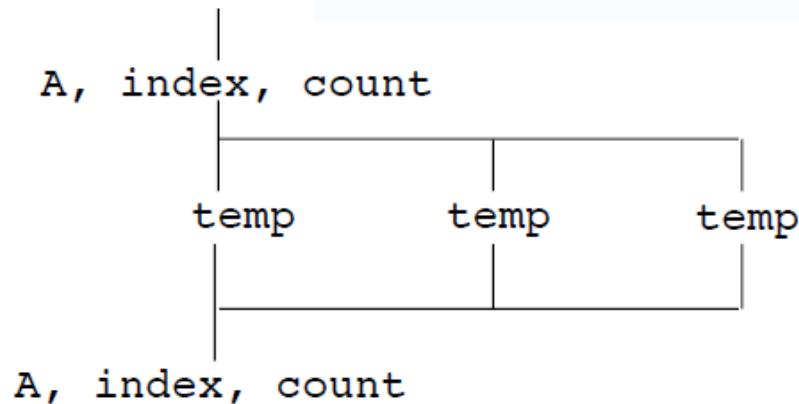
- Proměnné deklarované/použité **před příkazem parallel** jsou v paralelní oblasti implicitně sdílené.
  - např. indexové proměnné sekvenčních smyček musí být v dovětku označeny jako privátní.
  - indexové proměnné paralelních smyček nemusí být na seznamu v dovětku `private( )`, jsou implicitně privátní.
- Automatické proměnné deklarované v bloku příkazů uvnitř paralelní oblasti nebo lokální proměnné ve funkcích volaných z paralelní oblasti jsou privátní (na zásobníku).
  - Proměnné **static** jsou sdílené, a jsou staticky alokovány kompilátorem v předurčené oblasti sdílené paměti.

```
extern double A[10];  
void work(int * index)  
{  
    double temp[10];  
    static int count;  
}
```

**A, index a count**  
jsou **sdílené**  
všemi vlákny.

**temp** je lokální  
(**privátní**) v každém  
vlákně.

```
double A[10];  
int main(){  
    int index[10];  
    #pragma omp parallel  
        work(index);  
    printf("%d\n", index[0]);  
}
```



- **`firstprivate(list)`**... pro inicializaci stejnojmenných privátních proměnných všech vláken ve smyčce jsou použity původní hodnoty proměnné hlavního vlákna 0 (konstrukce C++ copy konstruktorem).
- **`lastprivate(list)`**... hodnota privátní proměnné patřící vláknu, které provedlo ***poslední sekvenční iteraci*** (resp. sekci označenou jako poslední) je přiřazena kopii proměnné vlákna 0.
- proměnná může být případně jak `firstprivate`, tak `lastprivate`.
- když je použit **`default(none)`**, musí být každá proměnná v některém seznamu **`(list)`** – **`shared`**, **`first/last/private`**.
  - Výhodné pro debug kódu.

```
main() {  
  int C, B; int A = 20, n = 100, idx, *data;  
  ... // alokace a načtení vektoru data
```

```
#pragma omp parallel
```

```
{  
  #pragma omp for firstprivate(A) \  
                  lastprivate(B,idx)
```

```
for (int i = 0; i < n; i++) {  
  B = A + i; /* A: je-li jen private, není def. */  
  if (data[i] == 0) idx = i;  
}
```

```
C = B; /* B: je-li jen private, není hodnota B a tedy C def. */  
      /* sdílené C = lastprivate B = 20 + n - 1 */
```

```
} /* konec paralelní oblasti: idx je náhodně nastaveno některými vlákny v některých  
   iteracích. Lastprivate je zde nesmysl, lépe je použít redukci (např. když hledáme  
   nejvyšší index nulového prvku ve vektoru data). */
```

- Globální proměnné, které mají být privátní pro každé vlákno v rámci celého programu, bez ohledu na lexikální rozsah direktivy parallel, lze definovat pomocí:

```
#pragma omp threadprivate (list)
```

- **Threadprivate** proměnné všech vláken mohou být inicializovány na začátku paralelní oblasti hodnotami hlavního vlákna pomocí **copyin(list)** nebo v době jejich definice.

- **Příklad:** vytvoření čítače pro každé vlákno:

```
int counter = 0;
```

```
#pragma omp threadprivate(counter)
```

```
int increment_counter()  ← volají vlákna v paralelní oblasti,  
{                        master i v sekvenční oblasti  
    counter++;           ← vlákno t(id) inkrementuje svůj čítač  
    return (counter);  
}
```

**Pokračování příště**