

7) OpenMP — Paralelizace smyček

- Doposud jsme se věnovali pouze jádru procesoru a co tam udělat pro zvýšení výkonu — zpracování instrukcí mimo pořadí, přeuspořádávání, přednačítání, spekulace, datový paralelismus, ...
- Nyní abstrahujeme od samotného jádra procesoru a budeme uvažovat paralelizaci na vyšší úrovni, tedy na více jádrech procesoru.
- Přednášky 7, 8, 10 jsou jak se to řeší softwarově a 11, 12, 13 jak je to udělané v hardware.

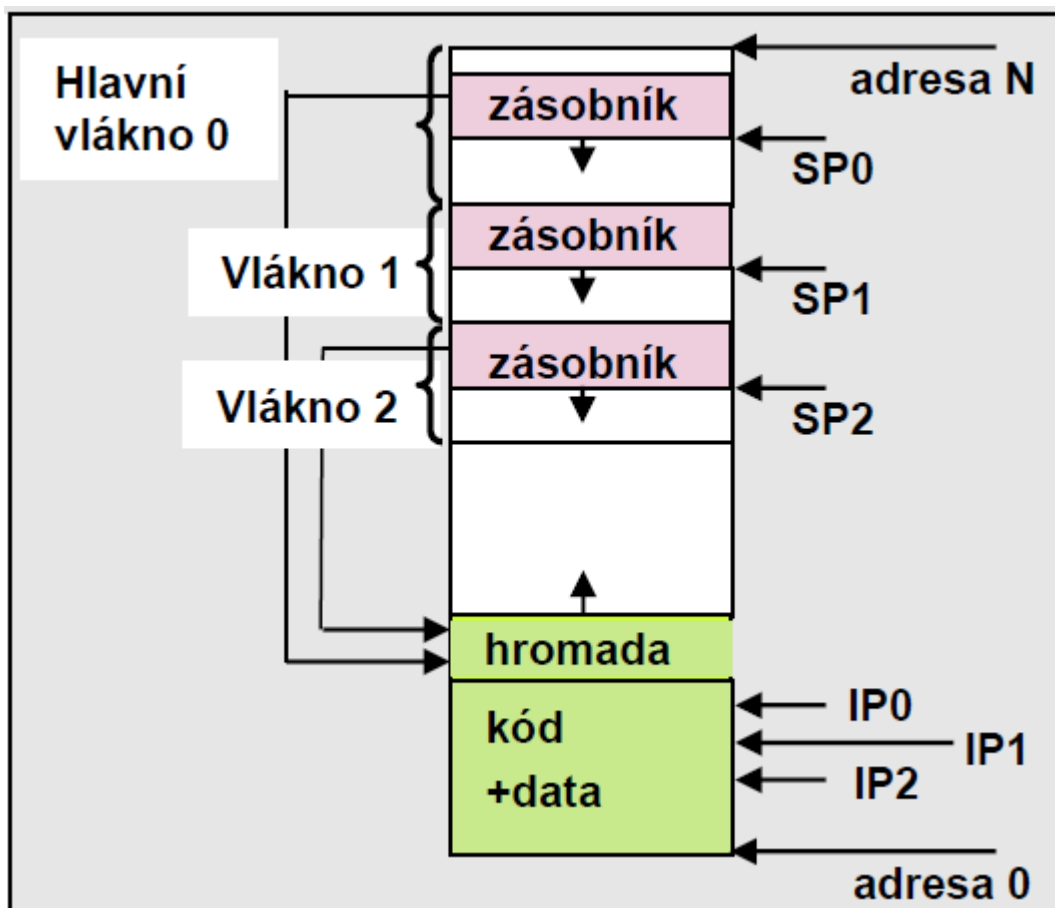
Úvod

Paralelní programovací modely

- SISD — Single Instruction Single Data
 - Skalární procesor
- SIMD — Single Instruction Multiple Data
 - Vektorové procesory
- MISD — Multiple Instruction Single Data
 - Používá se pouze ve velmi speciálních případech
- MIMD — Multiple Instruction Multiple Data
 - Vícevláknové procesory

Sdílený adresný prostor

- Vlákna fungují na systémech se sdílenou pamětí



- Každé vlákno má svůj zásobník, tedy cokoliv naalokované staticky, je privátní pro vlákno
- Halda je společná pro všechna vlákna, tedy cokoliv naalokované dynamicky (malloc, new, ...) je sdílené

Jak paralelizovat

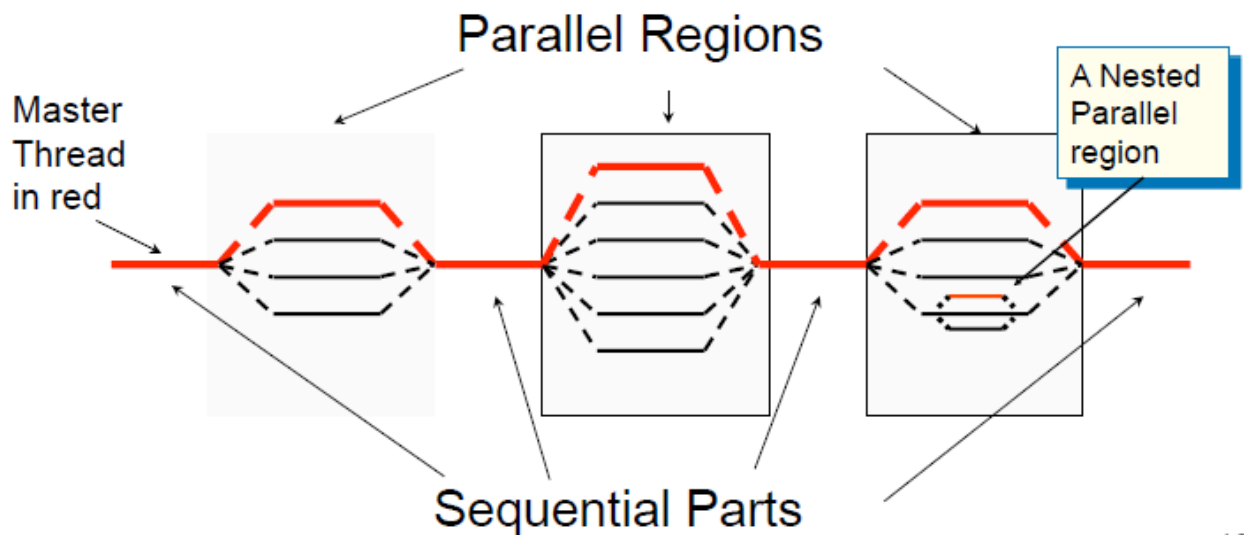
- Sekvenční jazyk + příkazy paralelního zpracování, komunikace a synchronizace
- Sekvenční jazyk + dynamické knihovny
- **Sekvenční jazyk + direktivy pro kompilátor (pragma)**
- OpenMP API — kombinace direktiv a knihovních programů

Výpočetní model vláken

- Vlákna jsou ovládaná a použita kernelem OS
- Hardware vlákno

Model zpracování vláken

- Strukturovaný blok



- Na konci bloku bariéra — sloučení vláken (zabití potomků)

Vytvoření týmu vláken

Direktiva

`pragma omp parallel`

- Vytvoří paralelních sekci — vlákna

```
omp_set_num_threads(4);
double a[1000];

#pragma omp parallel [clause[clause]... ]
{
    int id = omp_get_thread_num();
    work(id, a);
}
```

Dovětky

- Upravují chování direktivy `parallel`

`private(list)`

- Udělat privátní kopii sdílené proměnné
- Deklaruje novou privátní proměnnou, ale neinicializuje!

```
int a = 5;
#pragma omp parallel private(a)
```

```
{
    printf("a = %d\n", a); // vypise smeti
}
```

firstprivate(list)

- Stejný význam jako private, pouze s inicializací na hodnotu, ze které se tvoří kopie.

```
int a = 5;
#pragma omp parallel firstprivate(a)
{
    printf("a = %d\n", a); // vypise 5
}
```

- Privatizovat lze pouze uplné objekty, ne např. pouze nějaké prvky pole
- Privatizovat lze pouze objekty, které mají copy konstruktor nebo primitivní datové typy
 - C++ Vector lze, C struktura lze, ...
 - C array nelze (nevíme kolik má prvků), pokud to uděláme, dostaneme neinicializovaný pointer — segfault

```
float a[10] = {0};
#pragma omp parallel private(a)
{
    a[3] = 123; // segfault
}
```

shared(list)

default(shared | none)

- Výchozí stav, vše nad paralelní sekcí je sdílené, můžu změnit, každé proměnné můžu nastavit jestli má být sdílená nebo privátní

reduction(operator:list)

- Dílčí výsledky do finální hodnoty

if(logical expression)

- Kdy vytvořit paralelní kód

copyin(list)

num_threads(thread_count)

- Kolik má mít sekce vláken

Paralelizace smyček

- Paralelizuje se nejvnější smyčka
- Platí stejné podmínky jako u vektorizace

```
#pragma omp parallel
#pragma omp for
for (int i = 0; i < N; i++) {
    a[i] += b[i];
}
```

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`

Direktiva

pragma omp for

- Iterační paralelismus
- Vlákná si nějakým způsobem rozeberou iterace
- Zaručuje že je to vhodná smyčka (stejně podmínky jako pro vektorizaci)

Dovětky

private(list)

- Stejný význam

firstprivate(list)

- Stejný význam

lastprivate(list)

- Vezme se hodnota, která vznikne v poslední iteraci a uloží se zpět do původní proměnné mimo paralelní sekci

reduction(operator:list)

- Stejný význam
- Operátory:
 - +, -, *, min, max, bitové operace, nebo vlastní redukce

```
double A[MAX];
double ave = 0.0;

#pragma omp parallel for reduction (+: ave)
for (int i = 0; i < MAX; i++) {
    ave += A[i];
}
ave = ave / MAX
```

schedule(kind[, chunk_size])

- Jak rozložit zátěž mezi vlákna
- Kolik iterací dostane jaké vlákno?

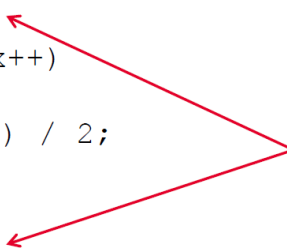
nowait

- Na konci smyčky není jinak implicitní bariéra

```
#pragma omp parallel default node \
    shared(n, a, b, c, d) private (i)
{
    #pragma omp for nowait
    for (int i=0; i < n-1; x++)
    {
        b[i] = (a[i] + [a[i+1]]) / 2;
    }

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

Zrušena
implicitní
bariera



`collapse`

- Vytvoří jednu smyčku ze dvou vnořených a tu paralelizuje

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

Paralelizace smyček se závislostmi

- Mohu zaměnit vnitřní / vnější smyčky tak, aby vnější byla bez závislostí a mohla se 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!

Plánování iterací

- Jak rozdělit iterace mezi vlákna?
- Lze specifikovat pomocí dovětku
 - `schedule({static, dynamic, guided}, [chunk_size])`

src: <http://jakascorner.com/blog/2016/06/omp-for-scheduling.html>

Chunksize

- Po kolika iteracích rozdělují
- Výchozí velikost = 1

Schedule type

Static

- Každému vláknu přidělím stejný počet iterací
- Vhodné pokud jsou iterace stejně náročné
- Nejmenší režie, rozdělení spočítá předem


```

schedule(static):
*****
                *****
                        *****
                                *****

```

```

schedule(static, 4):
****          ****          ****          ****
  ****      ****      ****      ****
    ****    ****    ****    ****
      ****  ****  ****  ****
        ****      ****      ****      ****

```

```

schedule(static, 8):
*****          *****
  *****      *****
    *****    *****
      *****  *****
        *****      *****

```

Dynamic

- Vhodné pokud je každá iterace jinak náročná a já nemám způsob jak to spočítat
- Čím menší chunk size, tím větší režie (synchronizovaný přístup k indexu smyčky — zamykání), ale zase horší vyvážení zátěže — je třeba experimentálně najít něco vhodného

```
schedule(dynamic):
```

```
*  ** * * * * *      * *  ** * * * *      * * *
*      *      * *      * *  *      * *  * * *
*      *      * *      * *  *      * *  * * * *
* *      *      * *      * *  *      * *  * * *
```

```
schedule(dynamic, 1):
```

```
*      *      *      * *      * *      * * * * *
* * *      * *      * * * *      * *      * * * * *
* * *      * *      * *      *      * * * *      * *
*      *      * *      * * *      *      * *      * * *
```

```
schedule(dynamic, 4):
```

```
      ****              ****              ****
****              ****      ****              ****      ****
      ****              ****      ****              ****      ****
      ****              ****              ****              ****
```

```
schedule(dynamic, 8):
```

```
      ****              ****
      ****              ****
****              ****              ****
      ****
```

Guided

- Na počátku větší chunk size a progresivně snižuju
- Tedy nějaké vyvažování zátěže
- Jak?
 - První přidělení chunku:
 - $q_0 = \text{number_of_iterations} / \text{number_of_threads}$
 - Každé další přidělení:
 - $q_i = q_{i-1} * (1 - 1 / \text{number_of_threads})$
- Příklad:
 - Smyčka s 1000 iteracemi
 - Paralelizace na 4 vláknech
 - První chunk: $1000 / 4 = 250$
 - Druhý chunk: $250 * 3 / 4 = 188$
 - Třetí chunk: $188 * 3 / 4 = 141$
 - ...

```
schedule(guided):
```

```
                *****
            *****
*****
                *****
            *****
*****
```

```
schedule(guided, 2):
```

```
            *****
                *****
            *****
*****
                *****
            *****
*****
```

```
schedule(guided, 4):
```

```
                *****
            *****
                *****
*****
                *****
            *****
*****
```

```
schedule(guided, 8):
```

```
            *****
*****
                *****
            *****
                *****
```

Příklady

Příklad 1

I Příklad: Privátní proměnné



```
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?

```
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(); // privatni

    x++; // x = smeti
    y++; // y = 7 v kazdem vlaknu
    z++; // race condition, zde by byla treba kriticka sekce
        // a nejake atomicke pricteni, potom by byl vysledek
        // z = 7, 8, 9, 10, 11 postupne pro dana vlakna

    a[thread_id] = 0; // pristup pres neinicializovany pointer,
                    // segfault
}
```

```

b[thread_id] = 1; // vsechna vlakna zapisuji na svůj index
                  // dvojku, kazde ma svoji privatni kopii
                  // pointeru

c[thread_id] = 2; // vsechna vlakna zapisuji na svůj index
                  // dvojku, jeden sdileny pointer

a += thread_id; // posunuti neinicializovaneho pointeru,
                // segfault

b += thread_id; // kazde vlakno si posune svůj pointer o svoje
                // id

c += thread_id; // race condition, blbost

*a = 5; // pristup pres neinicializovany pointer, segfault
*b = 5; // kazde vlakno zapise 5 na svůj index
*c = 5; // není garantovano kam se to zapise, kvuli race
        // condition
}

```

Příklad 2

! Příklad: matice $b(m \times n)$ krát vektor $c(n \times 1)$ = vektor $a(m \times 1)$! T FIT

```

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

```

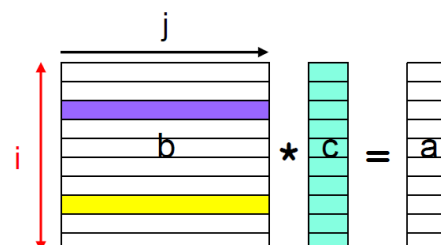
```

{
    #pragma omp parallel for default(none)\
        shared(m,n,a,b,c)
    for (int i=0; i < m; i++)
    {
        double sum = 0.0;
        for (int j=0; j < n; j++)
            sum += b[i][j]*c[j];
        a[i] = sum;
    }
}

```

paralelní smyčka

sekvenční smyčka



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

```

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

```

```

#pragma omp parallel for \
    default(none) \
    shared(m, n, a, b, c)
for (int i = 0; i < m; i++) {
    double sum = 0.0;
    for (int j = 0; j < n; j++) {
        sum += b[i][j] * c[j];
        a[i] = sum;
    }
}
}

```

Příklad 3

I Příklad redukce



```

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)

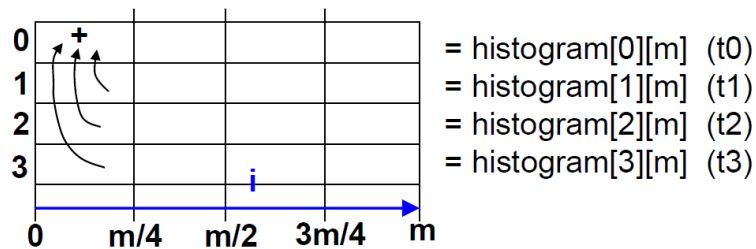
```

Příklad 4

I Příklad výpočtu histogram



- Vektor $a[n]$ má prvky integer v intervalu $\langle 0, m-1 \rangle$. Četnost výskytu různých hodnot se má znázornit histogramem $\text{histogram}[m]$.
 - Nejdřív 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 posčítá jednu část všech P histogramů do části jednoho globálního histogramu $\text{histogram}[0][m]$.



I Příklad: histogram hodnot prvků vektoru $a[n]$



```
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)
    for (i = 0; i < n; i++)          //segmenty vektoru
        histogram[myid][a[i]]+=1;
    #pragma omp for schedule(static)
    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];
    }
}
```

Příklad 5

I Řízené plánování, příklad 4 vlákna = 4 barvy



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

Příklad 6

I Příklad: rozvržení iterací na vlákna



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

	vlákno 0	vlákno 1	vlákno 2
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

Příklad 7

I Příklad sdílených a privátních dat

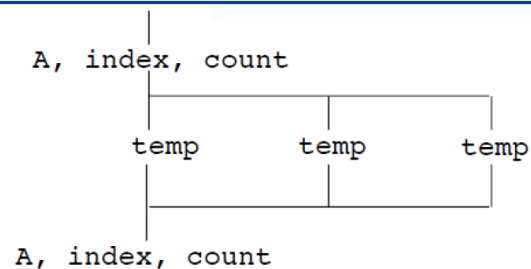


```
extern double A[10];
void work(int * index)
{
    double temp[10];
    static int count;
}
```

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]);
}
```



Příklad 8

I Příklad



```
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). */
```

8) OpenMP — Sekce a tasky

- S paralelníma oblastma je třeba šetřit, respektive jejich režie je poměrně náročná.

Paralelní sekce

- Pro funkční paralelismus
 - Řada funkcí, které je možné vykonávat nezávisle

Direktivy

pragma omp sections

- Do této oblasti přijdou vlákna a začnou si rozebírat jednotlivé sekce
- Mohu používat stejné dovětky jako při for a simd
- Má na konci bariéru (stejně jako parallel)

pragma omp section

- Jednotlivé sekce pro rozebrání
- Nevím pořadí vykonání ani jaké vlákno vykoná co
- Každou sekci jedno vlákno
- Počet sekcí není možné tvořit za běhu programu
- V OpenMP neexistuje žádný způsob jak komunikovat mezi sekcemi, musel bych napsat vlastní komunikaci přes signálování

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

- Mám 3 nezávislé množiny funkcí, vytvořím 3 sekce

pragma omp single

- Blok příkazů, která se vykoná pouze jednou a to prvním vláknem, které k němu dojde
- Na konci bloku je bariéra
- Dovětek copyprivate
 - Vláknem single zpřístupní svoje privátní proměnné ostatním vláknům

```
#pragma omp single
```

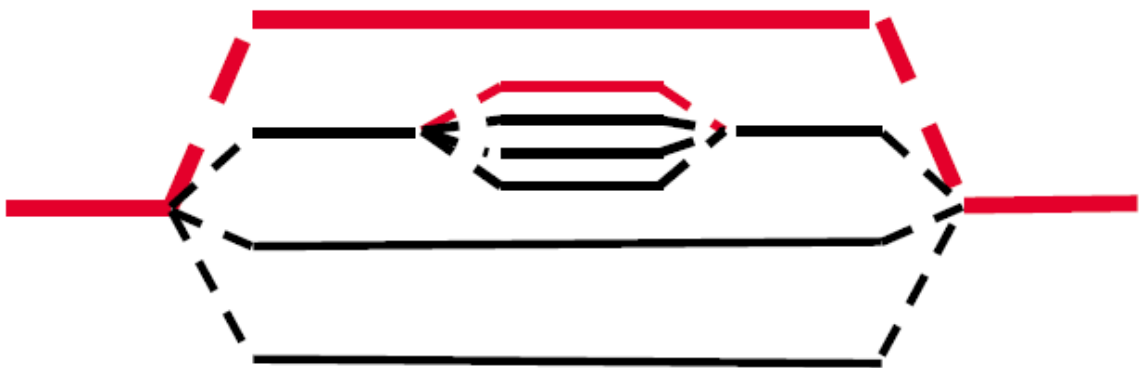
pragma omp master

- Blok příkazů, která se vykoná hlavním vláknem
- Na konci bloku není bariéra

```
#pragma omp master
```

Vnořený paralelismus

- Uvnitř sekce nový pragma omp parallel



Pravidla používání direktiv sdílení práce (for, sections, single)

- 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.

Tasky

- Blok příkazů, který je přidám do fronty, odkud si jej berou jednotlivá vlákna, co zrovna nemají co na práci.
- Jakmile nějaké vlákno narazí na task, přidá ho do fronty a jede dál.
- Oproti sekcím, můžeme generovat za chodu kolik a jak chceme!

Direktivy

pragma omp task

- Změna oproti zbytku OpenMP, všechny proměnné nad taskem nejsou public, ale firstprivate!
- Generování tasků probíhá vždy když někdo narazí na pragma omp task, tedy je třeba použít single (to má bariéru a čeká se na vygenerování tasků) nebo master

```
#pragma omp parallel
{
    #pragma omp master // Thread 0 přida tasky do fronty
    {
        #pragma omp task
        {
            fred();
        }
        #pragma omp task
        {
            daisy();
        }
        #pragma omp task
        {
            billy();
        }
    }
} // Zde všechna vlákna čekají a kontrolují frontu tasků,
// všechny tasky jsou před touto bariérou
```

pragma omp taskwait

- Všechny tasky definované v aktuálním tasku, musí být dokončeny, než pokračuju.
- Pro rekurzi

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            fred();
        }
        #pragma omp task
        {
            daisy();
        }
        #pragma omp taskwait
        #pragma omp task // fred() a daisy() musí byt
        // dokončeny drive, než se billy vubec prida do fronty
        {
            billy();
        }
    }
}
```

pragma omp taskgroup

- Na konci skupiny jsou všechny tasky dokončeny

pragma omp cancel

- Konec výpočtu, pro vyhledávání

I Příklad na OpenMP cancel – Je v matici nulový prvek?



```
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 přerušíme výpočet

Zde se ostatní vlákna dozví, že mají ukončit výpočet

Probírali se ještě další věci, ale spíš takové detaily z OpenMP, hádám nebude na zkoušce.

Příklady

Příklad 1)

I Příklad na tasky: Průchod vázaným seznamem



```
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
```

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

Všechny tasky dokončí zde

Příklad 2)

I Fibonacciva čísla sekvenčně - rekurzivně



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 $\text{fib}(n)$ sekvenčně a režii tasků v OpenMP vynechat:

`if (n ≤ 30) return seqfib(n);`

I Fibonacciva čísla paralelně s tasky



```
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);

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

pozastav rodič.
task, až dokončí
dceřiné tasky

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

```
int fib_seq(int n)
{
    if (n < 2) {
        return n;
    }

    int a = fib_seq(n - 1);
    int b = fib_seq(n - 2);
}
```

```

    return a + b;
}

int fibb(int n)
{
    if (n < 30) { // pro snadne vypocty nema smysl tvorit
                  // tasky, rezie by prevazovala
        return fib_seq(n);
    }
    else {

        int a, b;

        #pragma omp task shared(a)
        {
            a = fibb(n - 1);
        }

        #pragma omp task shared(b)
        {
            b = fibb(n - 2);
        }

        // zde je task pozastaven a vracen do fronty,
        // vlakno je volne a muze pracovat na necem jinem
        #pragma omp taskwait

        return a + b;
    }
}

int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            fibb(100);
        }
    } // zde je bariera a vlakna si rozebiraji tasky
}

```



```
return 0;  
}
```

10) OpenMP — synchronizace

Úvod

- Synchronizace slouží
 - K ochraně přístupů ke sdíleným datům
 - K čekání na nějakou událost (producent — konzument)
 - K vynucení pořadí akcí
 - ...

Direktivy

Vysoká úroveň

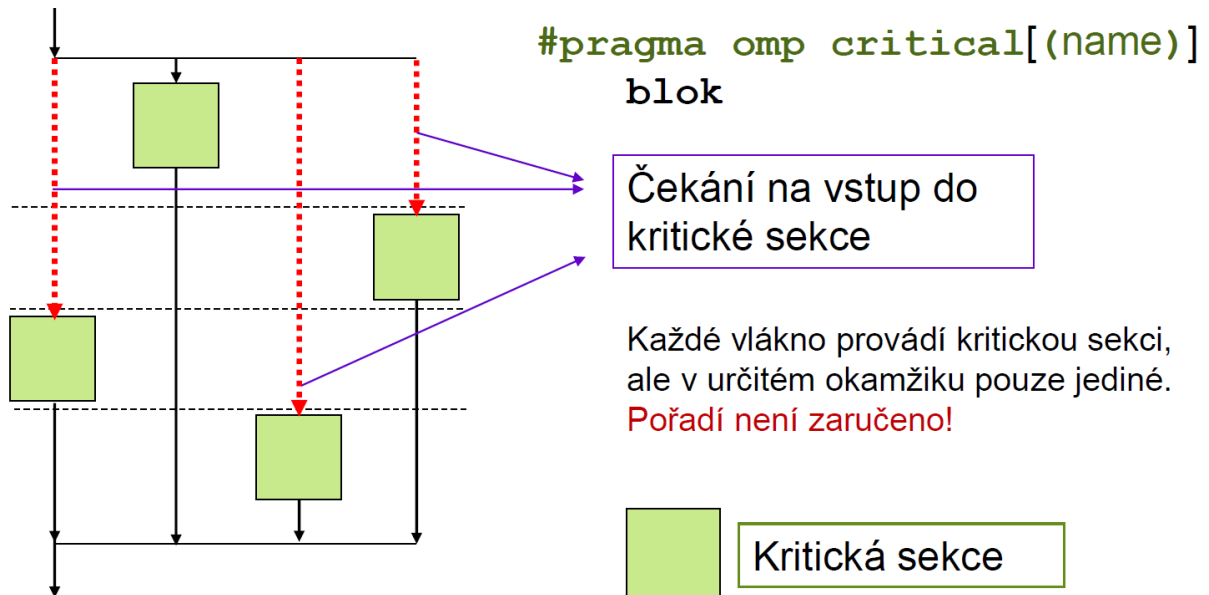
- critical
- atomic
- barrier
- master
- ordered

Nízká úroveň

- Jsou součástí direktiv vyšší úrovně
- flush
- locks

Vzájemné vyloučení

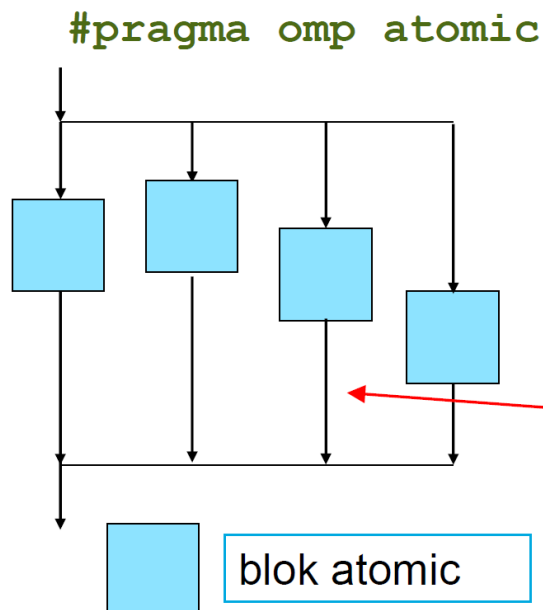
Kritická sekce



```
// hleda se index prvního nulového prvku
int first = n;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    if (a[i] == 0 && i < first) {
        #pragma omp critical
        // podmínka znovu, protože někdo při mém čekání na
kritickou
        // sekci mohl zapsat
        if (i < first) {
            first = i;
        }
    }
}
```

- Kritická sekce je jedna a pouze globální! Pokud chci více kritických sekcí, je nutné použít identifikátory.
- Poměrně velká režie, když to jde, používat atomic.

Atomická modifikace



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

- Chrání aktualizaci jednoho paměťového místa
- Pouze pro
 - Triviální datové typy, které načtu jednou instrukcí! (ne struktury)
 - int, float, double, char, ...
 - Triviální operace
 - +, *, /, bitwise AND, XOR, OR (&, ^, |), <<, >>

Dovětky

- read
- write
- update
 - Vychodi stav
- capture
 - Ctu hodnotu a zapisuju novou
- seq_cst
 - Vnucení konzistence, všechno co je předtím, se musí dokončit, žádné předbíhání

```
// ctu hodnotu a zapisuju novou
#pragma omp atomic capture
{
    old_value = *p;
    (*p)++;
}
```

```
}
```

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

```
// v podstate implementace redukce se scitanim
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;
}
```

Synchronizace událostmi

- Jde o to, zařídit pořadí nějakých událostí

Direktivy

Bariéra

- Výchozí u každé direktivy parallel, for, section, taskgroup
- Čeka se až dojdou všechna vlákna, poté se pokračuje
- Bariéry nejsou dovoleny v dynamickém rozsahu — nedává smysl
 - for, ordered, sections, single, master

```
#pragma omp barrier
```

Ordered

- Oblast, která se provádí v sekvenčním pořadí
- Například pro logování a debugování, jinak moc nedává smysl

```
void work(int k)
{
    #pragma omp ordered
    printf("%d ", k); // zde budou vlákna cekat
}

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

Master

- Označuje blok kódu v rámci paralelní oblasti, který je vykonáván hlavním vláknem, ostatní vlákna přeskochí
- Neobsahuje implicitní bariéru na konci
- Použití:
 - Pro čtení parametrů a generování tasků

```
#pragma omp master
```

Synchronizace programovaná uživatelem

Direktivy

Flush

- Zábrana, která specifikuje, kde se v kódu povoluje předbíhání
- Nad flushem nic nesmí předbíhat to pod ním
- Vnutí sekvenční konzistenci
- Volitelný dovětek seznam proměnných, který se musí "spláchnout"

Locks

- ...

```
// tato funkce je volala paralelne ruznymi vlakny
// implementace bariery pro N vlaken
void bariera(int N)
{
    static int counter = 0;

    #pragma omp flush

    #pragma omp atomic
    counter++;

    #pragma omp flush

    while (counter != N) {
        usleep(1);
        #pragma omp flush
    }

    #pragma omp master
    counter = 0;
}
```