MI-SPOL-12

Programový model nad sdílenou pamětí: OpenMP (paralelní regiony, paralelní vlákna, datový a funkční paralelismus, paměťový model, synchronizační nástroje).

Typy paralelních konstrukcí

- **Instruction Level Paralellism (ILP):** několik instrukcí lze vykonat souběžně, pokud jsou datově nezávislé
  - Datové závislosti:
    - True data dependency: instrukce čte registr, do kterého předchozí musí nejdřív zapsat
    - Output dependency: obě instrukce zapisují do stejného registru
    - Anti-dependency: jedna instrukce musí registr přečíst dříve, než jí ho jiná přepíše
- **Data paralelismus:** Prvky velké datové struktury rovnoměrně rozděleny mezi procesory, z nichž každý provádí synchronně výpočet nad přidělenou částí
- **Iterační paralelismus:** Datově nezávislé iterace cyklu prováděny současně samostatnými procesory
- **Funkční paralelismus:** Program rozdělen do nezávislých kusů kódu (strukturované bloky, funkce, rekurzivní volání...), které mohou být prováděny paralelně

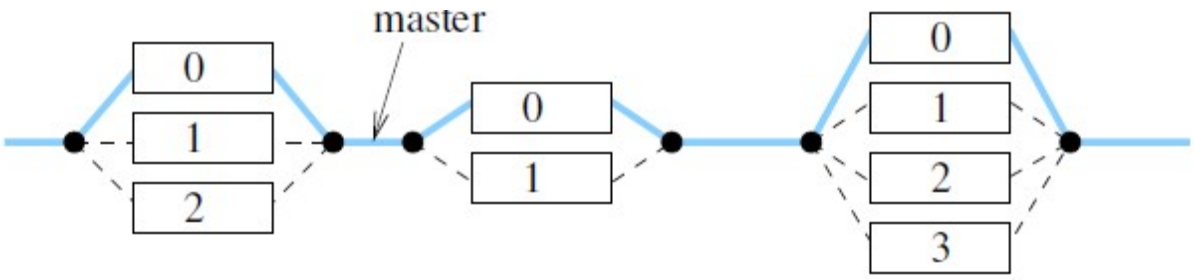
OpenMP: vysokoúronové API pro programování vícevláknových aplikací
SMP UMA nebo NUMA
Skládá se z *parametrizovatelných direktiv*, *systémových proměnných* a *knihovny nejčastějších operací běhového prostředí*

Vlákno: nejmenší blok kódu, který lze plánovat a provádět na úrovni operačního systému a který sdílí kontext výpočtu procesu

Explicitní model paralelního výpočtu: programátor má plnou kontrolu a zodopvědnost za paralelní výpočet

Paralelní regiony: vybrané části původně sekvenčního kódu.
Pomocí `fork-join` mechanismu jsou v nich vytvářena, prováděna a ukončována vlákna.

Mimo ně existuje pouze **master** vlákno (hlavní vlákno procesu)



OpenMP může využívat **zásobárnu recyklovatelných vláken** -- zmenšení režie vytváření a ukončování vláken

Motivace zavedení OpenMP

- přenositelnost
- standardizace
- jednoduchost

Nevýhody:

- není určeno pro distribuovanou paměť
- není zaručeno nejefektivnější využití sdílené paměti
- chybí automatická kontrola datových závislostí
- chybí synchronizace výpočtu s I/O

Obvyklý postup vícevláknové paralelizace

- Tvorba paralelních regionů pomocí OpenMP direktiv
- Zakázáno skákat z regionu ven nebo zvenčí dovnitř

```
#include<omp.h>
int main(){
    int x1, x2, s3;
#pragma omp parallel private(x1,x2) shared(s3) num_threads(5){
    // ...
    }
}
```

- Kompilace pomocí `-fopenmp` : správný OpenMP kód se s překladem bez přepínače zkompiluje do

korektního sekvenčního

Paměťový model

- **Volnejší konzistence** sdílené paměti: vlákna mohou udržovat lokální kopie proměnných, nejsou povinna každou změnu ihned propisovat do sdílené paměti -- zodpovědnost programátora (např. `flush()`)

Direktivy

Obecně:

```
#pragma omp direktiva klauzule1 klauzule2
{
    ...
}
```

Direktiva `parallel`

- Vytváření **paralelních regionů**
- **Klauzule:**
  - `if(podminka)` : **podmínka paralelizace** regionu
  - `num_threads(vyraz)` : **počet vláken** v paralelním regionu (vč. master vlákna)
  - `vlastnosti(seznam_promennych)` : vlastnosti proměnných v paralelním regionu
    - `shared` : skalární proměnná (ne pole ani struktura) **sdílena** všemi vlákny
    - `private` : proměnná je **lokální** v každém vláknu
    - `firstprivate` : proměnná **lokální** v každém vláknu, každé vlákno ji má **inicializovanou** na hodnotu, kterou měla před vstupem do regionu
    - `lastprivate` : proměnná **lokální**, hodnota sekvenčně poslední iterace paralelního cyklu překopírována do proměnné hlavního vlákna procesu
    - `default` : která z vlastností bude **výchozí**
    - `reduction(operator:variable)` : sdílená proměnná **lokálně nakopírována** do každého vlákna, po skončení regionu všechny instance **zredukovány** poomocí operátoru, výsledek zapsán do původní sdílené proměnné (operátor nesmí být přetížen, nemusí být kvůli zaokrouhlovacím chybám asociativní, jedna proměnná max v jedné redukci)
    - `threadprivate` : **globální platnost** `private` , proměnné si mezi paralelními regiony drží

své hodnoty (nejsou rušeny)

- Na konci regionu implicitní bariéra -- po jejím překonání nová vlákna ukončena a dál pokračuje jen master
- Pokud 1 vlákno ukončeno předčasně, končí celý program
- Vlákna číslována od 0 (hlavní) do $p-1$
  - Počet určen takto: vyhodnocení `if(podminka)` (nesplněno $\Rightarrow$ 1 vlákno), hodnota `num_threads(vyraz)`, hodnota posledního volání `omp_set_num_threads(vyraz)`, poslední hodnota systémové proměnné `OMP_NUM_THREADS`

Direktiva `for`

- OpenMP primárně zaměřeno na **datový paralelismus**
- `for` -- iterační paralelismus datově nezávislých iterací cyklu
- Na konci implicitně bariéra
- **Klauzule:**
  - `schedule(typ, chunk-size)` : způsob přidělení iterací vláknům
    - `static` : **staticky** cyklicky přiděleny bloky po sobě jdoucích iterací o velikosti `chunk-size` (pokud uvedeno, jinak $\frac{n}{p}$)
    - `dynamic` : **dynamicky** přidělovány bloky o velikosti `chunk-size` (defaultně 1) *(vlákno dostance další iteraci až když dokončí aktuální)*
    - `guided` : **dynamicky** přidělovány bloky x iterací, kde $x = \max(\lceil \#\text{dosud nepřidělených iterací}/p \rceil, \text{chunk-size})$. Chunk size defaultně 1
    - `runtime` : zvoleno až **v okamžimu spuštění** dle systémové proměnné `OMP_SCHEDULE`
    - `auto` : necháno na kompilátoru a OS
  - `collapse` : paralelizace vnořených cyklů (implicitně je `for` pouze na cyklus nejvyšší úrovně)
  - `ordered` : pořadí iterací stejné jako při sekvenčním provádění
  - `nowait` po dokončení není provedena bariéra

Direktiva `task`

- složitější **funkční paralelismus** s větší režií
- vhodné i pro **rekurzivní volání**
- přidělování výpočtu vláknům je **producent-konzument**
- **Úloha:** jednotka paralelního kódu obsahující:
  - ukazatel na začátek kódu, který se má provést
  - vstupní data
  - datovou strukturu, do které vloží svou identitu vlákno, jakmile úlohu začne provádět jako konzument
- Provedení direktivy `#pragma omp task` způsobí, že vlákno vykonávající tento kód:

    - vygeneruje novou úlohu a vloží ji do zásobárny úloh
    - V poolu úloha čeká, než si ji některé volné vlákno (konzument) vyzvedne a začne se provádět
- Klauzule `if(podminka)` -- efektivní řízení paralelizace
    - podmínka splněna: nová úloha vložena do poolu
    - podmínka nesplněna: aktuální vlákno odloží dočasně ordičovskou úlohu do poolu, začne sekvenčne vykonávat novou úlohu a po jejím dokončení si zpět vyzvedne rodiče a dokončí ji
- `# pragma omp taskwait` : úloha čeká na dokončení všech jejích přímých potomků

Direktiva `cancel`

- Bezpečný způsob předčasného ukončení paralelního regionu
- Provedením `cancel` vydá vlákno signál ostatním a přejde na závěrečnou bariéru
- Vlákna, která ke `cancel` dojdou až po tom jdou taky na bariéru
- Vlákna, která už byla za `cancel` dokončí standardně svůj kód paralelního regionu
- `# pragma omp cancel construct [if(expr)]` , kde `construct` může být `parallel, for, taskgroup, sections`

OpenMP operace

- `get_num_procs()` : počet CPU jader k dispozici pro OpenMP
- `omp_get_thread_num()` : index aktuálního vlákna
- `omp_get_num_threads()` : počet vláken v aktuálním paralelním regionu
- `omp_set_num_threads(int i)` : změní počet vláken v následujících paralelních regionech
- `omp_get_wtime()` : uběhnutý čas od nějakého okamžiku v minulosti (používáno párově pro měření času)

Synchronizační nástroje

- `barrier` : místo, kam paralelní vlákna daného paralelního regionu musí dorazit a **čekat na ostatní**
- `master` : daný blok kódu smí provést pouze **hlavní vlákno**
- `single` : daný blok smí provést pouze **jedno vlákno**
- `critical` : vytvoření kritické sekce pro zajištění **výlučného přístupu**
    - **Kritická sekce:** jedna nebo několik částí kódu paralelního regionu, které lze provádět v jednom okamžiku pouze jedním vláknem

    - Kritická sekce je defaultně **anonymní** a pokud jich je v kódu víc, vzájemné vyloučení platí **globálně** pro všechny
    - `# pragma omp critical name` vytvoří **pojmenovanou**
- `atomic` : daná paměťová operace nad paměťovou buňkou bude provedena atomicky -- **jednovláknově a nepřerušeně**
    - Operace typicky typu R-M-W (read-modify-write)
    - Lze aplikovat na operace `read` , `write` (na některých architekturách neatomické), `update` (`i += 1`) a `capture` (`my_i = i; i += 2`)
- `flush()` : propsání aktuálních hodnot daných sdílených proměnných do sdílené paměti
- `taskwait` : synchronizace synovských úloh s rodičovskou