6. Paralelní zpracování v OpenMP: Smyčky, sekce, tasky a synchronizační prostředky.

OpenMP je knihovna zjednodušující paralelní programování v jazycích C, C++ a Fortran. V C/C++ se využívá pomocí pragma direktiv preprocesoru a knihovních funkcí. Knihovna poskytuje možnost jak vektorizace výpočtu pomocí vektorových instrukcí, tak paralelizace výpočtu na více vláknech.

Vektorizace

Nejprve krátce zopakujeme způsob, jak se pomocí OpenMP vektorizuje (viz otázka 3). K vektorizaci smyčky se využívá direktiva #pragma omp simd, případně s dalšími volitelnými parametry. Pokud chceme nějakou naši funkci v programu vyznačit jako vektorizovatelnou, přidáme nad funkci #pragma omp declare simd. Použitím těchto direktiv kompilátoru říkáme, že vektorizace je možná a že jako programátor bereme zodpovědnost za dodržení pravidel vektorizace (např. že nejsou datové závislosti mezi jednotlivými iteracemi). Součet několika vektorů tedy můžeme realizovat například takto:

```
void add(float* a, float* b, float* c,
         float* d, float* e, int n)
{
  #pragma omp simd
  for (int i=0; i<n; i++)
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Pokud bychom měli například nějakou proměnnou, do které chtějí přičítat všechny iterace, tzn. je zde datová závislost, můžeme použít přívětek reduction, který umožní akumulaci hodnoty během smyčky. Podmínkou zde je, že pořadí operací musí být možné přeházet, což vzhledem k vlastnostem sčítání platí.

```
#pragma omp simd reduction(+:sum)
for(i = 0; i < *p; i++)
{
  A[i] = B[i] * C[i];
  sum = sum + A[i];
}
```

Kromě reduction je možné použít ještě následující dovětky:

- safelen(délka) – specifikuje maximální počet iterací, co lze vykonávat současně

- simdlen(délka) – specifikuje preferovanou délku vektorového registru
- linear(proměnná:krok) – specifikuje, že proměnná má hodnotu lineární vzhledem k číslu iterace
- aligned(proměnná:zarovnání) – specifikuje bytové zarovnání proměnné
- collapse(n) – umožňuje vektorizovat vnořené smyčky (typicky vektorizujeme pouze nejvnitřnější smyčku)

U SIMD-enabled funkcí (deklarovaných pomocí #pragma omp declare simd) můžeme ještě kromě výše zmíněných dovětků (simdlen, linear, aligned, …) využít:

- inbranch – funkce je vždy volána zevnitř if-podmínky, compilátor přidá do parametrů fce masku
- notinbranch – funkce nikdy není volána zevnitř if-podmínky

Přestože podmínky jsou většinou těžko vektorizovatelné, tyto dovětky to částečně umožňují.

Paralelní programování na více vláknech

Máme 3 základní způsoby paralelizace:

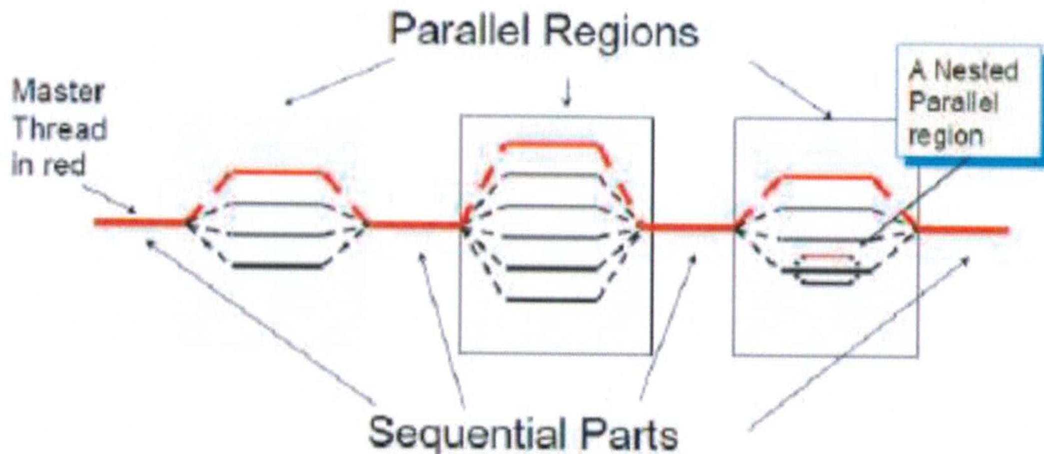
- využití sdíleného adresového prostoru (MIMD, SPMD)
- zasílání zpráv (MIMD, SPMD)
- datově paralelní model (SIMD) – viz vektorizace

OpenMP zjednodušuje programování s využitím sdíleného adresového prostoru. Vzhledem k tomu, že OpenMP je nezávislé na stroji a operačním systému, jsou vlákna v OpenMP abstraktní, konkrétní mapování na systémové prostředky pak záleží na cílové architektuře (např. pthread na Linuxových systémech). Každé vlákno má svůj instruction pointer, stack pointer, registry a privátní zásobník. Vlákna jednoho procesu mohou běžet na různých jádrech, mohou tedy běžet současně – operace v nich musí být thread-safe. Zároveň je možné vláknový paralelismus kombinovat s vektorizací – typicky vektorizujeme nejvnitřnější smyčku a paralelizujeme vnější smyčku.

Jak bylo naznačeno, s OpenMP je možné pracovat třemi základními způsoby:

- direktivy pro překladač (#pragma omp …)
- funkce z knihovny (hlavičkový soubor omp.h) – např. omp_get_num_threads(), omp_get_thread_num(), …
- proměnné prostředí (např. OMP_NUM_THREADS, …)

Program v OpenMP se začne vykonávat hlavním vláknem (master thread). Jakmile dojde k paralelnímu příkazu, vytvoří se tým vlákem (worker threads) – vlákna začnou vykonávat příkazy paralelně až dokud nenarazí na synchronizační prostředek, např. bariéru. Obecně je možné paralelizovat jakýkoliv blok, který má jeden vstup nahoře a jeden výstup dole. Program se může při běhu opakovaně větvit a zase spojovat (tzv. model fork-join), schéma běhu tedy může vypadat třeba takto:



Rozvětvení na tým vláken se v OpenMP provede využitím direktivy #pragma omp parallel – blok následující za touto direktivou provádí všechna vlákna, přičemž každé vlákno má své vlastní ID (master má ID 0), na které je možné se dotázat pomocí omp_get_thread_num(). Na konci bloku je implicitně bariéra, tzn. vlákna na sebe čekají a dojde ke spojení:

nastavení počtu vláken

Dovětky

```
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  }
```

Strukturovaný blok vykonávaný 4 vlákny

Možnými dovětky jsou:

- private(seznam proměnných) – seznam proměnných privátní každému vláknu (vytvoří novou lokální kopii každému vláknu). Proměnná není inicializována (v C++ se využije defaultní konstruktor) a její hodnota se nepřenáší žádným způsobem ven. Proměnné definované uvnitř paralelní oblasti jsou by-default privátní.
- firstprivate(seznam proměnných) – stejně jako private, ale proměnná v každém vláknu se inicializuje hodnotou, která původně byla sdílená (provede se copy konstruktor)
- lastprivate(seznam proměnných) – přenese poslední hodnotu privátní proměnné do sdílené proměnné
- shared(seznam proměnných) – seznam proměnných sdílených vlákny. Proměnné deklarované před příkazem parallel jsou by-default sdílené.
- default(none|shared) – nastavuje výchozí mód proměnných. default(shared) je jako kdybychom každou proměnnou uvedli v direktivě shared(). Naopak u default(none) je nutné explicitně specifikovat viditelnost každé proměnné.
- reduction – viz vektorizace, umožňuje přenést hodnotu privátních proměnných do oblasti za paralelní oblastí

- if(logický výraz) – rozvětvení je možné podmínit
- num_threads(n) – počet vláken

Paralelizace smyček

Direktiva parallel sama o sobě představuje program SPMD, tzn. každé vlákno provádí redundantně stejný kód, neprobíhá žádná spolupráce. Ke sdílení práce je nutné využít další prostředky – např. můžeme mezi vlákna rozdělit jednotlivé iterace smyčky pomoci direktivy #pragma omp for., např.:

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

Je nutné znát počet iterací již na vstupu do smyčky, tzn. nehodí se pro paralelizaci while cyklů, hlavní využití najde u cyklů, které je možné přepsat na for. Direktivě můžeme přidat další volitelná nastavení:

- private, firstprivate, lastprivate (stejně jako u #pragma omp parallel)
- nowait – odstraní implicitní bariéru na konci smyčky
- reduction
- schedule – umožňuje nastavit, jak je práce dělena (viz dále)

Pokud paralelní oblast obsahuje jen #pragma omp for (viz příklad výše), je možné zápis zkrátit jako #pragma omp parallel for. Máme 4 základní způsoby plánování iterací:

- schedule(static[, chunk_size]) – naplánování provedeno již během kompilace. Pokud není udán chunk_size, dostane každé vlákno cca stejně iterací. Jinak jsou iterace vláknům přiděleny cyklicky, každé dostane chunk_size (tzv. prokládané plánování) – užitečné, když se práce v iteracích mění lineárně vzhledem k iterační proměnné.
- schedule(dynamic[, chunk_size]) – iterace jsou přidělovány po blocích o velikosti chunk_size, synchronizuje se při každém přidělení – největší režie, ale dosahuje nejrovnoměrnějšího rozložení práce mezi vlákny, pokud se práce v jednotlivých iteracích liší.
- schedule(guided[, chunk_size]) – také částečně dynamické, ale přidělovaná porce se v průběhu výpočtu mění. Má menší synchronizační režii než dynamic.
- schedule(auto) – Plánování je ponecháno na runtime systému.

Příklad na viditelnost proměnných:

```
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). */
```

Sekce

Sekce v OpenMP jsou úseky kódu, které mohou běžet paralelně. Každá sekce v bloku sekcí je vykonávána jen jednou nějakým vláknem v týmu. Počet sekcí je pevně daný kódem – nejde měnit za běhu:

```
#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
```

V tomto případě si jedno vlákno vezme 1. sekci, druhé vlákno 2. sekci a třetí vlákno 3. sekci. Volitelné direktivy fungují stejně jak bylo popsáno výše u parallel/for.

Pokud chceme, aby nějaký příkaz/blok vykonalo pouze jedno, resp. pouze master vlákno, můžeme nad ním použít direktivu #pragma omp single, resp. #pragma omp master (jde využít například pro omezení I/O operací jen na jedno vlákno). Pozor: #pragma omp master nemá implicitní bariéru na konci narozdíl od #pragma omp single.

Tasky

Trochu odlišným paralelním mechanismem jsou tzv. tasky (úlohy). Těch narozdíl od sekcí může být dynamický počet. Task má nějaký kód, co musí provést, data a přiřazené vlakno. Typickou strukturou pak může být, že jedno vlákno generuje tasky, ty jsou pak prováděny vlákny týmu v libovolném pořadí nezávisle na sobě. Díky taskům můžeme paralelizovat while smyčky nebo rekurzivní výpočty. K vytvoření tasku se používá #pragma omp task, přičemž opět může být nutné specifikovat viditelnost proměnných pomocí shared/private/firstprivate. Proměnné se řídí následujícími pravidly:

- 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 proměnou.
- 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 =      firstprivate
    c =      shared
    d =      firstprivate
    e =      private
  }
}
```

Příklady:

```
#pragma omp parallel        <---- Zde se vytvoří vlákna.
{
  #pragma omp master        <---- Pouze master vlákno projde tuto část a vyrobí 3 tasky.
  {
    #pragma omp task
    fred();
                            3 nezávislé úlohy.
    #pragma omp task
    daisy();

    #pragma omp task        Ostatní vlákna nečekají a hned jdou zpracovávat.
    billy();
  }
}                           Zde je frotna čekajících tasků. V tomto bodě je garantováno dokončení všech tasků
```

Průchod 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 - implicitni bariéra
```


Pokud potřebujeme počkat na dokončení tasku před pokračováním v současném tasku, je možné využít #pragma omp taskwait, např. při paralelním výpočtu fibonacciho čísla musíme počkat než se dopočítají rekurzivní hodnoty:

```
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;
}
```


Dále je také možné specifikovat vzájemnou závislost tasků pomocí sdílených proměnných a klauzule depend. Pokud použijeme depend(in: var) task se nespustí dokud se nevykonají předchozí tasky, které se odkazují na proměnnou var v klauzuli depend(out: var).

Task/paralelní zpracování je možné v novějších verzích OpenMP přerušit pomocí #pragma omp cancel. Dané vlákno/úloha pokračuje ke konci přerušeného regionu. Ostatní vlákna pak v místě

direktivy #pragma omp cancellation point kontrolují, zda nebyl výpočet přerušen, a pokud byl, tak také skončí.

Synchronizační prostředky

Při paralelním programování musíme využít synchronizaci k:

- ochraně přístupu ke sdíleným datům
- čekání na nějakou událost
- vynucení pořadí akcí

Problematický z pohledu sdílení dat je čtení a zápis několika paralelních vláken do sdílené proměnné (data race). V OpenMP můžeme vytvořit kritickou sekci pomocí #pragma omp critical [name] – pouze jedno vlákno do ní může v jeden čas vstoupit. Podobně je možné použít #pragma omp atomic, což říká, že daný blok je prováděn nerozdělitelně. Pokud by bylo zapisováno do stejné proměnné, OpenMP se postará o serializaci podobně jako v případě kritické sekce. Rozdílem mezi atomic a critical je, že v atomické sekci může pracovat více vláken současně, pokud pracují nad různými daty (v kritické sekci to není možné). Atomic je implementován atomickými instrukcemi, zatímco critical pomocí zámků – atomic má typicky nižší režii. Příklad použití při hledání prvního nulového prvku vektoru:

```
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;
  }
}
```

Dvě kontroly na i < first: chceme vstupovat do kritické sekce jen když je to nutné (nechceme zamykat na každém prvku), ale zároveň musíme zajistit, že v mezičase nenalezlo jiné vlákno nulový prvek s nižším indexem.

Pokud bychom chtěli mít více vnořených zámků, musíme kritické sekce pojmenovat, např. jako #pragma omp critical foo. Abychom se vyhnuli deadlocku, je nutné kritické sekce zamykat vždy ve stejném pořadí.

Posledním užitečným prostředkem pro paralelní programování je direktiva #pragma omp flush [(list)]. Ta definuje bod, v němž má vlákno garantováno, že hodnoty všech proměnných (resp. proměnných v seznamu list, pokud je přítomen) jsou konzistentní s hlavní pamětí, tzn. viditelné všem vláknům. Tato direktiva řeší problém, že některé hodnoty mohou být ukládány jen v registrech, kam vidí vždy jen vlákno, kterému registr patří. Flush způsobí vypláchnutí registrů do hlavní paměti.

Kromě direktiv je možné použít i rutiny z knihovny OpenMP, např. omp_init_lock inicializuje zámek, omp_set_lock zamkne zámek a omp_unset_lock odemkne zámek.

Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele Fifinas.