Programovanie pre model so zdieľanou pamäťou – OpenMP

Druhým z uvedených pamäťových modelov používaný v paralelných programoch je model so zdieľanou pamäťou. Takýto paralelný program zväčša býva vykonávaný viacerými **vláknami**, ktoré majú prístup ku spoločne zdieľaným prostriedkom, ako napríklad pamäť. Využívanie tohto modelu paralelného programovania so zdieľanou pamäťou nie je fenomén spojený s nástupom používania viacjadrových procesorov. Tento model bolo možné uplatniť už predtým s použitím viacerých jednojadrových procesorov v rámci jedného systému.

Pre tento účel boli vyvinuté aj softvérové nástroje, ktoré umožnili vykonávanie programov pomocou viacerých vlákien. V 90-tych rokoch 20. storočia bolo navrhnuté rozhranie (API) POSIX threads [1]. Podstatou tohto rozhrania bolo, že programátor explicitne navrhol, čo ktoré vlákno bude robiť a ako sa bude správať. Toto API bolo možné použiť s pomocou bežného prekladača na každom systéme, ktorý obsahoval knižnicu **pthreads**. Jeho výhodou a zároveň nevýhodou bolo, že sa jedná o nízkoúrovňové API, čo programátorovi poskytuje dobrú kontrolu nad vykonávaním vlákien, avšak za cenu vysokých požiadaviek na programátorské zručnosti.

V súčasnosti je vo vysokovýkonnom počítaní dominantným rozhraním pre programovanie paralelných programov so zdieľanou pamäťou rozhranie **OpenMP** (Open Multi-Processing) [2]. Jeho vznik možno datovať v roku 1997, kedy sa skupina programátorov a počítačových vedcov zhodla, že pre vytvorenie rozsiahlych vysokovýkonných paralelných programov je knižnica pthread príliš zložitá, čo viedlo k vzniku špecifikácie rozhrania OpenMP. Toto rozhranie umožnilo písanie paralelných programov na vyššej úrovni a tiež postupné paralelizovanie sériových programov bez potreby ich zásadného celkového predizajnovania nutného s použitím rozhrania MPI alebo pthreads [3].

Toto rozhranie bolo podporované od svojho vzniku viacerými počítačovými spoločnosťami, ako napríklad Compaq, Digital, IBM, Intel alebo Silicon Graphics. V súčasnosti existuje fórum, kde môžu členovia komunity zaoberajúcej sa vývojom rozhrania zdieľať svoje skúsenosti a informácie. Verzia OpenMP 2.5 (máj 2005) sa etablovala ako istý štandard. Po nej nasledovali ďalšie verzie OpenMP 3.0 (máj 2008), OpenMP 3.1 (júl 2011), OpenMP 4.0 (júl 2013) a OpenMP 4.1 (júl 2015). Súčasne najnovší štandard predstavuje verzia OpenMP 5.0 z novembra 2018.

Rozhranie OpenMP umožňuje vytvorenie a vykonávanie viacerých vlákien v prostredí s jedným procesorom pomocou zdieľania času, ako aj v SMP systémoch s väčším počtom procesorov. Na rozdiel od POSIX threads vyžaduje špeciálnu podporu prekladača jazyka C, C++ alebo Fortran. Ide o rozhranie, ktoré využíva tzv. direktívy prekladača `pragma`. V prípade, že ich použitý prekladač nepodporuje, je možné

program napísať tak, aby ho bolo možné skompilovať aj ako bežný sériový program.

Paralelný OpenMP program sa spúšťa v operačnom systéme ako jeden proces, ktorému operačný systém pridelí určité prostriedky, ako sú pamäť, procesor, registre a samotný výpočtový čas. V rámci tohto procesu môže existovať niekoľko navzájom nezávislých prúdov inštrukcií, ktoré sú vykonávané v podobe viacerých vlákien. Jednotlivé úlohy pridelené na riešenie vláknam môžu predstavovať podúlohy paralelného programu. Keďže sú všetky vlákna vykonávané v rámci toho istého procesu a teda aj v rámci toho istého virtuálneho adresného priestoru, majú všetky vlákna prístup k spoločným zdieľaným prostriedkom, kde si môžu vymieňať údaje alebo odovzdávať správy. Okrem toho má každé vlákno určité prostriedky, ktoré sú dostupné len samotnému vláknu, ako napríklad zásobník, register pre počítadlo programu alebo pamäť pre uloženie súkromných premenných.

V tejto kapitole sa oboznámime s koncepciou vytvárania, kompilovania a spúšťania paralelných programov so zdieľanou pamäťou s použitím rozhrania OpenMP. Tiež sa budeme venovať problematike úpravy existujúceho sériového programu na paralelný. Rozhranie OpenMP poskytuje širokú škálu rôznych direktív a klauzúl, ako aj množstvo funkcií v knižnici. Cieľom tejto učebnice nie je pokryť celé spektrum funkcionalít, ale oboznámiť čitateľa so základným konceptom vytvárania paralelných OpenMP programov vrátane najčastejšie používaných direktív a funkcií. Pre zistenie ďalších možností odporúčame štúdium dokumentácie a manuálov OpenMP [2, 4].

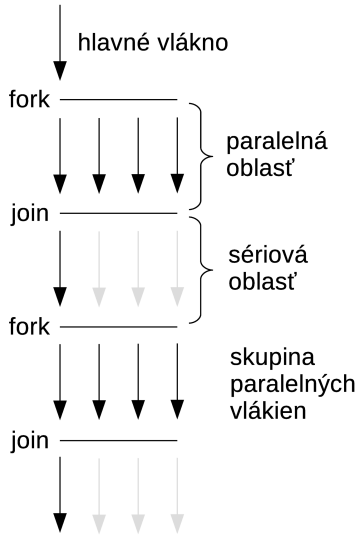
1 Jednoduchý program v prostredí OpenMP

Ako sme sa už boli zmienili, paralelný OpenMP program vytvoríme pomocou špeciálnych direktív určených pre preprocesor označovaných `#pragma`. Tieto umožňujú využívať funkcionality nad rámec špecifikácie štandardného jazyka. V prípade, že tieto direktívy prekladačom nie sú podporované, je možné ich vynechať.

V jazyku C alebo C++ je možné zadávať tieto direktívy v tvare:

`#pragma omp` *direktíva [klauzula[[,]klauzula]...] nový riadok*

V prípade, že nie je možné napísať direktívu na jeden riadok, je potrebné pre znak konca riadku (teda pred stlačením klávesy ENTER) zadať znak \.



Obr. 1: Model vykonávania OpenMP paralelného programu

Paralelná oblasť

Vytvorme teda jednoduchý program 1 *Hello world* s použitím OpenMP direktív. Na vytvorenie paralelnej oblasti v programe použijeme direktívu `#pragma omp parallel`. To znamená, že oblasť nasledujúca bezprostredne za touto direktívou bude vykonávaná paralelne v podobe viacerých vlákien.

Vykonávanie OpenMP programu začína ako jednovláknový program, kde inštrukcie vykonáva hlavné vlákno (master thread). Paralelné vykonávanie programu v skutočnosti nastane až v momente, keď program vstúpi do **paralelnej oblasti**. Po odchode z paralelnej oblasti pokračuje vykonávanie programu len v hlavnom vlákne a všetky ostatné vlákna (znázornené sivou farbou) sú pripojené k hlavnému vláknu. Takýto model označujeme ako **fork-join model**, znázornený na obrázku 1. Počet paralelných vlákien v skupine sa môže v jednotlivých paralelných oblastiach líšiť. V rámci paralelnej oblasti vlákna zdieľajú množstvo spoločných prostriedkov, ako napríklad `stdin` a `stdout`, avšak každé vlákno má svoj vlastný zásobník a počítadlo programu. Na konci paralelnej oblasti sa nachádza implicitná bariéra, to znamená, že vlákna, ktoré už dokončili svoju časť úlohy počkajú na dokončenie ostatných vlákien. Následne sú všetky vlákna okrem hlavného vlákna ukončené a pripojené v hlavnému vláknu.

Zdrojový kód 1: OpenMP program Hello world

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[])
4  {
5    #pragma omp parallel
6    {
7      printf("Hello world\n");
8    }
9
10   return 0;
11 }
```

Kompilovanie a spúšťanie OpenMP programov

Na mnohých systémoch je možné použiť na skompilovanie paralelného OpenMP programu bežný prekladač, ako napríklad `gcc` s tým, že je potrebné doplniť špeciálny parameter prekladača `-fopenmp` podľa príkladu:

```
gcc -g -Wall -o hello hello.c -fopenmp
```

Skompilovaný program potom môžme spustiť ako bežný sériový program pomocou príkazu:

```
./hello
```

a program vygeneruje nasledovný výstup obsahujúci niekoľko riadkov:

```
Hello world
Hello world
Hello world
Hello world
```

Počet vypísaných riadkov bude určený systémom a vo väčšine prípadov bude zodpovedať počtu dostupných procesorov v rámci použitého výpočtového systému. Počet vytvorených paralelných vlákien v

Poznámka: Niektoré staršie prekladače nemusia OpenMP podporovať. Niektoré prekladače môžu vyžadovať iný príkaz na skompilovanie OpenMP paralelného programu. Presné inštrukcie pre kompilovanie programu je možné získať od správcu systému.

Poznámka: Pri písaní paralelného programu používajúceho viacero vlákien je potrebné používať tzv. thread-safe verzie funkcií, ktoré dovoľujú, aby mohli byť zavolané vo viacerých vláknach súčasne bez vzniku konfliktu. Niekedy sú nesprávne označované ako reentrantné funkcie, ktoré môžu byť počas vykonávania prerušené a bezpečne opäť zavolané skôr, ako budú dokončené predchádzajúce volania funkcie [5, 6].

rámci procesu je možné aj explicitne určiť pomocou premennej prostredia príkazového riadku `OMP_NUM_THREADS` nasledovne (príklad pre sh, ksh, bash):

```
export OMP_NUM_THREADS=2
```

Po opätovnom spustení programu by sa zobrazil výstup obsahujúci dva riadky s pozdravom:

```
Hello world
Hello world
```

Pomocou tejto premennej prostredia `OMP_NUM_THREADS` určujeme počet vlákien, ktoré sa majú vytvoriť. V prípade dynamického určovania počtu vlákien táto premenná určuje maximálny počet možných použitých vlákien. Okrem tejto premennej je možné nastaviť aj ďalšie premenné. Premenná `OMP_SCHEDULE` sa dá použiť s direktívou `for` pri runtime plánovaní vykonávania vlákien. Premenná `OMP_DYNAMIC` určuje, či systém môže dynamicky nastavovať počet vlákien počas vykonávania programu. Premenná `OMP_NESTED` určuje, či je možné do seba vnárať paralelné oblasti. Premenná `OMP_MAX_ACTIVE_LEVELS`, určuje maximálny počet vnorených paralelných oblastí.

Klauzuly pre paralelnú oblasť

Pri použití OpenMP paralelnej oblasti je možné použiť nasledovné klauzuly:

- `private(zoznam)`,
- `firstprivate(zoznam)`,
- `shared(zoznam)`,
- `copying(zoznam)`,
- `proc_bind(master | close | spread)`
- `reduction(operácia:zoznam)`,
- `default(shared | none)`,
- `num_threads(počet_vlákien)`.

Presnejší význam jednotlivých klauzúl bude vysvetlený v podkapitole 2.

Druhou možnosťou, ako určiť počet vytvorených vlákien priamo v programe, je rozšírenie direktívy o klauzulu `num_threads(počet_vlákien)`. Riadok 5 programu 1 by potom vyzeral nasledovne:

```
#pragma omp parallel num_threads(2)
```

pričom hodnota v zátvorke, zadaná ako konštanta alebo premenná, určuje počet vytvorených vlákien pre danú paralelnú oblasť.

Knižnica OpenMP

Rozhranie OpenMP okrem samotných direktív pragma disponuje aj množstvom funkcií, pomocou ktorých je možné zisťovať stav alebo nastavovať spôsob vykonávania vlákien. Podobne ako v MPI, aj v OpenMP je možné jednotlivé vlákna identifikovať na základe ich

poradového čísla od 0 do $n - 1$, pričom n predstavuje celkový počet vlákien v určitej paralelnej oblasti.

Pre použitie funkcií z tejto knižnice je potrebné vložiť do zdrojového kódu programu hlavičkový súbor `omp.h`. Je potrebné pripomenúť, že dobre napísaný OpenMP program by malo byť možné skompilovať aj v systéme bez podpory OpenMP ako obyčajný sériový program. To by však nebolo možné po vložení hlavičkového súboru, pretože tento by nebolo možné v systéme nájsť a preklad programu by skončil s chybou. Na vyriešenie tohto problému je možné použiť funkcionalitu preprocesora umožňujúcu podmienený preklad a hlavičkový súbor `omp.h` do zdrojového kódu vkladať len v tom prípade, že systém podporuje OpenMP.

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

Úloha: Zistite, či je definované makro _OPENMP. Toto makro reprezentuje desiatkové celé číslo. Napíšte program, ktorý túto hodnotu vypíše. Aký je význam tejto hodnoty?

Funkcia omp_get_thread_num

```
int omp_get_thread_num (void)
```

Funkcia ako návratovú hodnotu udáva poradové číslo vlákna v rámci skupiny vlákien. Na základe tejto hodnoty je možné identifikovať vlákno, prípadne určiť, ktorú časť podúlohy má vykonávať.

Funkcia omp_get_num_threads

```
int omp_get_num_threads (void)
```

Funkcia ako návratovú hodnotu udáva celkový počet vlákien v skupine. V prípade volania funkcie vo vnorenej paralelnej oblasti sa táto hodnota vzťahuje k najvnútornejšej oblasti. Volanie funkcie mimo paralelnej oblasti vráti hodnotu rovnú 1.

Funkcia omp_set_num_threads

```
void omp_set_num_threads (int num_threads)
```

Funkcia nastavuje počet vlákien použitých na vykonávanie nasledujúcej paralelnej oblasti v prípade, že nie je použitá klauzula `num_threads`.

Funkcia omp_get_max_threads

```
int omp_get_max_threads (void)
```

Funkcia ako návratovú hodnotu udáva maximálny počet vlákien, ktoré môžu byť použité na vytvorenie novej skupiny vlákien.

Upravme program 1 tak, aby každý proces vypísal svoje poradové číslo a celkový počet vlákien v skupine, podľa programu 2. Tento program je možné skompilovať s podporou aj bez podpory OpenMP (teda s použitím aj bez použitia prepínača `-fopenmp`). Na riadku 11

Úloha: Aký bude výstup programu 2, ak by sme z neho odstránili klauzulu private?

bolo potrebné pridať klauzulu určujúcu premennú `thread_num` ako súkromnú premennú. Tiež si môžme všimnúť, že premenné `thread_num` a `num_threads` sú na riadku 8 inicializované s hodnotami 0 a 1. Týmto sme docielili to, že v prípade, že by bol program prekladaný bez podpory OpenMP, vo výstupe programu sa objavia tieto predvolené hodnoty premenných.

Zdrojový kód 2: Rozšírený OpenMP program Hello world

```
1  #include <stdio.h>
2  #ifdef _OPENMP  //podmienený preklad
3    #include <omp.h>
4  #endif
5
6  int main(int argc, char* argv[])
7  {
8    int thread_num = 0, num_threads = 1;
9
10   #pragma omp parallel private(thread_num)
11   { //začiatok paralelnej oblasti
12     #ifdef _OPENMP  //podmienený preklad
13       thread_num = omp_get_thread_num();
14       num_threads = omp_get_num_threads();
15     #endif
16     printf("Hello world, vlakno %d z %d\n",
             thread_num, num_threads);
17   } //koniec paralelnej oblasti
18
19   return 0;
20 }
```

Poznámka: Je možné, že program bude dávať rovnaký výstup aj bez použitia klauzuly private. Dôvodom je, že každé vlákno použije iný register pre uloženie hodnoty premennej thread_num. Na to sa však spoliehať nemôžme, pretože ak by bola hodnota uložená v pamäti, program by nefungoval správne.

Vytvorený program skúste spustiť tak, aby bol vykonávaný viacerými vláknami. Môžme si všimnúť, že poradie vypísania výstupu jednotlivých riadkov na `stdout` nie je pre vlákna určené, a preto bude výstup pri každom spustení paralelného programu vypísaný v náhodnom poradí.

2 Rozdeľovanie práce

V rámci OpenMP paralelného programu je možné rozdeliť úlohu na podúlohy tak, aby boli jednotlivé podúlohy vykonávané jednotlivými vláknami. Toto rozdeľovanie práce je možné vykonať viacerými spôsobmi pomocou direktív [7]:

- for,
- section,
- task,
- single.

Tieto direktívy sa spravidla používajú priamo vo vnútri paralelnej oblasti (lexikálneho alebo statického rozsahu). Je však možné definovať aj dynamický rozsah paralelnej oblasti. V takomto prípade sa

môže nachádzať použitá direktíva aj mimo paralelnej oblasti definovanej pomocou `#pragma omp parallel`. Typickým príkladom môže byť použitie direktív pre rozdeľovanie práce alebo synchronizáciu vo funkcii, ktorá bola zavolaná v rámci paralelnej oblasti. Takéto oblasti označujeme ako osirotené (orphaned) [8].

Direktíva for

```
#pragma omp for
```

Pomocou direktívy `for` je možné rozdeliť prácu vykonávanú v jednotlivých iteráciách cyklu `for` medzi všetky vlákna. Na prvý pohľad sa môže zdať, že pomocou tejto direktívy je možné veľmi ľahko paralelizovať veľké množstvo úloh. Treba si však uvedomiť, že túto direktívu nie je možné použiť s iným typom cyklu, ako napríklad `while` alebo `do-while`. Aj keby sme tieto typy cyklov nahradili cyklom `for`, nebolo by ich možné vykonávať paralelne. Dôvodom je fakt, že pre použitie tejto direktívy je nutné, aby bolo možné vyčísliť počet opakovaní cyklu ešte pred jeho začatím na základe výrazov uvedených v príkaze `for(start; end; increment)`. Inak povedané, cyklus `for` musí byť v kánonickej forme. To znamená, že:

- riadiaca premenná cyklu musí byť údajového typu int alebo ukazovateľ (nie float),
- výrazy `start`, `end` a `increment` musia byť navzájom kompatibilného údajového typu,
- výrazy `start`, `end` a `increment` sa nesmú zmeniť počas vykonávania tela cyklu,
- hodnota riadiacej premennej cyklu sa nesmie meniť v tele cyklu (iba pri prechode na ďalšiu iteráciu na základe výrazu `increment`).

Výraz `start` môže obsahovať iba jednoduché priradenie (napríklad `i = 0`). Výraz `end` môže obsahovať niektorý z relačných operátorov `< <= => >`. Výraz `increment` môže obsahovať nasledovné operácie `++i`, `--i`, `i++`, `++i`, `i += incr`, `i -= incr`, `v = v + incr`, `v = v - incr`, `v = incr + v`, `v = incr - v`.

Na základe týchto obmedzení je systém počas behu schopný vopred určiť počet iterácií cyklu a následne rozdeliť tieto iterácie medzi vlákna. Ďalší problém môže nastať v prípade, ak by jednotlivé iterácie cyklu na seba nadväzovali, to znamená, že v určitej iterácii by sme potrebovali použiť výsledok získaný v predchádzajúcej iterácii cyklu. Keďže jednotlivé iterácie cyklu sú priradené rôznym vláknam, nie je možné zaručiť, že budú vykonávané v potrebnom poradí.

Úloha: V nasledujúcom cykle je zjavné, že jednotlivé iterácie na seba nadväzujú.
a[0] = 0;
for(i = 1; i < n; i++) a[i] = a[i-1] +i;
Pokúste sa zdrojový kód upraviť tak, aby bolo možné odstrániť závislosť medzi iteráciami.

Na nasledujúcom programe 3 si vysvetlíme použitie direktívy `for`. Cyklus `for` musí nasledovať bezprostredne za riadkom s direktívou `#pragma omp for`. Na riadku 7 si môžme všimnúť, že premenná `n` sa bude používať ako zdieľaná, zatiaľ čo premenná `i` nesmie byť zdieľaná, pretože by si jej hodnotu vlákna navzájom prepisovali, a preto sa bude používať ako súkromná pre každé vlákno. Po dokončení všetkých iterácií cyklu sú všetky vlákna synchronizované implicitnou bariérou.

Úloha: Napíšte program, ktorý zistí a vypíše, aké je prednastavené rozvrhovanie pre vykonávanie iterácií cyklu for viacerými vláknami, čiže ktoré iterácie sú pridelené ktorým vláknam.

Zdrojový kód 3: Program s direktívou for

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

int main(int argc, char* argv[])
{
  int i, n = 10;
  #pragma omp parallel default(none) shared(n) private(i)
  {
    #pragma omp for
    for(i = 0; i < n; i++) {
      printf("Vlakno %d z %d: iteracia %d\n",
          omp_get_thread_num(), omp_get_num_threads(), i);
    }
  }
  return 0;
}
```

V prípade, že paralelná oblasť obsahuje iba `for` cyklus, ktorý má byť vykonaný paralelne a nič iné, je možné tieto dve direktívy spojiť do jedného riadku nasledovne:

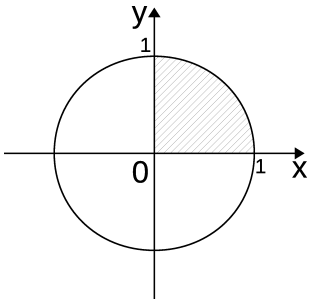
```
#pragma omp parallel for default(none) shared(n) private(i)
```

Klauzuly pre direktívu for

Pri použití OpenMP direktívy `for` je možné použiť nasledovné klauzuly:

- `private(zoznam)`,
- `firstprivate(zoznam)`,
- `lastprivate(zoznam)`,
- `schedule(typ[:počet])`,
- `collapse(počet)`,
- `ordered(počet)`,
- `reduction(operácia:zoznam)`,
- `nowait`.

Presnejší význam jednotlivých klauzúl bude vysvetlený v podkapitole 2.



Obr. 2: Výpočet hodnoty obsahu plochy pomocou určitého integrálu

Riešená úloha

Na nasledujúcej úlohe si ukážeme, ako je možné pomocou OpenMP paralelného programu vypočítať hodnotu konštanty π s určitou presnosťou. Podstata výpočtu bude spočívať v numerickom výpočte hodnoty určitého integrálu pomocou obdĺžnikovej metódy. Obsah kruhu s polomerom $r = 1$ je rovný hodnote konštanty π. Zároveň vieme pomocou integrálu určiť obsah plochy v prvom kvadrante pod krivkou opisujúcou jednotkovú kružnicu so stredom v počiatku súradnicovej

sústavy (vzťah 1), tak, ako je znázornené na obrázku 2. Hodnotu π potom vieme vypočítať podľa vzťahu 2.

$$x^2 + y^2 = r^2 \tag{1}$$

$$\pi = 4 \int_0^1 \sqrt{1 - x^2} dx \tag{2}$$

Výpočet môžme zrealizovať podľa programu 4.

Zdrojový kód 4: Program na výpočet hodnoty π

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

int main(int argc, char* argv[])
{
  long i, n;
  double x, y, dx, pi = 0;

  //zistenie hodnoty n
  n = atol(argv[1]);

  //výpočet hodnoty šírky plôšky
  dx = 1.0 / n;

  //paralelný výpočet iterácií cyklu
  #pragma omp parallel for default(none) private(i, x, y)
    shared(dx, n) reduction(+:pi)
  for(i = 0; i < n; i++) {
    x = dx * i;
    y = sqrt(1-x*x);
    pi += dx * y;
  }

  printf("Hodnota PI: %.16f\n", 4*pi);

  return 0;
}
```

Podstata obdĺžnikovej metódy spočíva v tom, že sa počítaná plocha rozdelí na veľký počet n úzkych plôšok, pričom sa predpokladá, že sú tak úzke, že túto plochu možno nahradiť obdĺžnikom so šírkou $1/n$ a výškou rovnou odpovedajúcej funkčnej hodnote pre príslušnú plôšku nachádzajúcu sa na príslušnej hodnote na osi x. Počet plôšok, na ktoré sa má plocha rozdeliť zadávame ako parameter pri spúšťaní programu.

Poznámka: Program je potrebné kompilovať s prepínačom -lm, pretože pri výpočte bola použitá matematická knižnica.

Program môžme spustiť niekoľkokrát pre rôzne hodnoty n, pričom sledujeme čas vykonávania programu a výslednú hodnotu π. Tiež môžme overiť, aký vplyv bude mať zmena hodnoty premennej prostredia `OMP_NUM_THREADS` na čas vykonávania programu, ktorý môžme odmerať príkazom `time`.

Direktíva sections

```
#pragma omp sections
```

Ďalšiu direktívu pre rozdeľovanie práce, pomocou ktorej je možné určiť jednotlivé podúlohy predstavujú sekcie. Pomocou nej je možné definovať nezávislé podúlohy programu. Každá sekcia je potom priradená jednému z vlákien vytvorených v paralelnej oblasti. Ak paralelná oblasť obsahuje viac vlákien ako sekcií, všetky sekcie sú vykonávané paralelne a niektoré vlákna nebudú vykonávať žiadnu sekciu. V prípade, že je sekcií viac ako dostupných vlákien, budú tieto sekcie postupne priraďované na vykonávanie vláknam. Vlákna sú po dokončení všetkých sekcií synchronizované implicitnou bariérou.

Na nasledujúcom programe 5 si vysvetlíme použitie direktívy `sections`. Použitie `#pragma omp section` pri prvej sekcii nie je povinné.

Podobne ako pri direktíve `for`, aj tu je možné spojiť direktívu pre paralelnú oblasť a sekcie do jedného riadku nasledovne:

```
#pragma omp parallel sections
```

Pri použití paralelných sekcií je potrebné venovať zvýšenú pozornosť na vyvažovanie záťaže medzi vláknami. V prípade, že jednotlivé sekcie budú obsahovať rôzne výpočtovo náročné úlohy, môže nastať situácia, že niektoré vlákna dokončia svoju prácu skôr a budú musieť na konci sekcií čakať na dokončenie poslednej sekcie.

Klauzuly pre direktívu sections

Pri použití OpenMP direktívy `sections` je možné použiť nasledovné klauzuly:

- `private(zoznam)`,
- `firstprivate(zoznam)`,
- `lastprivate(zoznam)`,
- `reduction(operácia:zoznam)`,
- `nowait`.

Presnejší význam jednotlivých klauzúl bude vysvetlený v podkapitole 2.

Zdrojový kód 5: Program s direktívou sections

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

int main(int argc, char* argv[])
{
  #pragma omp parallel
  { //začiatok paralelnej oblasti
    #pragma omp sections
    {   //začiatok sekcií
      #pragma omp section   //sekcia A
      {
        printf("sekciu A vykonava vlakno %d z %d\n",
               omp_get_thread_num(), omp_get_num_threads());
      }
      #pragma omp section   //sekcia B
      {
        printf("sekciu B vykonava vlakno %d z %d\n",
               omp_get_thread_num(), omp_get_num_threads());
      }
      #pragma omp section   //sekcia C
      {
        printf("sekciu C vykonava vlakno %d z %d\n",
               omp_get_thread_num(), omp_get_num_threads());
      }
      #pragma omp section   //sekcia D
      {
        printf("sekciu D vykonava vlakno %d z %d\n",
               omp_get_thread_num(), omp_get_num_threads());
      }
    }   //koniec sekcií
  } //koniec paralelnej oblasti

  return 0;
}
```

Direktíva task

`#pragma omp task`

Na vykonávanie OpenMP paralelného programu sa používa viacero vlákien, ktoré vykonávajú im pridelené čiastkové úlohy. Pri vstupe hlavného vlákna do paralelnej oblasti sa vytvorí skupina vlákien a množina implicitných podúloh, pridelených jednotlivo vláknam. Ak počas ich vykonávania nastane požiadavka na vytvorenie ďalšej podúlohy, je na to možné použiť direktívu `task`.

Nová vytvorená podúloha bude vykonávať kód definovaný v príslušnom bloku ako dcérska podúloha. Táto podúloha bude mať prístup k rovnakým údajom ako ostatné vlákna v rámci najvnútornejšieho paralelného regiónu. Dcérska úloha môže pozastaviť vykonávanie pôvodnej podúlohy vlákna a spustiť vykonávanie dcérskej podúlohy

alebo ho odsunúť na neskôr. Čakanie na dokončenie dcérskej podúlohy je možné zabezpečiť použitím direktívy `taskwait`.

Na nasledujúcom programe 6 si vysvetlíme použitie direktívy `task`. Program rekurzívne vypočíta n-tý člen Fibonacciho postupnosti ($F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ pre $n \leq 2$).

Zdrojový kód 6: Program s direktívou task

```
#include <stdio.h>
#include <omp.h>
//rekurzívna funkcia
int F(int n)
{
  int i, j;
  if(n < 2) return(1);
  else {
    //vytvorenie novej podúlohy
    #pragma omp task shared(i)
    {
      i = F(n-1);
      printf("%d-ty clen: vlakno %d\n",
             n-1, omp_get_thread_num());
    }
    //vytvorenie novej podúlohy
    #pragma omp task shared(j)
    {
      j = F(n-2);
      printf("%d-ty clen: vlakno %d\n",
             n-2, omp_get_thread_num());
    }
    //explicitná synchronizácia vlákien
    #pragma omp taskwait
    return(i + j);
  }
}

int main(int argc, char* argv[])
{
  int i;
  #pragma omp parallel
  { //začiatok paralelnej oblasti
    #pargma omp single nowait
    //cyklus bude vykonávaný iba jedným vláknom
    for(i = 0; i < 6; i++)
      printf("F(%d) = %d\n", i, F(i));
  } //koniec paralelnej oblasti

  return 0;
}
```

Klauzuly pre direktívu task

Pri použití OpenMP direktívy `task` je možné použiť nasledovné klauzuly:

- `untied`,
- `mergeable`,
- `private(zoznam)`,
- `firstprivate(zoznam)`,
- `shared(zoznam)`,
- `depend(závislost':zoznam)`,
- `default(shared | none)`.

Presnejší význam jednotlivých klauzúl bude vysvetlený v podkapitole 2.

Direktíva single

`pragma omp single`

Pomocou tejto direktívy je možné zabezpečiť, že blok kódu nasledujúci za touto direktívou bude vykonávaný iba jedným vláknom zo skupiny vlákien v rámci paralelnej oblasti. Nedá sa určiť, ktorému vláknu bude pridelené vykonávanie bloku. Ostatné vlákna nevykonávajú žiadnu činnosť a čakajú, kým nie je blok dokončený. Na konci bloku sú vlákna synchronizované pomocou implicitnej bariéry. Po dokončení bloku pokračujú všetky vlákna vo svojej činnosti.

Na nasledujúcom programe 7 si vysvetlíme použitie direktívy `single`. Program je potrebné spustiť s viacerými vláknami.

Zdrojový kód 7: Program s direktívou single

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char* argv[])
5  {
6    #pragma omp parallel
7    { //začiatok paralelnej oblasti
8      #pragma omp single
9      {   //začiatok bloku vykonávaného 1 vláknom
10       printf("blok single vykonava vlakno %d\n",
                 omp_get_thread_num());
11     }   //koniec bloku vykonávaného 1 vláknom
12
13     printf("paralelnu oblast vykonava vlakno %d\n",
                omp_get_thread_num());
14   } //koniec paralelnej oblasti
15
16   return 0;
17 }
```

Klauzuly pre direktívu single

Pri použití OpenMP direktívy `single` je možné použiť nasledovné klauzuly:

- `private(zoznam)`,
- `firstprivate(zoznam)`,
- `copyprivate(zoznam)`,
- `nowait`.

Presnejší význam jednotlivých klauzúl bude vysvetlený v podkapitole 2.

Klauzuly direktív

Jednotlivé OpenMP direktívy umožňujú zadávať rôzne doplnkové informácie formou klauzúl. Pomocou nich je možné lepšie kontrolovať vykonávanie konštrukcií paralelného programu. Niektoré klauzuly môžu obsahovať ako parameter aj hodnoty uložené v premenných alebo hodnotu výrazu. Keďže klauzuly sa spracúvajú ešte pred ich vykonaním, hodnoty ich argumentov musia byť známe pred vstupom do paralelnej oblasti. Poradie zadávania klauzúl nie je striktne dané a môžu byť v ľubovoľnom poradí. V nasledujúcej časti uvedieme opis niektorých najčastejšie používaných klauzúl.

Klauzula shared

`shared(zoznam)`

Klauzula umožňuje definovať zoznam premenných, ktoré budú pre všetky vlákna spoločné (hodnoty v týchto premenných budú zdieľané medzi vláknami v skupine). Všetky premenné zo zoznamu sú uložené v pamäti ako jedna kópia.

Premenné sú inicializované s hodnotou, ktorú premenné obsahovali pred vstupom do paralelnej oblasti. V paralelnej oblasti ich môžu vlákna čítať alebo modifikovať. Pri výstupe z paralelnej oblasti premenné v hlavnom vlákne prevezmú výsledné hodnoty premenných.

V prípade, že nie je v direktívach `parallel` alebo `task` použitá klauzula `default` ani `private`, všetky premenné sú predvolene považované za zdieľané.

Klauzula private

`private(zoznam)`

Klauzula umožňuje definovať zoznam premenných, ktoré budú pre každé vlákno súkromné (hodnota v týchto premenných nebude zdieľaná medzi vláknami v skupine).

Každé vlákno bude disponovať svojimi vlastnými premennými, ktorých vlastnosti a údajový typ je identický s pôvodnými premennými.

Pri vstupe a výstupe z paralelnej oblasti sú hodnoty v premenných nedefinované.

Niektoré premenné nie je potrebné prehlásiť ako súkromné, napríklad riadiacu premennú cyklu `for`, ako aj formálne parametre a premenné použité v lokálnych funkciách v rámci paralelnej oblasti. Tieto premenné sú automaticky považované za súkromné.

Poznámka: Inicializačná hodnota súkromnej premennej nie je rovná pôvodnej hodnote premennej pred vstupom do paralelnej oblasti.

Klauzula firstprivate

```
firstprivate(zoznam)
```

Klauzula umožňuje definovať zoznam premenných, ktoré budú pre každé vlákno súkromné, pričom tieto premenné inicializuje s pôvodnými hodnotami odpovedajúcich premenných pred vstupom do paralelnej oblasti. Pripomeňme, že pri použití klauzuly `private` nie je hodnota v premenných pri vstupe a výstupe z paralelnej oblasti definovaná.

Klauzula lastprivate

```
lastprivate(zoznam)
```

Klauzula umožňuje definovať zoznam premenných, ktoré budú pre každé vlákno súkromné, pričom tieto pri výstupe z paralelnej oblasti odovzdajú svoju hodnotu pôvodným premenným v hlavnom vlákne. Pripomeňme, že pri použití klauzuly `private` nie je hodnota v premenných pri vstupe a výstupe z paralelnej oblasti definovaná. Pri použití klauzuly s direktívou `for` budú pri výstupe z paralelnej oblasti odovzdané hodnoty takýchto premenných zo sekvenčne poslednej iterácie cyklu. Pri použití s direktívou `section` budú predané hodnoty premenných uvedených v zozname z lexikálne poslednej sekcie (poslednej uvedenej sekcie, nie poslednej vykonávanej).

Klauzula default

```
default(shared | none)
```

Klauzula umožňuje explicitne určiť, či majú byť premenné v rámci paralelnej oblasti prednastavené ako zdieľané, alebo bude zdieľanie každej premennej určené explicitne.

V prípade, že je použitá hodnota `shared`, sú všetky používané premenné v rámci paralelnej oblasti používané ako zdieľané. V prípade použitia hodnoty `none` je potrebné explicitne zadať atribút pre zdieľanie každej použitej premennej v paralelnej oblasti.

Odporúča sa používať klauzulu `default(none)`, čo síce môže byť menej pohodlné, avšak takýmto spôsobom je možné lepšie predísť neúmyselným chybám.

Klauzula nowait

`nowait`

Klauzula umožňuje vypnúť synchronizáciu implicitnou bariérou po dokončení všetkých iterácií cyklu.

Direktívy pre rozdeľovanie práce (`for`, `sectons` a `single`) na svojom konci synchronizujú vlákna pomocou implicitnej bariéry prv, než program pokračuje ďalej. V dôsledku toho dochádza k tomu, že vlákna musia na seba čakať a nevykonávajú žiadnu prácu. S použitím klauzuly `nowait` vlákno po skončení svojej podúlohy pokračuje do ďalšej časti programu.

Použitie klauzuly môže viesť k skráteniu času vykonávania programu. Na druhej strane je potrebné si byť istý, že odstránenie synchronizácie na konci bloku nebude mať za následok chybu v programe.

Klauzula schedule

`schedule(typ[,počet])`

Úloha: Modifikujte program 4 tak, že v ňom použijete klauzulu schedule. Otestujte rýchlosť programu pre rôzne typy rozvrhovania a rôzneho počtu iterácií v dávkach. Program upravte tak, aby vypísal, ktoré iterácie cyklu vykonáva ktoré vlákno. Pozorujte a pokúste sa vysvetliť, ako je určené rozvrhovanie typu guided.

Klauzula umožňuje definovať spôsob rozvrhovania iterácií vláknam, pričom určitý počet iterácií môže byť pridelený jednému vláknu naraz.

Parameter `počet` (chunk) určuje počet po sebe nasledujúcich iterácií, ktoré majú byť vláknu priradené v jednej dávke. Takto je možné definovať granularitu rozdeľovania v rámci implicitne vytvorených podúloh. V prípade, že počet nie je zadaný, použije sa hodnota 1.

Ako typ rozvrhovania je možné použiť hodnoty:

- `static` – iterácie sú rozdelené na dávky, ktoré sú zaradom priraďované vláknam spôsobom round-robin,
- `dynamic` – iterácie sú rozdelené na dávky, ktoré sú priraďované vláknam podľa požiadavky (voľnému vláknu je priradená ďalšia dávka),
- `guided` – iterácie sú rozdelené na dávky, a priraďované vláknam v zmysle dynamického rozvrhovania s tým rozdielom, že počet iterácií v dávke sa postupne zmenšuje až k zadanému počtu (okrem poslednej dávky, ktorá môže byť aj menšia než zadaný počet),
- `auto` – rozhodnutie o prideľovaní iterácií vláknam je ponechané na prekladač a/alebo systém pri vykonávaní programu,
- `runtime` – spôsob rozvrhovania aj počet iterácií v dávke sa riadia na základe definovanej premennej prostredia `OMP_SCHEDULE`.

Klauzula reduction

`reduction(operácia:zoznam)`

Klauzula umožňuje definovať zoznam premenných, z ktorých budú po dokončení paralelnej oblasti výsledky skombinované pomocou operácie redukcie.

Pri niektorých paralelných výpočtoch je potrebné na konci výpočtu získať globálny výsledok kombináciou čiastkových výsledkov z jednotlivých vlákien. Efektívnejším a menej náročným riešením na rozdiel od toho, aby každé vlákno pripojilo svoj výsledok do zdieľanej premennej, je použitie operácie redukcie. Premenné uvedené v zozname sú počas vykonávania všetkých vlákien považované za súkromné a budú inicializované hodnotou. Po dokončení paralelnej oblasti sa výsledky z týchto premenných zo všetkých vlákien skombinujú pomocou zvolenej binárnej, asociatívnej operácie. V tabuľke 1 je uvedený zoznam operácií, ktoré je možné pre redukciu použiť.

Tabuľka 1: Zoznam možných operácií pre redukciu

Operátor	Inicializačná hodnota
`+`	0
`*`	1
`-`	0
`&`	~0
`|`	0
`^`	0
`&&`	1
`||`	0
`max`	najmenšia možná hodnota
`min`	najväčšia možná hodnota

Klauzula num_threads

threads `num_threads(počet_vlákien)`

Klauzula umožňuje nastaviť počet vlákien, ktoré majú byť použité na vykonanie paralelnej oblasti.

Počet vlákien, ktoré majú vykonávať paralelnú oblasť je možné definovať pomocou premennej prostredia `OMP_NUM_THREADS` alebo nastaviť pomocou funkcie `omp_set_num_threads`. Klauzula `num_threads` dovoľuje dočasne zmeniť počet vlákien v rámci paralelného regiónu a vzťahuje sa len k direktíve, s ktorou bola použitá. Všetky nasledujúce direktívy budú vykonávané s pôvodným počtom vlákien.

Táto klauzula je užitočná, keď je správnosť programu obmedzená na presný počet vlákien, ktoré paralelný program vykonávajú. Tiež je ju možné použiť na minimalizáciu času vykonávania programu tým, že pomocou nej vieme vhodne upraviť počet vlákien vykonávajúcich paralelnú oblasť.

Klauzula copyin

`copyin(zoznam)`

Klauzula umožňuje definovať zoznam súkromných premenných, do ktorých bude vo všetkých vláknach vykonávajúcich paralelnú oblasť

nakopírovaná hodnota z premennej hlavného vlákna. Nakopírovanie hodnôt je vykonané hneď po vytvorení vlákien, skôr než začnú vykonávať samotný program.

Klauzula collapse

`collapse(počet)`

Klauzula umožňuje určiť počet vnorených cyklov, ktoré majú byť použité na vytvorenie jedného priestoru cyklov.

3 Synchronizácia

Pri vykonávaní paralelného OpenMP programu viacerými vláknami sú vlákna zvyčajne synchronizované implicitnou bariérou pri vstupe a výstupe z paralelnej oblasti. Niekedy však môže nastať situácia, keď je potrebné vlákna synchronizovať explicitne alebo inak riadiť ich vykonávanie. V nasledujúcej časti uvedieme niektoré bežne používané direktívy OpenMP, ktoré je možné použiť za týmto účelom.

Direktíva master

`#pragma omp master`

Pomocou tejto direktívy je možné definovať určitý blok zdrojového kódu programu v rámci paralelnej oblasti, ktorý má byť vykonávaný iba hlavným vláknom. Ostatné vlákna tento blok nevykonávajú. Pri vstupe a výstupe z bloku nie je použitá implicitná bariéra, a preto ho ostatné vlákna vynechajú.

Táto direktíva je svojou funkciou podobná direktíve `single`. Hlavné rozdiely spočívajú v tom, že direktíva `single` má pri vstupe a výstupe z oblasti implicitnú bariéru. Taktiež časť programu označená direktívou `single` môže byť vykonávaná ktorýmkoľvek jedným vláknom, zatiaľ čo pri `master` je vykonávaná hlavným vláknom. Direktíva `master` je efektívnejšia z pohľadu jej väčšej jednoduchosti.

Používa sa v prípadoch, keď potrebujeme, aby hlavné vlákno obslúžilo napríklad I/O požiadavky (načítanie vstupu, vypísanie výsledku), zatiaľ čo ostatné vlákna už môžu pokračovať vo vykonávaní nasledujúcej časti programu. Pre lepšie pochopenie uvádzame ukážku časti programu 8 [7].

Zdrojový kód 8: Program s direktívou master

```
#pragma omp parallel
{
  #pragma omp for
  {
    ...     //prvá časť výpočtov v cykle
  }
  #pragma omp master
  printf(...)   //výpis priebežných výsledkov
  ...       //ďalšia časť výpočtov
}
```

Direktíva barrier

`#pragma omp barrier`

Jedným z predpokladov pre správnu funkčnosť paralelného programu so zdieľanou pamäťou je to, že k zdieľaným premenným sa bude pristupovať v správnom poradí. To znamená, že nebudeme používať hodnotu uloženú v zdieľanej premennej skôr, než táto hodnota bude nastavená iným vláknom. Z tohto dôvodu OpenMP poskytuje nástroj pre synchronizáciu vlákien. Pomocou neho je možné zabezpečiť, že vlákna budú k zdieľanej premennej pristupovať v správnom poradí.

Ako sme sa už zmienili, na konci každej paralelnej oblasti alebo oblasti pre rozdeľovanie práce sa nachádza implicitná bariéra. Explicitnú synchronizáciu pomocou bariéry je možné použiť na doplnenie ďalších bodov synchronizácie v prípade potreby. Každé vlákno v skupine vykonávajúcej paralelnú oblasť musí prejsť bariérou. Toto mu je však umožnené až v momente, keď všetky vlákna v skupine dosiahnu toto miesto a bariérou prejdú spoločne. Potom vlákna pokračujú vo vykonávaní svojich podúloh nezávisle.

Pre lepšie pochopenie uvádzame časť príkladu programu 9 s použitím explicitnej bariéry [7]. Po vykonaní zatriedenia prvkov v riadkoch matice a, vlákna musia na seba počkať. Tým sa zabezpečí, že pred vykonaním ďalších výpočtov s hodnotami uloženými v matici a tieto budú určite zoradené.

Zdrojový kód 9: Program s direktívou barrier

```
#pragma omp parallel private(i, step)
{
  i = omp_get_thread_num();
  step = omp_get_num_threads();
  while (i < n) {
    sort(a[i][0];
    i += step;
  }
  #pragma omp barrier
  ...   //ďalšie výpočty s maticou a
}
```

Direktíva taskwait

`#pragma omp taskwait`

Pomocou tejto direktívy je možné vytvoriť bod plánovania úloh, ktorý umožňuje systému kontrolovať vykonávanie podúloh. V prípade, že vlákno príde na takýto bod, jeho vykonávanie je pozastavené dovtedy, kým nie sú dokončené všetky jeho dcérske podúlohy vytvorené pred príchodom do tohto bodu. Túto direktívu nie je možné kombinovať s použitím príkazov `if`, `while`, `do`, `switch` alebo `label`.

Direktíva critical

`#pragma omp critical [` *meno* `]`

Úloha: Upravte program 4 tak, aby nebolo potrebné použiť klauzulu reduction. Pre výpočet hodnoty výsledku vo vláknach použite súkromné premenné a čiastočné výsledky z nich skombinujte do zdieľanej premennej s použitím kritickej oblasti.

Pri vykonávaní paralelného programu so zdieľanou pamäťou pristupujú viaceré vlákna ku zdieľaným premenným. V dôsledku toho nastáva situácia, že navzájom súperia o prístup k takejto premennej. Pre zabezpečenie správneho fungovania paralelného programu je potrebné zabezpečiť, aby bol prístup v tom istom čase umožnený len jednému vláknu. Časť programu, kde vlákno manipuluje s takouto premennou sa nazýva **kritická oblasť**. Na riadenie prístupu vlákien do kritickej oblasti je možné použiť **vzájomné vylúčenie** (mutex) pomocou direktívy `critical`.

Použitie direktívy určuje, že nasledujúci blok programu, nachádzajúci sa v kritickej oblasti, bude vykonávaný súčasne iba jedným vláknom. Nepovinnou časťou direktívy je meno kritickej oblasti. V programe sa môže nachádzať kritických oblastí s rovnakým menom aj viac. V takom prípade bude vlákno čakať na začiatku kritickej oblasti dovtedy, kým sa nebude žiadne iné vlákno nachádzať v ktorejkoľvek z kritických oblastí s rovnakým menom.

Direktíva ordered

`#pragma omp ordered`

Táto direktíva sa používa spoločne s cyklom `for` a zabezpečuje, aby boli príkazy v bloku nachádzajúcom sa v tele cyklu, označenom direktívou `ordered`, vykonávané v poradí odpovedajúcom poradiu iterácií cyklu. To znamená, že uvedené bloky v tele cyklu sa musia vykonávať sekvenčne, zatiaľ čo zvyšná časť tela cyklu môže byť vykonávaná paralelne.

Pomocou direktívy `ordered` môže program jednoducho vypísať výsledky v zoradenom poradí podľa poradia iterácií cyklu. Modifikujeme program 3 pridaním direktívy `ordered` a získame uvedený program 10.

Zdrojový kód 10: Program s direktívou ordered

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

int main(int argc, char* argv[])
{
  int i, n = 10;

  #pragma omp parallel for ordered default(none) shared(n)
    private(i)
  for(i = 0; i < n; i++) {
    printf("Vlakno %d z %d iteracia %d\n",
           omp_get_thread_num(), omp_get_num_threads(), i);

    #pragma omp ordered
    { //začiatok bloku ordered
      //príkazy budú vykonané v poradí iterácií cyklu
      printf("Iteracia v poradi %d\n", i);
    } //koniec bloku ordered
  }

  return 0;
}
```

Direktíva atomic

`#pragma omp atomic [ read | write | update | capture ]`

Pomocou tejto direktívy je možné zabezpečiť prístup viacerých vlákien k zdieľanej premennej tak, aby boli operácie čítania, zmeny, zápisu vykonané atomicky. Tým sa odstráni problém kritickej oblasti pri manipulácií s hodnotou uloženou v zdieľanej premennej. Jej význam je podobný ako význam direktívy `critical`, avšak je v porovnaní s direktívou `critical` efektívnejšia obzvlášť v prípadoch, ak procesor podporuje nerozdeliteľné inštrukcie pre manipuláciu s údajmi uloženými v pamäti. Naopak, na rozdiel od direktívy `critical`, má značné obmedzenia v tom, s akými výrazmi a blokmi programu je ju možné

použiť. Prehľad možných výrazov pre všetky klauzuly je uvedený v tabuľke 2.

Tabuľka 2: Zoznam povolených výrazov pre klauzulu atomic

Klauzula	Výraz	
read	v = x;	
write	x = expr;	

update
(alebo neuvedená)
	x++; x–;
	x binop= expr;
	x = expr binop x;
		++x; –x;
		x = binop + expr;

capture	v = x++; v = x–;
	v = x binop= expr;
	v = x = expr binop x;
		v = ++x; v = –xx;
		v = x = x binop expr;
		v = x = x binop expr;

Blok programu označený direktívou `atomic` môže byť v nasledujúcej forme:

```
{v = x; x binop= expr;}
{x binop= expr; v = x;}
{v = x; x = x binop expr;}
{v = x; x = expr binop x;}
{x = x binop expr; v = x;}
{x = expr binop x; v = x;}
{v = x; x = expr;}
{v = x; x++;}
{v = x; ++x;}
{++x; v = x;}
{x++; v = x;}
{v = x; x--;}
{v = x; --x;}
{--x; v = x;}
{x--; v = x;}
```

pričom operácia `binop` môže byť operáciou: `+`, `-`, `*`, `/`, `&`, `^`, `|`, `<<`, `>>`. `x` a `v` sú skalárne premenné a `expr` je výraz so skalárnou hodnotou. Pre názornosť uvádzame ukážku použitia v časti zdrojového kódu programu 11.

Zdrojový kód 11: Program s direktívou atomic

```
1 ...
2 #pragma omp atomic update
3 //operácia pripočítania 1 bude vykonaná atomicky
4 counter = counter + 1;
5 ...
```

Direktíva flush

```
#pragma omp flush [ zoznam ]
```

Klauzula umožní, aby boli vykonané zmeny hodnoty zdieľanej premennej jedným vláknom okamžite viditeľné pre ostatné vlákna.

Pri používaní zdieľaných premenných v paralelnom OpenMP programe sa domnievame, že hodnoty v nich uložené sú dostupné pre všetky vlákna v rámci paralelnej oblasti. Tento predpoklad však nemusí byť správny obzvlášť pri používaní moderných architektúr paralelných systémov pre vysokovýkonné počítanie. Tieto architektúry z dôvodu zvýšenia rýchlosti výpočtov používajú vyrovnávaciu pamäť. Pri zmene hodnoty zdieľanej premennej vláknom táto nie je okamžite zapísaná, a teda ani viditeľná pre ostatné vlákna, do hlavnej pamäte, ale iba do vyrovnávacej pamäte. Tomuto problému hovoríme **konzistentnosť údajov**.

Aby bola zachovaná konzistencia údajov v paralelnom OpenMP programe, používajú sa implicitné operácie pre vyprázdnenie a zápis bufra.

Tieto sa používajú pri synchronizácií vlákien, ako napríklad pri bariére, pri vstupe a výstupe z paralelnej, kritickej a zoradenej oblasti, pri výstupe oblasti rozdeľovania práce (pokiaľ nie je použitá klauzula `nowait`) a tesne pred a po každom bode plánovania podúloh. Nie je použitá pri vstupe do oblasti rozdeľovania práce a pri vstupe a výstupe oblasti master.

4 Meranie času v OpenMP programe

V podkapitole **??** sme sa zmienili o niekoľkých spôsoboch merania času vykonávania celého programu alebo určitej jeho časti. Knižnica OpenMP v sebe tiež obsahuje funkcie, ktoré je možné na tento účel použiť. V rámci OpenMP je na meranie času určená funkcia

```
double omp_get_wtime()
```

ktorá vracia ako návratovú hodnotu reálne číslo predstavujúce čas v sekundách, ktorý uplynul od určitého časového bodu v minulosti. Počas vykonávania procesov je garantované, že sa tento časový bod nezmení.

Čas vykonávania programu alebo jeho časti potom môžme vypočítať ako rozdiel dvoch hodnôt získaných volaním funkcie hneď tesne pred a po skončení sledovanej časti programu. Uvedený čas v sekundách predstavuje tzv. *wall clock* čas, čo je skutočný čas vykonávania programu.

Ďalšou užitočnou funkciou pri meraní času je:

```
double omp_get_wtick()
```

pomocou ktorej je možné určiť presnosť merania času. Funkcia vracia ako návratovú hodnotu reálne číslo, ktoré udáva čas v sekundách medzi dvoma tikmi hodín. Napríklad, ak by systém meral čas s presnosťou na $1\mu s$, tak funkcia by vrátila hodnotu 10^{-6}, čo odpovedá počtu sekúnd medzi dvoma tikmi.

Riešená úloha

Následujúci program 12 nám poslúži ako ukážka paralelného programu s použitím rozhrania OpenMP. Jedná sa o triediaci algoritmus, ktorého vstupom je náhodne usporiadané pole reálnych čísel.

Môžme si všimnúť, že paralelná oblasť v sebe obsahuje niekoľko cyklov `for`. Cykly na riadkoch 18 a 29 nie sú obzvlášť zaujímavé. Všimnime si však dva vnorené cykly na riadkoch 22 a 23. Tieto cykly sú vykonávané paralelne spoločne, čo sme zadali pomocou klauzuly `collapse(2)`.

Ďalšia dôležitá vec, ktorú netreba prehliadnuť je použitie direktívy `critical`, pomocou ktorej je ošetrený prístup k zdieľanej premennej

`idx`. Táto direktíva zabezpečí, že zmenu hodnoty tejto premennej môže súčasne vykonávať iba jedno vlákno.

Na riadku 37 a 39 si môžeme všimnúť použitie klauzuly `ordered`, pomocou ktorej sme zabezpečili to, aby prvky z poľa boli vypísané v správnom poradí. Toto riešenie sa môže javiť ako komplikované a nie príliš efektívne, výhodnejšie by bolo hodnoty z poľa vypísať až po skončení paralelnej oblasti, avšak nám išlo o demonštratívnu ukážku použitia klauzuly. Podobne je tomu aj pri použití direktívy `single` na riadku 42, ktorá zabezpečí, aby sa znak pre nový riadok vypísal iba v jednom vlákne, čiže sa zobrazí iba raz.

5 Efektívnosť paralelného OpenMP programu

Hoci sa rozhranie OpenMP môže javiť ako jednoduchý nástroj pre písanie paralelných programov, je dobré mať na pamäti problémy, ktoré môžu nastať pri vytváraní paralelných programov so zdieľanou pamäťou, a ako sa s nimi vysporiadať.

Podobne ako pri MPI, aj tu nám môžu informácie získané na základe profilovania programu poskytnúť cenné informácie o možných slabých miestach paralelného programu. Takéto profilovanie je možné vykonať pre každé vlákno samostatne, podobne ako pri sériovom programe. Druhá možnosť je použiť pokročilé nástroje, ako napríklad Intel Vtune Profiler [9], pomocou ktorého je ľahšie detegovať nevyváženosť rozvrhovania podúloh, serializované časti programu alebo aj čas réžie paralelizácie cyklu `for`.

Medzi typické problémy, s ktorými sa môžme stretnúť pri písaní OpenMP paralelných programov patria vykonávanie sériovej časti programu a nevyvážené rozdeľovanie záťaže. Komunikácia v porovnaní s MPI nie je problematická až do takej miery, pretože čas prístupu k údajom v zdieľanej pamäti je podstatne kratší ako pri komunikácii medzi výpočtovými uzlami. Avšak aj tu je potrebné mať na pamäti problematiku NUMA architektúry [10].

Zdrojový kód 12: OpenMP program na triedenie poľa

```
#include <stdio.h>
#ifdef _OPENMP
  #include <omp.h>
#endif
#define MAX 10

int main(int argc, char* argv[])
{
  float b[max], a[MAX] = {1.4, 1.2, 1.1, 1.1, 0.47, 0.99,
    7.86, 2.3, 9.0, 6.7};
  int idx[MAX], i, j;
  double start, stop;

  #pragma omp parallel
  { //začiatok paralelnej oblasti
#ifdef _OPENMP
    start = omp_get_wtime();    //začiatok merania času
#endif
    #pragma omp for //paralelné vykonávanie iterácií cyklu
    for(i = 0; i < MAX; i++) idx[i] = 0;
    //paralelne vykonávané iterácie 2 vnorených cyklov
    #pragma omp for collapse(2) private(i, j)
    for(i = 0; i < MAX; i++)
      for(j = 0; j < MAX; j++)
        if((a[i] > a[j]) || ((a[i] == a[j]) && (i > j)))
      //v kritickej oblasti môže byť len 1 vlákno
      #pragma omp critical
      idx[i]++;
    //paralelne vykonávané iterácie cyklu
    #pragma omp for
    for(i = 0; i < MAX; i++)
      b[idx[i]] = a[i];
#ifdef _OPENMP
    //koniec merania času
    stop = omp_get_wtime();
#endif
    //paralelne vykonávané iterácie cyklu v poradí iterácií
    #pragma omp for ordered
    for(i = 0; i < MAX; i++)
      #pragma omp ordered
      printf("%6.2f", b[i]);
    //blok single vykoná len 1 vlákno
    #pragma omp single
    printf("\n");
  } //koniec paralelnej oblasti
#ifdef _OPENMP
  printf("Cas: %.9f s\nPresnost: %e s\n", stop - start,
    omp_get_wtick());
#endif

  return 0;
}
```

Optimalizácia paralelného programu

Vytváranie a ukončovanie paralelnej oblasti je vždy spojené s určitou réžiou, a teda potrebným časom navyše v porovnaní so sériovým programom. Musia sa vytvoriť alebo zobudiť vlákna, určiť počet vlákien v skupine, v prípade rozvrhovania určiť, ktoré podúlohy bude vykonávať ktoré vlákno a taktiež vykonanie synchronizácie implicitnou bariérou na konci paralelnej oblasti. Tento čas réžie je možné znížiť aplikovaním nasledujúcich odporúčaní:

- program vykonávať sériovo, pokiaľ sa paralelizácia neoplatí,
- vyhýbať sa implicitným bariéram na konci blokov pomocou klauzuly `nowait`, ak je to možné,
- pokúsiť sa znížiť počet paralelných oblastí (paralelizáciou najvonkajšieho z vnorených cyklov, inak bude potrebné opätovne uspávať a zobúdzať vlákna),
- vyhýbať sa nesprávnemu "triviálnemu"vyvažovaniu záťaže (počet podúloh alebo blokov iterácií cyklu by mal byť väčší než počet použitých vlákien, inak niektoré vlákna nebudú vykonávať žiadnu podúlohu),
- vyhýbať sa plánovaniu `dynamic` a `guided` typu pokiaľ máme veľké množstvo malých úloh, pretože takéto rozvrhovanie vyžaduje ďalší čas réžie,
- vyhýbať sa prístupu k zdieľaným prostriedkom pomocou definovania kritickej oblasti, pretože to vedie k sériovému vykonávaniu tejto časti programu (je možné vytvoriť podstrom zdieľaných prostriedkov a neblokovať prístup k celému prostriedku, napríklad maticu rozdeliť na riadky a pomocou kritickej oblasti riadiť prístup k jej riadkom namiesto celej matice),
- predchádzať nesprávnemu zdieľaniu z dôvodu použitia vyrovnávacej pamäte.

Literatúra

[1] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997 (cited on page 1).

[2] OpenMP ARB. *OpenMP Application Programming Interface*. https://www.openmp.org/spec-html/5.0/openmp.html. 2018 (cited on pages 1, 2).

[3] Peter S. Pacheco. *An Introduction to Parallel Programming*. MA USA: Morgan Kaufmann, 2011 (cited on page 1).

[4] OpenMP ARB. *OpenMP 5.0 API Syntax Reference Guide*. https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-0519-web.pdf. 2018 (cited on page 2).

[5] Gerassimos Barlas. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014 (cited on page 3).

[6] IBM. *Writing reentrant and threadsafe code*. https://www.ibm.com/support/knowledgecenter/ssw_aix_71/generalprogramming/writing_reentrant_thread_safe_code.html. 2019 (cited on page 3).

[7] Zbigniew J. Czech. *Introduction to parallel computing*. UK: Cambridge University Press, 2016 (cited on pages 6, 18, 19).

[8] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001 (cited on page 7).

[9] *Intel Vtune Profiler*. https://software.intel.com/en-us/vtune. 2019 (cited on page 25).

[10] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010 (cited on page 25).