

Paralelní programování pro vícejádrové stroje s použitím OpenMP

Cvičení 3

B4B36PDV – Paralelní a distribuované výpočty
FEL ČVUT

Minulé cvičení:

“Vlákna a jejich synchronizace v C++ 11...”

Minulé cvičení:

“Vlákna a jejich synchronizace v C++ 11...”

Programování vícevláknových aplikací ručně může být dřina. Proč znovu objevovat kolo, když můžeme použít hotové řešení?

Minulé cvičení:

“Vlákna a jejich synchronizace v C++ 11...”

Programování vícevláknových aplikací ručně může být dřina. Proč znovu objevovat kolo, když můžeme použít hotové řešení?

Dnešní cvičení:

OpenMP

- Opakování z minulého cvičení
- Úvod do OpenMP
- Paralelní bloky se sdílenou pamětí a synchronizace
- Redukce s OpenMP
- Rozvrhování výpočtu v OpenMP

- Zadání druhé domácí úlohy

Opakování z minulého cvičení

<http://goo.gl/a6BEMb>

Co je OpenMP?

- API pro psání vícevláknových aplikací se sdílenou pamětí
- Sada directiv, proměnných prostředí a rutin pro kompilátor a programátory
- Ulehčuje psaní vícevláknových aplikací v C/C++ a Fortran na většině platform s podporou většiny instrukčních sad a operačních systémů

- API pro psání vícevláknových aplikací se sdílenou pamětí
- Sada directiv, proměnných prostředí a rutin pro kompilátor a programátory
- Ulehčuje psaní vícevláknových aplikací v C/C++ a Fortran na většině platform s podporou většiny instrukčních sad a operačních systémů

Jako základní referenční příručku můžete použít

<https://msdn.microsoft.com/en-us/library/tt15eb9t.aspx>.

Otestujte si své prostředí

`omp_get_num_procs()`

Počet procesorů, které OpenMP využívá v době volání funkce

`omp_get_num_threads()`

Počet vláken, které OpenMP využívá v době volání funkce

`omp_get_max_threads()`

Maximální počet vláken, které OpenMP může využít

`omp_in_parallel()`

Vrací nenulovou hodnotu, pokud jsme uvnitř paralelního bloku

`omp_get_nested()`

Vrací nenulu, pokud je povoleno vnořování paralelních bloků

Otestujte si své prostředí

`omp_get_num_procs()`

Počet procesorů, které OpenMP využívá v době volání funkce

`omp_get_num_threads()`

Počet vláken, které OpenMP využívá v době volání funkce

`omp_get_max_threads()`

Maximální počet vláken, které OpenMP může využít

`omp_in_parallel()`

Vrací nenulovou hodnotu, pokud jsme uvnitř paralelního bloku

`omp_get_nested()`

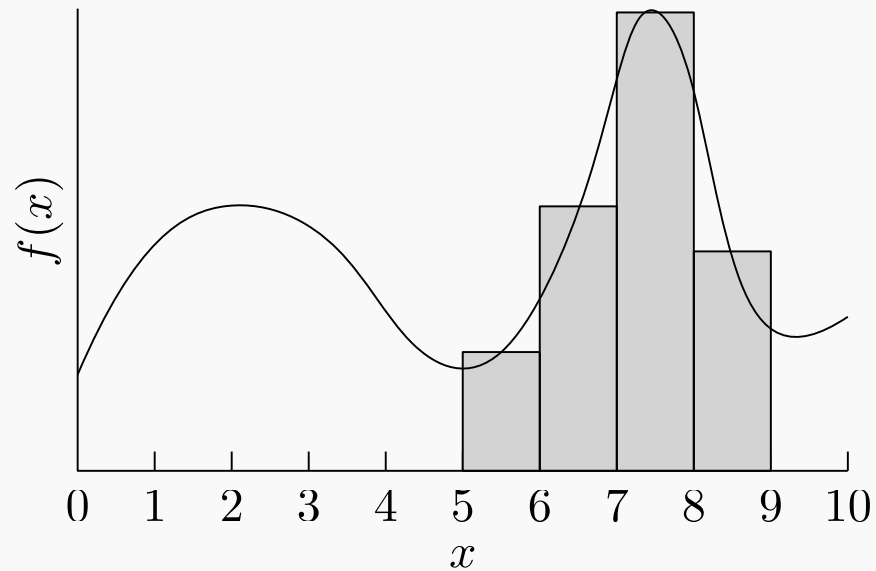
Vrací nenulu, pokud je povoleno vnořování paralelních bloků

Otestujte si své prostředí

Otestujte na souboru `1openmp_test.cpp`.

Cvičení: Numerická integrace

$$\int_5^9 f(x) dx \sim \text{plocha } \blacksquare$$



```
double integrate(  
    std::function<double(double)> integrand,  
    double a, double step_size, int step_count);
```

Doimplementujte sekvenční verzi numerické integrace

Doimplementujte tělo metody `integrate_sequential` v souboru `2integrate.cpp`. Použijte obdélníkovou metodu, kdy jako “výšku” obdélníku použijete hodnotu funkce uprostřed intervalu.

`integrand`

Funkce, kterou máte za úkol numericky zintegrovat

`a`

Dolní mez integrálu

`step_size`

Velikost kroku (šířka obdélníku)

`step_count`

Počet kroků (horní mez je $a + \text{step_size} * \text{step_count}$)

Doimplementujte sekvenční verzi numerické integrace

Doimplementujte tělo metody `integrate_sequential` v souboru `2integrate.cpp`. Použijte obdélníkovou metodu, kdy jako “výšku” obdélníku použijete hodnotu funkce uprostřed intervalu.

`integrand`

Funkce, kterou máte za úkol numericky zintegrovat

`a`

Dolní mez integrálu

`step_size`

Velikost kroku (šířka obdélníku)

`step_count`

Počet kroků (horní mez je $a + \text{step_size} * \text{step_count}$)

Jaké problémy budeme mít, pokud budeme chtít tento sekvenční kód paralelizovat?

Alternativy k mutexům a atomickým proměným v OpenMP

#pragma omp parallel

```
int num_threads = 0;
#pragma omp parallel
{
    // Zde jsme vytvorili tym vlaken, ktera vykonavaji nasledujici kod
    num_threads += 1;
}
```

#pragma omp parallel

```
int num_threads = 0;
#pragma omp parallel
{
    // Zde jsme vytvorili tym vlaken, ktera vykonavaji nasledujici kod
    num_threads += 1;
}
```

Jaký bude výsledek?

#pragma omp critical (mutex')

```
int num_threads = 0;
#pragma omp parallel
{
    // Zde muze byt vice vlaken soucasne ...

    #pragma omp critical
    {
        // .. ,ale inkrementaci provadi vzdy maximalne jedno vlakno
        num_threads += 1;
    }

    // Zde opet muze byt vice vlaken soucasne
}
```

Doimplementujte metodu `integrate_omp_critical`

Doimplementujte metodu `integrate_omp_critical` v `2integrate.cpp`. Využijte k tomu `#pragma omp parallel` a `#pragma omp critical`.

Tip: Po spuštění vláken v bloku `#pragma omp parallel` si můžete napočítat rozsahy indexů, které jednotlivá vlákna budou zpracovávat. Pro zjištění indexu aktuálního vlákna použijte metodu `omp_get_thread_num()`. Zjistit celkový počet vláken lze pomocí `omp_get_num_threads()`.

Na minulém cvičení jsme si ukázali, že mutexy mohou být pomalé.

Opravdu pomalé.

Jednoduché operace nad jednou proměnnou lze řešit *hardwarovým* zámekem – provedením atomické operace

```
int num_threads = 0;
#pragma omp parallel
{
    #pragma omp atomic
    num_threads += 1;
}
```

Ne všechny operace lze provést atomicky!

Typicky pouze: $x++$, $x--$, $++x$, $--x$ a $x OP=$
expr, kde

$OP \in \{+, -, *, /, \&, ^, |, \ll, \gg\}$

Pokud kompilátor nemá k dispozici danou atomickou operaci, použije záložní plán: mutex.

Doimplementujte metodu `integrate_omp_atomic`

Doimplementujte metodu `integrate_omp_atomic` v `2integrate.cpp`.
Místo kritické sekce využijte `#pragma omp atomic`. Jakého zrychlení touto úpravou dosáhneme?

Redukce v OpenMP

To samé lze ale udělat elegantněji a efektivněji:

```
int num_threads = 0;
#pragma omp parallel reduction(+:num_threads)
{
    num_threads += 1;
}
```

OpenMP pak zajistí, že se částečné výsledky *lokálních* proměnných `num_threads` po konci bloku posčítají

Následující “operátory” jsou podporované (OpenMP verze 3+):

- Aritmetické: +, *, -, max, min
- Logické: &, &&, |, ||, ^

Doimplementujte metodu `integrate_omp_reduction`

Doimplementujte tělo metody `integrate_omp_reduction` v souboru `2integrate.cpp`. Nahraďte `#pragma omp atomic` redukcí.

#pragma omp parallel for

Kód s redukcí lze napsat ještě jednodušeji.

Rozsahy pro vlákna si nemusíme počítat ručně a můžeme práci nechat na OpenMP:

```
double acc = 0.0;

#pragma omp parallel for reduction(+:acc) //schedule(static)
for(int i = 0 ; i < step_count ; i++) {
    const double cx = a + (2*i + 1.0)*step_size/2;
    acc += integrand(cx)*step_size;
}
return acc;
```

Proč při integraci funkce $f(x) = x$
dosahujeme většího zrychlení?

Proč při integraci funkce $f(x) = x$
dosahujeme většího zrychlení?

Výpočet $f(x) = x$ trvá konstantní dobu a práce je tak mezi vlákna rozdělena rovnoměrně.

To neplatí o funkci $f(x) = \frac{1}{\exp(x)}$, kterou aproximujeme numerickou integrací s proměnlivým počtem kroků.

Doimplementujte metodu `integrate_omp_for_dynamic`

Doimplementujte tělo metody `integrate_omp_for_dynamic`.
Statické rozvrhování `schedule(static)` nahradte dynamickým
`schedule(dynamic)`. Jaký má tato volba dopad na rychlost numerické
integrace $f(x) = x$ a $f(x) = \frac{1}{\exp(x)}$?

#pragma omp parallel for schedule

Obecná syntaxe (možno použít i další parametry jako např. `reduction`):

```
#pragma omp parallel for schedule(type[,chunk_size])
```

#pragma omp parallel for schedule

Obecná syntaxe (možno použít i další parametry jako např. `reduction`):

```
#pragma omp parallel for schedule(type[,chunk_size])
```

`chunk_size` udává minimální velikost bloku, se kterým se plánuje, např:

```
#pragma omp parallel for schedule(dynamic,16)
```

zajistí, že si vlákno po dokončení práce na aktuálním bloku dat řekne o další blok o 16 prvcích.

#pragma omp parallel for schedule

Obecná syntaxe (možno použít i další parametry jako např. `reduction`):

```
#pragma omp parallel for schedule(type[, chunk_size])
```

`chunk_size` udává minimální velikost bloku, se kterým se plánuje, např:

```
#pragma omp parallel for schedule(dynamic, 16)
```

zajistí, že si vlákno po dokončení práce na aktuálním bloku řekne o další blok o 16 prvcích.

dynamic

Vlákna si *dynamicky* alokují bloky, které mají počítat

guided

Dynamické plánování, kde se velikost bloků v průběhu výpočtu zmenšuje

static

Každé vlákno má svůj blok přiřazený napevno (když skončí dříve, musí čekat)

runtime

Rozhodnuto za běhu na základě nastavení prostředí
(`export OMP_SCHEDULE="dynamic, 100"`)

Zadání druhé domácí úlohy

V 2. domácí úloze si budete moci vyzkoušet, že úspěšnost různých způsobů paralelizace **závisí** do značné míry na **vstupních datech**.

Na vstupu dostanete vektor složený z vektorů náhodně generovaných čísel.

Vaším úkolem je čísla v každém vektoru **sečíst** a tento součet vložit do vektoru s řešením na index odpovídající pořadí vektoru, který jste sčítali.

Doimplementujte metody v `SumsOfVectors.cpp` a zajistěte, že

- Výpočet sum je paralelní a každá metoda vrátí korektní výsledky
- Metody využívají požadované způsoby paralelizace

Doimplementujte metody v `SumsOfVectors.cpp` a zajistěte, že

- Výpočet sum je paralelní a každá metoda vrátí korektní výsledky
- Metody využívají požadované způsoby paralelizace

Za spravné výsledky na každé ze **čtyř** datových sad dostanete 2b.