Programování architektur se sdílenou pamětí - OpenMP

Architektury se sdílenou pamětí - vlákna

Programy pro architektury se sdílenou pamětí spouštějí několik souběžných vláken.
Standardy:

- POSIX - standard pro manipulaci s vlákny
- OpenMP - standard pro podporu vláken na úrovni překladače

OpenMP je podporováno v icc/icpc, experimentálně v gcc/g++.

Základní direktiva OpenMP

OpenMP využívá direktivy preprocesoru.

`#pragma omp` *directive* [*clause list*]

Základní direktiva OpenMP

OpenMP využívá direktivy preprocesoru.

`#pragma omp` *directive* `[`*clause list*`]`

Direktiva `parallel` způsobí, že následující blok instrukcí bude zpracován více vlákny.

```
#pragma omp parallel [clouse list]
{
...
}
```

Direktiva `parallel`

Pomocí `[`*clouse list*`]` lze udat:

- podmínku (pouze jednu) paralelizace: `if(` ... `)`
- počet vláken: `num_threads(` *integer expression* `)`
- zacházení s daty
  - `private(` *variable list* `)`
    Určuje lokální proměnné = každé vlákno má svou vlastní kopii.
  - `firstprivate(` *variable list* `)`
    Stejně jako `private`, ale u všech kopii se nastaví hodnota, kterou měla proměnná před rozvětvením běhu programu na vlákna.
  - `shared(` *variable list* `)`
    Tyto proměnné budou sdílené mezi vlákny.
  - `reduction(` *operator*: *variable list* `)`
    Dané proměnné budou mít lokální kopie a nakonec se provede redukce pomocí asociativní operace: +,*,&, |,&&, ||.

Příklady

```
#pragma omp parallel if( is_parallel == true )
num_threads(8) private(a) firstprivate(b)
{
...
}
```

Příklady

```
#pragma omp parallel if( is_parallel == true )
num_threads(8) private(a) firstprivate(b)
{
...
}
```

```
#pragma omp parallel if( size > 1000 )
num_threads( MIN( size/1000+1,8) )
reduction(+:sum)
{
...
}
```

Pomocné funkce (1)

Funkce pro identifikaci vláken:

`omp_get_num_threads()` - vrací počet vláken

`omp_get_thread_num()` - vrací celočíselný identifikátor vlákna

Určení souběžných úloh

Po spuštění více vláken je nutné říci, co mají jednotlivá vlákna provádět.

- všechna vlákna provádějí stejnou úlohu = dělí se o `for` cyklus
- každé vlákno provádí jinou úlohu = zpracovávají sekce (`sections`) různého kódu

Paralelizace `for` cyklů

`#pragma omp for[`*clause list*`]`

Klausule pro ošetření proměnných:

- `private`
- `firstprivate`
- `reduction`
- `lastprivate` = hodnota proměnné je nastavena v posledním průběhu `for` cyklu

Paralelizace `for` cyklů

Klasule pro rozdělení iterací mezi vlákny - `schedule`

`schedule(` *schedulling_class[,parameter]*)

Třídy:

- `static`
- `dynamic`
- `guided`
- `runtime`

Statické přidělování iterací

```
schedule( static[, chunk-size] )
```

Každé vlákno postupně dostává stejný počet iterací daný pomocí `chunk-size`.

Není-li `chunk-size` uvedeno, jsou všechny iterace rozděleny na `n` stejných částí, kde `n` je počet vláken.

Statické přidělování iterací

```
schedule( static[, chunk-size] )
```

Každé vlákno postupně dostává stejný počet iterací daný pomocí `chunk-size`.

Není-li `chunk-size` uvedeno, jsou všechny iterace rozděleny na `n` stejných částí, kde `n` je počet vláken.

Příklad: 128 iterací, 4 vlákna
`schedule( static )` = 4 $\times$ 32 iterací
`schedule( static, 16 )` = 8 $\times$ 16 iterací

Dynamické přidělování iterací

```
schedule( dynamic[, chunk-size] )
```

Funguje podobně jako dynamické přidělovaní iterací. Nové iterace jsou ale přidány vláknu, které skončí svou praci jako první. Některá vlákna tak mohou provést více iterací, než ostatní.

Řízené přidělování iterací

```
schedule( guided[, chunk-size] )
```

Příklad: 100 iterací rozdělených po 5 $\Rightarrow$ 20 kousků pro 16 vláken.

`guided` s každým přidělením nových iterací exponenciálně zmenšuje *chunk size*. `chunk-size` udává dolní mez pro počet přidělených iterací.

Přidělování iterací určené za chodu programu

`schedule(runtime)`

Podle systemové proměnné `OMP_SCHEDULE` se určí, zda se má použít `static`, `dynamic` nebo `guided`.

Vhodné při vývoji programu pro zjištění nejvhodnější volby.

Synchronizace mezi jednotlivými `for` cykly

Standardně se nezačíná nový cyklus, dokud všechna vlákna neskončila práci na předchozím cyklu - bariéra mezi cykly. Pokud to není nutné, lze použít klauzuli `nowait`.

```
#pragma omp parallel
{
  #pragma omp for nowait
    for( i = 0; i < nmax; i ++ ){ ... }
  #pragma omp for
    for( i = 0; i < mmax; i ++ ){ ... }
}
```

Zpracování různých úloh každým vláknem

Provádí se pomocí direktivy `omp sections`:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
         { TaskA(); }
      #pragma omp section
         { TaskB(); }
   }
}
```

Zkrácené psaní direktiv

Lze psát:

```
#pragma omp parallel for shared(n)
```

```
#pragma omp parallel sections
```

Vložení direktivy `parallel`

Musí být nastavena systemová proměnná
`OMP_NESTED = TRUE`.

```
#pragma omp parallel for ...
  for( i = 0; i < N; i ++ )
  #pragma omp parallel for ...
    for( j = 0; j < N; j ++ )
    #pargma omp paralle for ...
      for( k = 0; k < N; k ++ )
      #pragma omp parallel for ...
```

Sychronizace

Bariéra = žádné vlákno nesmí pokračovat, dokud všechna ostatní nedosáhnou bariéry.

```
#pragma omp barriere
```

Bloky pro jedno vlákno

```
#pragma omp single { ... }
```

Tento blok bude zpracován jen jedním (prvním) vláknem. Pokud není uvedeno `nowait`, ostatní vlákna čekají na konci bloku.

```
#pragma omp master { ... }
```

Tento blok bude zpracován jen vláknem s ID = 0, ostatní vlákna nečekají.

Kritické bloky

Kritické bloky obsahují kód, který může současně provádět jen jedno vlákno.

```
#pragma omp critical [(name)]
```

Příklad: Částečné úlohy pro jednotlivá vlákna lze distribuovat pomocí centrální sktruktury (fronty). Přístup k ní pak může mít v daný okamžik jen jedno vlákno.

Kritické bloky - příklad

```
#pragma omp parallel sections
{
  #pragma omp section
  { /* producer thread */
    task = producer_task();
    #pragma omp critical (task_queu)
    { insert_into_queu( task ); }
  }
  #pragma omp parallel section
  { /* consumer thread */
    #pragma omp critical (task_queu)
    { task = extract_from_queu(); }
  }
}
```

Ošetření dat v OpenMP

- všechny proměnné, které používá jen jedno vlákno, by měly být označené jako `private`

Ošetření dat v OpenMP

- všechny proměnné, které používá jen jedno vlákno, by měly být označené jako `private`
- při častém čtení proměnné nastavené již dříve v programu je vhodné označit ji jako `firstprivate`

Ošetření dat v OpenMP

- všechny proměnné, které používá jen jedno vlákno, by měly být označené jako `private`
- při častém čtení proměnné nastavené již dříve v programu je vhodné označit ji jako `firstprivate`
- přístup všech vláken ke stejným datům je vhodné provádět pomocí lokálních proměnných (vzhledem k vláknu) a nakonec použít redukci

Ošetření dat v OpenMP

- všechny proměnné, které používá jen jedno vlákno, by měly být označené jako `private`
- při častém čtení proměnné nastavené již dříve v programu je vhodné označit ji jako `firstprivate`
- přístup všech vláken ke stejným datům je vhodné provádět pomocí lokálních proměnných (vzhledem k vláknu) a nakonec použít redukci
- přistupují-li vlákna k různým částem velkého bloku dat, je dobré je předem explicitně rozdělit

Ošetření dat v OpenMP

- všechny proměnné, které používá jen jedno vlákno, by měly být označené jako `private`
- při častém čtení proměnné nastavené již dříve v programu je vhodné označit ji jako `firstprivate`
- přístup všech vláken ke stejným datům je vhodné provádět pomocí lokálních proměnných (vzhledem k vláknu) a nakonec použít redukci
- přistupují-li vlákna k různým částem velkého bloku dat, je dobré je předem explicitně rozdělit
- zbývající typ proměnných již musí být zřejmě sdílen

Funkce knihovny OpenMP

Je nutné použít hlavičkový soubor

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

```
void omp_set_num_threads( int num_threads);
int omp_get_num_threads();
int omp_get_thread_num();
int omp_get_num_procs();
int omp_in_paralle();
```

Systemové proměnné

- `OMP_NUM_THREADS`
  `setenv OMP_NUM_THREADS 8`

Systemové proměnné

- `OMP_NUM_THREADS`
  `setenv OMP_NUM_THREADS 8`
- `OMP_DYNAMIC` - umožňuje použití funkci `omp_set_num_threads` nebo klausuli `num_threads`
  `setenv OMP_DYNAMIC "TRUE"`

Systemové proměnné

- `OMP_NUM_THREADS`
  `setenv OMP_NUM_THREADS 8`
- `OMP_DYNAMIC` - umožňuje použití funkci `omp_set_num_threads` nebo klausuli `num_threads`
  `setenv OMP_DYNAMIC "TRUE"`
- `OMP_NESTED`

Systemové proměnné

- `OMP_NUM_THREADS`
  `setenv OMP_NUM_THREADS 8`
- `OMP_DYNAMIC` - umožňuje použití funkci `omp_set_num_threads` nebo klausuli `num_threads`
  `setenv OMP_DYNAMIC "TRUE"`
- `OMP_NESTED`
- `OMP_SCHEDULE`
  `setenv OMP_SCHEDULE "static,4"`
  `setenv OMP_SCHEDULE "dynamic"`
  `setenv OMP_SCHEDULE "guided"`