Bendrų kintamųjų apsauga OpenMP

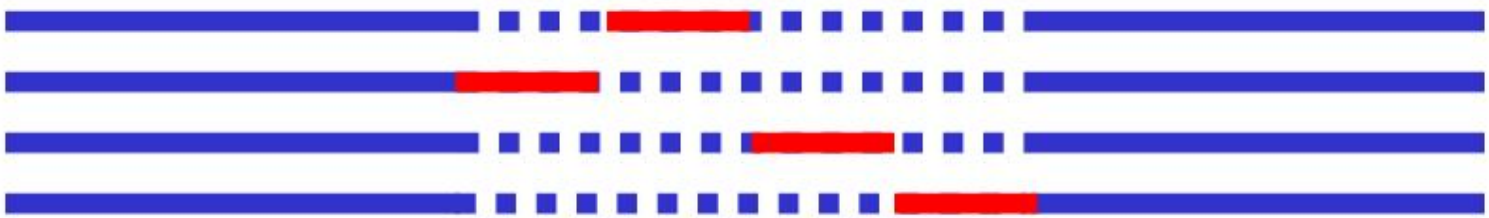
OpenMP kritinės sritys ir užraktai

#pragma omp critical

- #pragma omp critical [(name)]
  sakinių blokas

The directive identifies a section of code that must be executed by a single thread at a time.
A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name.

`#pragma omp critical` veikimas



KS apsauga su #pragma omp critical: pavyzdys

```
#pragma omp parallel shared(c, n)
{
  for (int i=0; i<n; i++) {
    #pragma omp critical
    {
      int a = c;
      a++; c = a;
    }
  }
}
```

Duomenų tipas `omp_lock_t`

A type that holds the status of a lock, whether the lock is available or if a thread owns a lock.

`void omp_init_lock(omp_lock_t *lock);`

The function provide the only means of initializing a lock. It initializes the lock associated with the parameter **`lock`** for use in subsequent calls.

The initial state is unlocked (that is, no thread owns the lock).

`void omp_destroy_lock(omp_lock_t *lock);`

The function ensures that the pointed to **`lock`** variable is uninitialized.

The argument to the function must point to an initialized **`lock`** variable that is locked.

`void omp_set_lock(omp_lock_t *lock);`

The function blocks the thread executing until the specified **`lock`** is available and then sets the lock. A simple lock is available if it is unlocked.

The argument to the **`omp_set_lock`** function must point to an initialized **`lock`** variable. Ownership of the **`lock`** is granted to the thread executing the function.

`void omp_unset_lock(omp_lock_t *lock);`

The function provides the means of releasing ownership of a **`lock`**. The argument of the function must point to an initialized **`lock`** variable owned by the thread executing the function.

The behaviour is undefined if the thread does not own that **`lock`**. The function releases the thread executing the function from ownership of the **`lock`**.

`int omp_test_lock(omp_lock_t *lock);`

The function attempt to set a lock but do not block execution of the thread. The argument must point to an initialized **`lock`** variable.

The function attempt to set a lock in the same manner as **`omp_set_lock`**, except that it do not block execution of the thread. The function returns non-zero if the lock is successfully set; otherwise, it returns zero.

KS apsauga su `lock`: pavyzdys

```
omp_lock_t *raktas;
raktas = new omp_lock_t;
omp_init_lock(raktas);
// ---- Lygiagretus kodas ----
#pragma omp parallel shared(c, cikluSk)
{
  for (int i=0; i<cikluSk; i++) {
    omp_set_lock(raktas);
      int a = c;
      a++; c = a;
    omp_unset_lock(raktas);
  }
}
omp_destroy_lock(raktas);
```

OpenMP bendrieji ir privatūs kintamieji

Bendrieji kintamieji

- Kompiuterio atmintyje yra tik viena bendrojo (*shared*) kintamojo kopija. Bendrasis kintamasis matomas kiekvienoje gijų rinkinio gijoje.
- Vienoje gijoje pakeista bendrojo kintamojo reikšmė gali būti matoma kitoje gijoje.
- Jei nenurodyta kitaip, visi programos kintamieji yra bendri visoms lygiagrečios srities gijoms.

Privatūs kintamieji

- Privatus (*private*) kintamasis turi savo kopijas kiekvienoje gijoje. Kiekviena kopija matoma tik vienoje gijoje.
- Vienoje gijoje pakeista privataus kintamojo reikšmė nematoma kitose gijose.
- Kintamieji yra privatūs trimis atvejais:
  1. lygiagrečiojo (*for*) ciklo indeksas yra privatus,
  2. lygiagrečios srities bloke paskelbti lokalūs kintamieji yra privatūs,
  3. visi kintamieji, išvardinti *pragma omp* direktyvoje kaip *private*, *firstprivate*, *lastprivate* arba *reduction*, yra privatūs.

Privatūs kintamieji: *private* parinktis

Parinktis nurodo, kad sukuriama po vieną kintamojo kopiją kiekvienai gijai; pradinė reikšmė – numatytoji to kintamojo tipo konstruktoriuje (gali būti ir neapibrėžta).

private parinkties poveikis

- Panaudojimo pavyzdys:

```
int c = 99, gijuSk = 3;
omp_set_num_threads(gijuSk);
#pragma omp parallel private(c)
{
  c = omp_get_thread_num();
}
```

- Rezultatas:

```
Pr.reikšmės:  c = 99, gijuSk = 3, gijosNr = 0
c = -64, gijoje gauta reikšmė:  c = 2
c = -1208621368, gijoje gauta reikšmė:  c = 0
c = -64, gijoje gauta reikšmė:  c = 1
Baigus gijų darbą:  c = 99
```

Privatūs kintamieji: *firstprivate* parinktis

Parinktis skiriasi nuo *private* tuo, kad į kiekvieną giją kopijuojama kintamojo reikšmė naudojant kopijos konstruktorių.

firstprivate parinkties poveikis

- Panaudojimo pavyzdys:

```
int c = 99, gijuSk = 3;
omp_set_num_threads(gijuSk);
#pragma omp parallel firstprivate(c)
{
  c = omp_get_thread_num();
}
```

- Rezultatas:

```
Pr.reikšmės:  c = 99, gijuSk = 3, gijosNr = 0
c = 99, gijoje gauta reikšmė:  c = 2
c = 99, gijoje gauta reikšmė:  c = 0
c = 99, gijoje gauta reikšmė:  c = 1
Baigus gijų darbą:  c = 99
```

Privatūs kintamieji: *lastprivate* parinktis

Parinktis skiriasi nuo *private* tuo, kad paskutinėje iteracijoje ar sekcijoje gauta reikšmė kopijos priskyrimo operatoriumi perduodama į pagrindinę giją.

lastprivate parinkties poveikis

- Panaudojimo pavyzdys:

```
int c = 99, gijuSk = 3;
omp_set_num_threads(gijuSk);
#pragma omp parallel lastprivate(c)
{
  c = omp_get_thread_num();
}
```

- Rezultatas:

```
Pr.reikšmės:  c = 99, gijuSk = 3, gijosNr = 0
c = -12088082, gijoje gauta reikšmė:  c = 2
c = -121032, gijoje gauta reikšmė:  c = 0
c = 32, gijoje gauta reikšmė:  c = 1
Baigus gijų darbą:  c = 1
```

Privatūs kintamieji: *reduction* parinktis

Parinktis skiriasi nuo *private* tuo, kad kartu su kintamuoju perduodamas ir operatorius;
reduction kintamasis turi būti skaliarinis kintamasis,
inicializacijos metu įgyja reikšmę, numatytą tam operatoriui;
bloko gale *reduction* operatorius pritaikomas visoms kopijoms ir pradinei kintamojo reikšmei.

reduction parinkties poveikis

- Panaudojimo pavyzdys:

```
int c = 5, gSk = 50, k = 3;
omp_set_num_threads(gSk);
#pragma omp parallel shared(k) reduction(+:c)
{
  for (int i=1; i<=k; i++)
    c += i;
}
```

- Rezultatas:

```
Gauta reikšmė:  c = 305
```

Kokie bus rezultatai?

```
int a = 10, b = 0, c = 0;
int A[] = {5, 4, 8, 20, 1, 2, 50, 7, 6, 3};
omp_set_num_threads(a);
#pragma omp parallel shared(A,c) reduction(+:b)
{
  int r = omp_get_thread_num();
  #pragma omp critical
  {
    if (A[r] > c) c = A[r];
    b = A[r];
  }
}
```

Kitos OpenMP galimybės

#pragma omp for

- #pragma omp for [clause[[,] clause] ...]
  <for_loop>

The directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

#pragma omp for pavyzdys

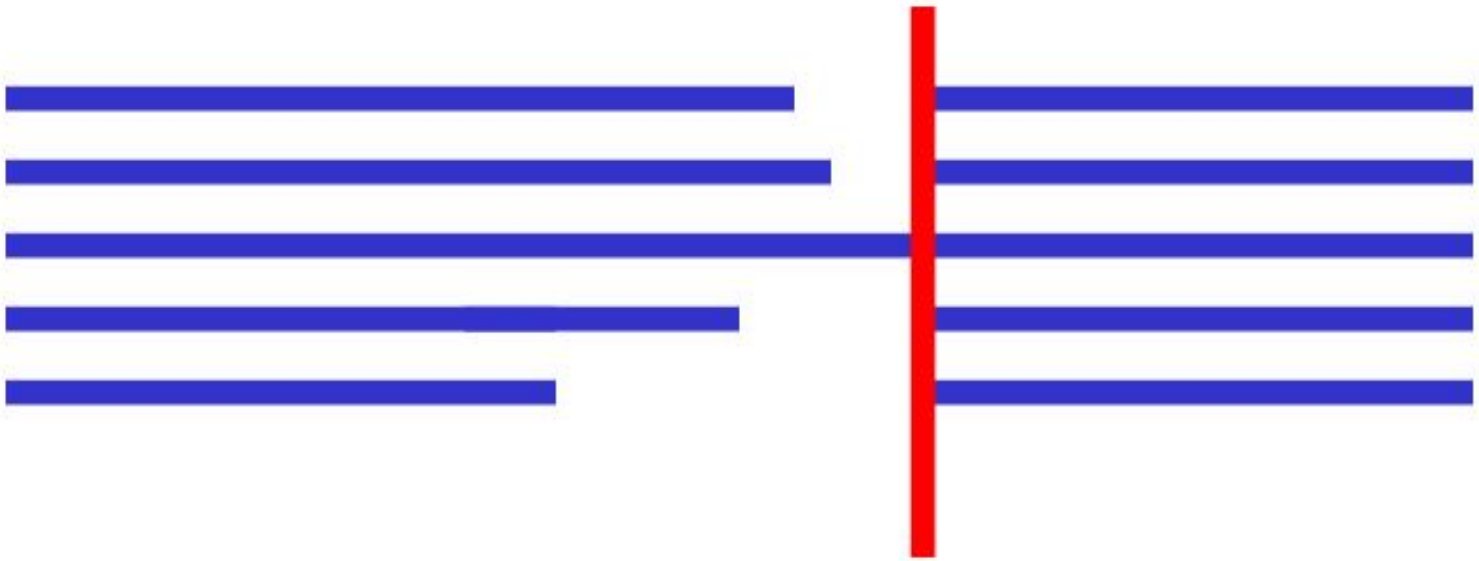
```
#pragma omp parallel shared(a,b) private(j)
{
  #pragma omp for
  for (j=0; j<N; j++)
    a[j] = a[j] + b[j];
}
```

#pragma omp barrier

- #pragma omp barrier

The directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point.
Statement execution past the omp barrier point then continues in parallel.

`#pragma omp barrier` veikimas



`#pragma omp barrier` pavyzdys

```
#pragma omp parallel sections
{
  #pragma omp section
  {
    thread_A_func();
  }
  #pragma omp section
  {
    thread_B_func();
  }
  #pragma omp barrier
  function_to_run_after_above_completes();
}
```

#pragma omp master

- #pragma omp master
  statement_block

Synchronization construct identifying a section of code that must be run only by the master thread.

`#pragma omp master` veikimas

Klausimai pakartojimui

1. Kaip realizuoti monitorių OpenMP priemonėmis?
2. Kuo pasižymi #pragma omp critical?
3. Kuo pasižymi omp_lock?
4. Kokie yra skirtumai tarp #pragma omp critical ir omp_lock?
5. Kokiu būdu įgyvendinama sąlyginė sinchronizacija OpenMP monitoriuje?
6. Kuo ypatingi bendrieji gijų kintamieji?
7. Kuo ypatingi privatūs gijų kintamieji?