Змістовний модуль:

РОЗРОБКА ПАРАЛЕЛЬНИХ АЛГОРИТМІВ І ПРОГРАМ

Розділ: Технології розробки паралельних програм

Лекція 11. ТЕХНОЛОГІЯ OPEN MP. ПРОДОВЖЕННЯ

ПИТАННЯ ДЛЯ ВИВЧЕННЯ

1. Додаткові параметри для parallel та for
2. Класи змінних в *C*++ програмах .
3. Класи змінних для OPEN MP
4. Засоби синхронізації потоків одного процесу для Windows
5. Засоби синхронізації OPEN MP. atomic
6. Засоби синхронізації OPEN MP. Критичні секції
7. Засоби синхронізації OPEN MP. Замки

ДОДАТКОВІ ПАРАМЕТРИ ДЛЯ ДИРЕКТИВИ FOR

parallel

***if*(Константний вираз);**

***num_threads*(Константний вираз);**

***reduction* (Операція: Список)**

private(Список змінних);

firstprivate(Список змінних);

default(shared | none);

shared(Список змінних);

copyin(Список змінних);

proc_bind(master | close | spread)

for

reduction(Операція: Список)

private(Список змінних)

firstprivate(Список змінних)

schedule(Спосіб[, Розмір])

collapse(n)

nowait

ordered

lastprivate(Список змінних)

ДОДАТКОВІ ПАРАМЕТРИ ДЛЯ ЦИКЛІВ. ПАРАМЕТРИ NOWAIT

При паралельному виконанні циклу автоматично виконується очікування завершення всіх ітерацій до переходу до оператора, що виконується після циклу. Якщо таке очікування не потрібно, його можна відключити. Для цього використовується параметр ***nowait***.

Приклад.

Хай треба обчислити

```
#pragma omp parallel for
for (int i = 0; i < n; ++i)
      Y[i] = f1(x[i]);
b = …
#pragma omp parallel for
for (int i = 0; i < n; ++i)
      Z[i] = f2(Y[i], b);
```

Чи можна для виконання 2 циклу не чекати виконання 1 циклу?

Так. По замовченню режим static, визначення ітерацій до виконання циклу!!!

ДОДАТКОВІ ПАРАМЕТРИ ДЛЯ ЦИКЛІВ. ПАРАМЕТР ORDERED

Параметр ordered

Якщо задано, кожний потік може бути виконано тільки в порядку черзі!!!

```
#pragma omp parallel for num_threads (4) ordered
    for (i = 0; i < sizeof (x) / sizeof (float); i++) {
        u[i] = f (x [i] * y [i]);
        #pragma omp ordered
        {
            printf ("thread # = %d\ti = %d\tu[i]= %g\\n",
                omp_get_thread_num (), i, u [i],);
        }
    }
```

ДОДАТКОВІ ПАРАМЕТРИ ДЛЯ ЦИКЛІВ. ПАРАМЕТР ORDERED

Параметр ordered
Якщо задано, кожний потік може бути виконано тільки в порядку черзі!!!

```
#pragma omp parallel for num_threads (4) ordered
    for (i = 0; i < sizeof (x) / sizeof (float); i++) {
        u[i] = x [i] * x [i];
        #pragma omp ordered
        {
            y [i ] = a * y [i + 1];
            printf ("thread # = %d\ti = %d\ty[i]= %g\ty[i + 1]= %g\n",
                omp_get_thread_num (), i, y [i], y [i + 1]);
        }
        z[i] = z [i] / 2;
        printf ("thread # = %d\ti = %d\tz[i]= %g\n",
        omp_get_thread_num (), i, z [i]);
    }
```

ВКЛАДЕННЯ ПАРАЛЕЛЬНИХ ДІЛЯНОК КОДУ

Код:

```
#pragma omp parallel num_threads (3)
    {
_tprintf (_T("Hello, world, thread num = %d\n"),
    omp_get_thread_num ());
    }
```

Очікуваний результат

Hello, world, thread num = 0

Hello, world, thread num = 1

Hello, world, thread num = 2

ВКЛАДЕННЯ ПАРАЛЕЛЬНИХ ДІЛЯНОК КОДУ

***//omp_set_nested (1*); 3 або 9 потоків!!!**

```
#pragma omp parallel num_threads (3)
    {
        #pragma omp parallel num_threads (3)
        {
_tprintf (_T("Hello, world, thread num = %d\n"), omp_get_thread_num ());
        }
    }
```

Замість дев'яти кратного повторення рядка одержуємо:

Hello, world, thread num = 0
Hello, world, thread num = 0
Hello, world, thread num = 0

Якщо рекурсивна функція, можна обмежити кількість рівнів:

omp_set_max_active_levels – Версія 3!!!

ОСОБЛИВОСТІ ВИКОРИСТАННЯ СЕКЦІЙ

Спосіб 1.

Виконувати різний код залежно від потоку, що його виконує. Нагадуємо, що всі потоки мають номера, починаючи з 0. У цьому випадку програму має вигляд:

```
int Thread = omp_get_num_thread ();
if (Thread == 0)
    fun0 (...);
else
if (Thread == 1)
    fun1 (...);
…
```

Недоліки:

1 якщо кількість функцій більше, ніж кількість потоків, то цей метод використовувати не можна.

2 розголудження – не можна передбачити

ОСОБЛИВОСТІ ВИКОРИСТАННЯ СЕКЦІЙ

Спосіб 2. Використовується, якщо всі функції, які треба виконати, мають однаковий заголовок. У цьому випадку формується масив функцій, які треба виконати й цикл формується для виконання цього масиву.

Приклад. Нехай необхідно виконати функції:

DWORD Fun1 (PVOID p){…}
DWORD Fun2 (PVOID p){…}
…
DWORD Fun (PVOID p){…}

Визначимо тип заголовка для цих функцій:

*typedef DWORD (*PFUN) (PVOID p);*

Визначимо масив функцій, які треба виконати:

PFUN pFuns [] = {Fun1, Fun2, ..., Fun};

Цикл для паралельного виконання цих функцій:

```
#pragma omp parallel for
for (int i = 0; i < sizeof (pFuns) / sizeof (pFuns [0]; ++i))
    (pFuns[i]) (...)
```

Недолік – потребує переробки коду.

ОСОБЛИВОСТІ ВИКОРИСТАННЯ СЕКЦІЙ

Розподіл навантаження!!!

```
#pragma omp parallel [Параметри]
{
    …
    #pragma omp sections [Параметри]{
        #pragma omp section]{
            Секція 1
        }
        #pragma omp section]{
            Секція 1
        }
        …
    }
    …
Ї
```

Скорочений вид директиви:

```
#pragma omp parallel sections [параметри]
    #pragma omp section]{bn bn
            Секція 1
        }
        #pragma omp section]{
            Секція 1
        }
        …
    }
```

ОСОБЛИВОСТІ ВИКОРИСТАННЯ СЕКЦІЙ

Параметри;
reduction(*reduction-identifier:list*)
nowait

private(*list*),

firstprivate(*list*),

lastprivate(*list*),, nowait

ОСОБЛИВОСТІ ВИКОРИСТАННЯ СЕКЦІЙ. ПРИКЛАД

Хай необхідно обчислити A = B*B – C * C

1. **2 множення та одне –. Якщо 2 ядра, то фактично t(*) + t(-)/2**
2. **A = (B-C)(B+C). T(-) + t(*)/2**

Обираємо 2 варіант

```
#pragma omp sections
{
      #pragma omp section
      {
            msub (B, C, R1, n);
      }
      #pragma omp section
      {
            madd (B, C, R2, n);
      }

}
ParMmul (R1,R2, A, n);
```

ОСОБЛИВОСТІ ВИКОРИСТАННЯ СЕКЦІЙ. ПРИКЛАД

```
void ParMulMatr(float *a, float *b, float *c, size_t n){
        memset(c, 0, n * n * sizeof(float));
        #pragma omp parallel for
        for (int i = 0; i < n; i++)
        {
                float *pc = &c[i * n];
                float *pa = &a[i * n];
                for (int j = 0; j < n; j++){
                        float r = pa[j];
                        float *pb = &b[j * n];
                        for (int k = 0; k < n; k++)
                                pc[k] += r * pb[k];
                }
        }
} 11.7   0.39    0.24
```

КЛАСИ ПА’МЯТІ В С++ ПРОГРАМАХ

Зовнішні змінні. Змінна називається *зовнішньою*, якщо вона задана поза функціями. Ця змінна доступна у всіх функціях, які задані після опису такий змінної, та навіть в функціях з інших файлів, якщо визначено extern.

Приклад:

```
int x [10];
int main(){}
```

Статичні змінні. Визначаються поза функціями або всередині них. Якщо статична змінна визначена усередині функцій, вона доступна тільки усередині цієї функції, але після виходу з функції, пам'ять, виділена для неї, а значить і її поточне значення, зберігається. Якщо статична змінна оголошена поза функціями, вона доступна для всіх функцій, які визначені нижче.

Приклад:

```
int fun (){
        static int first = 0;
        if (first == 0){
                …  ;first = 1;}
        }
```

I

Локальні змінні. Визначаються в функції. Пам'ять виділяється після входу в функцію. Область дії змінної - блок, де вона оголошена. Пам'ять автоматично звільняється після завершення функції. Початкове значення не визначене.

Приклад:

```
int fun (){
        int first = 0;
}
```

КЛАСИ ЗМІННИХ В С++ ПРОГРАМАХ

Зовнішні змінні. Змінна називається *зовнішньою*, якщо вона визначена поза функцій. Ця змінна доступна у всіх функціях, які задані після опису такий змінної.

Приклад:

```
int x [10];
int main(){}
```

Статичні змінні. Визначаються поза функціями або усередині них. Якщо статична змінна визначена усередині функцій, вона видима тільки усередині цієї функції, але після виходу з функції, пам'ять, виділена для неї, а значить і її поточне значення, зберігається. Якщо статична змінна оголошена поза функціями, вона видима для всіх функцій, які визначені нижче.

Приклад:

```
int fun (){
        static int first = 0;
        if (first == 0){Инициализация; first = 1;}
}
int main(){}
```

Локальні змінні. Визначаються в функції. Пам'ять виділяється після входу в функцію. Область дії змінної - блок, де вона оголошена. Пам'ять автоматично звільняється після завершення функції. Початкове значення не визначене.

Приклад:

```
int fun (){
        int first = 0;
}
```

КЛАСИ ЗМІННИХ ДЛЯ OPEN MP

Всі змінні діляться на 2 класи **private** і **shared**.

- Клас ***private*** змушує компілятор кожному потоку виділити свою пам'ять для цієї змінної (свій екземпляр змінної). Змінні цього класу мають область дії - тільки один потік, зміна значення цієї змінної усередині потоку не впливає на її значення в інших потоках. Повинне бути задане початкове значення до використання, тому що воно при створенні копій не визначено
- Змінні класу ***shared*** існують у єдиному екземплярі, є загальнодоступними змінними для всіх потоків. При доступі до цих змінних необхідно забезпечити ексклюзивний доступ, якщо значення таких змінних може змінюватися. Виключення становить список змінних для параметра ***reduction***, коректність доступу до цих змінних забезпечена автоматично

ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ ЗА ЗАМОВЧУВАННЯМ

1. Якщо змінна задана **поза паралельною області**, вона вважається змінної загального доступу (***shared***). Виключення - параметр циклу, для якого створюються свої копії.
2. Якщо пам'ять під змінну виділена **динамічно**, то сама пам'ять типу ***shared***, а покажчик на неї – заданого типу.
3. **Статичні дані** вважаються типу ***shared***.
4. **Змінні, оголошені як константи (*const*)**, уважаються типу ***shared***.
5. Якщо **змінна оголошена усередині паралельної області**, вона вважається особистою змінною потоку (***private***).

ПРОГРАМНЕ ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ

Параметр ***shared*** (Змінна [, Змінна, …])...

Параметр ***private*** (Змінна[,Змінна, …])...

Приклади.

Приклад 1.

```
int a = 0;
#pragma omp parallel private(a){
        a++; …
}
```

Чому помилка?

Код помилковий, тому що не визначене початкове значення локальної копії для паралельного потоку

Компілятор мови MS2008 видає попередження про використання локальної змінної, якій не задано початкового значення.

Виправлений код: (в залежності від необхідності)

```
int a = 0;
#pragma omp parallel {
Atomic
        a++;
…
}
```

```
int a = 0;
#pragma omp parallel private(a){
        a = 0;…
        a++;
        …
}
```

ПРОГРАМНЕ ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ

Приклад 2. Визначити очікуваний результат, якщо кількість потоків за замовчуванням дорівнює 3.

```
int a;
#pragma omp parallel private(a) num_threads (3){
...
a = 0;
#pragma omp for
for (int i = 0; i < 10; i++){
 #pragma omp atomic
 a++;
}
#pragma omp critical{
 cout << "a = " << a;
}
}
```

ПРОГРАМНЕ ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ

Приклад 2. Визначити очікуваний результат, якщо кількість потоків за замовчуванням дорівнює 3.

```
int a;
#pragma omp parallel private(a) num_threads (3){
...
a = 0;
#pragma omp for
for (int i = 0; i < 10; i++){
 #pragma omp atomic
 a++;
}
#pragma omp critical{
 cout << "a = " << a;
}
```

} Змінна ***a*** є приватною для кожного потоку, у тому числі й потоків, які формуються циклом ***for***. Тому змінна ***a*** визначає, скільки ітерацій виконується в кожному циклі. Значення рівні 4, 3, 3

ПРОГРАМНЕ ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ. ПРИКЛАД 3

```
int a = 10;
#pragma omp parallel private(a)
{
	#pragma omp sections
	{
		#pragma omp section
		{
		a += 1; printf("sec 0 thread num = %d, a = %d\n", omp_get_thread_num(), a);
		}
		#pragma omp section
		{
		a += 100; printf("sec 1 thread num = %d, a = %d\n", omp_get_thread_num(), a);
		}

	}
	printf("paral thread num = %d, a = %d\n", omp_get_thread_num(), a);
} printf("not paral thread num = %d, a = %d\n", omp_get_thread_num(), a);
```

ПРОГРАМНЕ ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ. ПРИКЛАД 3

```
int a = 10;
#pragma omp parallel private(a)
{
	#pragma omp sections
	{
		#pragma omp section
		{
		a += 1; printf("sec 0 thr_num = %d, a = %d\n", omp_get_thread_num(), a); // sec 0, _, 1
		}
		#pragma omp section
		{
		a += 100; printf("sec 1 thr_num = %d, a = %d\n",omp_get_thread_num(), a); //sec 1, _,100
		}

	}
	printf("paral thr_num = %d, a = %d\n", omp_get_thread_num(), a); // (_, 100) {_, 1} (_, 0) (_, 0)
} printf("not paral thread num = %d, a = %d\n", omp_get_thread_num(), a); // (0, 10)
```

ПРОГРАМНЕ ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ. ПРИКЛАД 4

Приклад 4

```
#pragma omp parallel
    {
        int x [10];
        #pragma omp for ordered
            for (int i = 0; i < sizeof (x)/ sizeof (x [0]); ++i)
            {
                x [i] = i;

            }
        #pragma omp single
            for (int i = 0; i < sizeof (x)/ sizeof (x [0]); ++i)
                printf ("x[%d] = %x\n", i, x [i]);
    }
```

Хай потоків 4.

ПРОГРАМНЕ ВИЗНАЧЕННЯ КЛАСУ ПАМ'ЯТІ. ПРИКЛАД 4

Приклад 4

x[0] = 104f7d0 x[1] = 104f824 x[2] = f5da0d5 x[3] = a32527a3 x[4] = fffffffe

x[5] = f643071 x[6] = 12618e1 x[7] = 1262170 x[8] = 8 x[9] = 9

x[0] = 5 x[1] = e4f7b0 x[2] = f5da0d5 x[3] = 3 x[4] = 4

x[5] = 5 x[6] = 12618e1 x[7] = 1262170 x[8] = 1 x[9] = 64

x[0] = 0 x[1] = 1 x[2] = 2 x[3] = 3 x[4] = 4f0000

x[5] = 4f0000 x[6] = 939ad0 x[7] = f x[8] = 939aec x[9] = 1ffff

x[0] = 5 x[1] = f4fc18 x[2] = f5da0d5 x[3] = a32527a3 x[4] = fffffff

x[5] = f643071 x[6] = 6 x[7] = 7 x[8] = 2 x[9] = 0

ПАРАМЕТР FIRSTPRIVATE

Клас ***firstprivate***.

```
int x = 0;
#pragma omp parallel private (x){
    x = ::x;
}
```

Приклад:

```
int a = 10;
#pragma omp parallel firstprivate(a) num_threads (4)
{
    _tprintf (_T("thread = %d\ta = %d\n"), omp_get_thread_num (), a);
}
```

Зазвичай використовується, якщо необхідно задати однакові початкові значення для усіх потоків!!!

ПАРАМЕТР LASTPRIVATE

Клас ***lastprivate*** задає список змінних, які по завершенню паралельної секції стають такими, якими вони були б при послідовному виконанні.

Зазвичай використовується для параметра циклу, значення якого використовується за межами циклу.

```
int i;
#pragma omp parallel
{
#pragma omp for lastprivate(i)
for (i=0; i<n-1; ++i)
a[i] = b[i] + b[i+1];
}
a[i]=b[i]; /* i == n-1 */
```

ПАРАМЕТРИ FIRSTPRIVATE, LASTPRIVATE. ПРИКЛАД

```
int a = 10;
#pragma omp parallel num_threads (2){
    #pragma omp sections firstprivate (a), lastprivate(a){
        #pragma omp section {
            _tprintf (_T("thread = %d\ta = %d\n"), omp_get_thread_num (), a);
            a = 100;
            _tprintf (_T("thread = %d\ta = %d\n"), omp_get_thread_num (), a);
        }
        #pragma omp section   {
        _tprintf (_T("thread = %d\ta = %d\n"), omp_get_thread_num (), a);
        a = 200;
        _tprintf (_T("thread = %d\ta = %d\n"), omp_get_thread_num (), a);
        }
    }
    #pragma omp barrier
    #pragma omp critical
    {
        _tprintf (_T("thread = %d\ta = %d\n"), omp_get_thread_num (), a);
    }
}
```

ПАРАМЕТРИ FIRSTPRIVATE, LASTPRIVATE. ПРИКЛАД

ithread = 0 a = 10
thread = 2 a = 10
thread = 0 a = 100
thread = 2 a = 200
thread =1 a = 200
thread =3 a = 200
thread =0 a = 200
thread =2 a = 200

ПАРАМЕТРИ DEFAULT, THREADPRIVATE, COPYIN

Для деяких змінних є правила визначення типу по замовченню. Можна явно визначити цей тип (параметр *default*):

default(shared | none) ***shared*** – усі змінні типу ***shared****;*

none *– для усіх змінних треба задавати тип явно.*

Завдання особистої змінної паралельної області: ***private***

Завдання особистих змінних для групи паралельних областей використовується директива:

threadprivate.

Аналог TLS (Thread Local Storage). Змінна визначається як зовнішня змінна. Для мастер потоку ця пам’ять використовується.

Для решти потоків – свої власні копії.

Ініціалізація необхідна до першого використання.

Для копіювання початкового значення в потоки copyin (список)

Приклад. Хай використовується стільки клієнтів, скільки ядер. Сформувати імена клієнтів UsrName<Номер потоку>. В паралельній секції використовувати відповідні імена

THREADPRIVATE, ПРИКЛАД

```
char UsrName[16] = { 0 };
#pragma omp threadprivate (UsrName)
void fun1(){
    sprintf_s(UsrName, "UsrName %d", omp_get_thread_num());
    printf("fun1: thread %d %s\n", omp_get_thread_num(), UsrName);
}
void fun2(){
    printf("fun2: thread %d %s\n", omp_get_thread_num(), UsrName);
}
void fun3(){
    printf("fun3: thread %d %s\n", omp_get_thread_num(), UsrName);
}
#pragma omp parallel
{
    fun1(); fun2(); fun3();
}
```

THREADPRIVATE, ПРИКЛАД

fun1: thread 0 UsrName 0
fun1: thread 3 UsrName 3
fun2: thread 0 UsrName 0
fun2: thread 3 UsrName 3
fun3: thread 0 UsrName 0
fun3: thread 3 UsrName 3
fun1: thread 2 UsrName 2
fun1: thread 1 UsrName 1
fun2: thread 2 UsrName 2
fun2: thread 1 UsrName 1
fun3: thread 2 UsrName 2
fun3: thread 1 UsrName 1

ПРИКЛАД ВИКОРИСТАННЯ РІЗНИХ КЛАСІВ ПАМ'ЯТІ

Знайти помилки:

```
int x, y, z[1000];
#pragma omp threadprivate(x)
void fun (int a) {
        const int c = 1;
        int i = 0;
        #pragma omp parallel default(none) private(a) shared(z){
            int j = omp_get_num_threads();          // OK or Error?
            a = z[j];                               // OK or Error?
            x = c;                                  // OK or Error?
            z[i] = y;                               // OK or Error?
            #pragma omp for firstprivate(y)
                        for (i=0; i<10 ; i++)
                        {
                                    z[i] = y;       // OK or Error?
                        }
            z[i] = y;                               // OK or Error?
        }
}
```

ПРИКЛАД ВИКОРИСТАННЯ РІЗНИХ КЛАСІВ ПАМ'ЯТІ

```
void fun(int a) {
    const int c = 1;
    int i = 0, y = 2;
#pragma omp parallel default(none) firstprivate(y, i) private(a) shared(z)
{
    int j = omp_get_num_threads();
    y = 2 * j + y;
    a = z[j];
    x = c;
    z[i] = y;
    #pragma omp for
    for (i = 0; i<10; i++)
            z[i] = y;
    z[i] = y;
}
}
```

ВИКОРИСТАННЯ КЕШУ (FALSE SHARING)

Хай при виконанні потоків формуються значення елементів масиву, номер елементу співпадає з номером потоку.

Тоді в Кеші вони знаходяться в одному елементі, при запису – кеш не дійсний, якщо змінюється хоч один елемент

Приклад

```
double work (int i, int j){
double di = i, dj = j;
return sqrt (di+1) * sin (di + 1) * tan (dj + 1);
}
void Fun1 (double *A, int n, int m){
#pragma omp parallel for
for (int j = 0; j < n; j++)
for (int i = 0; i < m; i++)
A[j]+= work (i, j);
}
```

```
void Fun2 (double *A, int n, int m){
#pragma omp parallel for
 for (int j = 0; j < n; j++){
                double temp = 0;
                 for (int i = 0; i < m; i++)
                                temp+= work (i, j);
                 A[j] = temp;
}
```

time:

1.09 0.54 0.36

}

ВИКОРИСТАННЯ КЕШУ (FALSE SHARING)

```
const int n = 1024, m = 1024*1024;
double A [n];
…
Fun1 ( A, n, m);
…
Fun2 ( A, n, m);
```

Отримане значення 13.25 VS 13.35

```
const int n = 8, m = 1024*1024 * 128;
double A [n];
…
Fun1 ( A, n, m);
…
Fun2 ( A, n, m);
```

Отримане значення 28.96 VS 14.01

Чому?

Усі елементи масиву А записуються в один блок Кешу, Кожного разу Кеш не дійсний!!!

ОБМЕЖЕННЯ НА СПИСОК ЗМІННИХ ПРИ ВИЗНАЧЕННІ ЇХНІХ КЛАСІВ. РЕКОМЕНДАЦІЇ З ВИКОРИСТАННЯ

- Змінна в виразах *private*, *firstprivate*, *lastprivate*, не повинна мати посилальний тип (тобто бути покажчиком або посиланням). Якби це було покажчиком або посиланням, то фактично кожний потік би працював з однієї й тією же областю пам'яті, що приводило б до проблем спільного доступу.
- Якщо змінна в цих виразах є екземпляром класу, у цьому класі повинен бути визначений конструктор копіювання. Якщо немає такого конструктора, то некоректно будуть створені поля, які є масивами.

Попередній огляд типів змінних показує, наскільки важливо правильно вибрати **клас змінної.** Для того щоб програміст не забував установлювати класи всіх змінних, які використовують, що зменшить імовірність помилки, необхідно задати параметр *default*(*none*). У цьому випадку компілятор помічає як помилкові оператори використання змінних, для яких клас не заданий.

СИНХРОНІЗАЦІЯ ПОТОКІВ

Засоби синхронізації потоків одного процесу для Windows

- **Interlocked …** функції для захисту простих змінних у випадку виконання найпростіших операцій;
- **Критичні секції** для забезпечення виконання заданої ділянки коду одночасно тільки одним потоком (ексклюзивне виконання).
- **Захист від взаємних блокувань** необхідний, якщо одночасно використовується кілька критичних секцій і (або) об'єктів синхронізації. При цьому захист може бути виконано тільки за рахунок їхнього коректного використання.
- **Досягнення необхідного порядку виконання потоків** забезпечується за рахунок використання об'єктів ядра, наприклад подій, семафорів.

ОГЛЯД ЗАСОБІВ СИНХРОНІЗАЦІЇ OPEN MP

Директива ***barrier*** – для очікування завершення усіх паралельних гілок

Директива ***atomic*** – для атомарного виконання заданої операції над змінною;

Критичні секції – для ексклюзивного виконання заданої ділянки коду. Реалізуються за допомогою директив;

Замки - для ексклюзивного виконання заданої ділянки коду. Реалізуються за допомогою функцій;

ДИРЕКТИВА BARRIER

Загальний вид директиви:

#pragma omp barrier

Використовується, якщо необхідно явно вказати, що в даній крапці коду повинні бути завершені всі паралельні потоки.

Неявно ця директива використовується наприкінці циклів, паралельних областей, секцій.

Приклад. Якщо в паралельній області є оператор, який рахує кількість потоків, то треба, щоб усі потоки виконали цей оператор.

ДИРЕКТИВА ATOMIC

Загальний вид директиви:

#pragma omp atomic

Оператор

Атомарно *обчислюється значення лівої частини оператора, що може бути простою змінною.*

Допускаються оператори виду:

x= x <Бінарна операція >expr; → x <Бінарна операція >(expr)

x= expr <Бінарна операція >x → (expr) <Бінарна операція >x

x <Бінарна операція >= expr;

x++; ++x; x--; --x

Де

*<Бінарна операція > - операція +, - *, /, &, |, ^, <<, >>.*

Приклад. *Дотепер для визначення кількості потоків усередині паралельної секції використався оператор count++. Треба написати:*

```
int count = 0;
#pragma omp parallel{
#pragma omp atomic
++count;
}
printf("count =%d\n", count);
```

ПОРІВНЯННЯ ATOMIC ТА REDUCTION

Хай необхідно в паралельній області виконати код: count+=value, де count – зовнішня змінна для цієї області

Цю операцію можна виконати 2 способами: atomic та reduction.

Код для обох варіантів наведено нижче.

Що краще?

int count = 0; *#pragma omp parallel* *{* *#pragma omp atomic* *++count;* *}* *printf("count =%d\n", count);*	*int count = 0;* *#pragma omp parallel\ reduction(+:count)* *{* *++count;* *}* *printf("count =%d\n", count);*

КРИТИЧНІ СЕКЦІЇ

Загальний вид директиви:

#pragma omp critical
[(<ім'я_критичної_секції>)]{Блок}

#pragma omp critical cs
{
…
}
	…
	EnterCriticalSection(&cs)
	…
	LeaveCriticalSection(&cs)

ПРИКЛАДИ ВИКОРИСТАННЯ КРИТИЧНИХ СЕКЦІЙ

Приклад 1. Нехай паралельна секція використовує оператор виведення:

wcout << _T(“Start = “ << Start << endl;

Для того щоб кожний потік виводив цілком свій рядок, необхідно використовувати критичну секцію:

EnterCriticalSection (&cs); *wcout << _T(“Start = “ << Start << endl;* *LeaveCriticalSection (&cs);*	*#pragma omp critical* *{* *wcout << _T(“Start = “ << Start << endl;* *}*

ПРИКЛАДИ ВИКОРИСТАННЯ КРИТИЧНИХ СЕКЦІЙ

Забеспечити паралельне виконання функції обчислення максимуму та його номеру

```
int max1(float *x, int n){
float Max = x[0];
int Num = 0;
for (int i = 1; i < n; ++i){
if (x[i] > Max){
Max = x[i];Num = i;
}}return Num;}
```

```
int max2(float *x, int n){
float Max = x[0];
int Num = 0;
#pragma omp parallel for
for (int i = 1; i < n; ++i){
if (x[i] > Max){
Max = x[i];Num = i;
}}return Num;
}
```

Помилковий варіант

```
int max3(float *x, int n){
float Max = x[0];
int Num = 0;
#pragma omp parallel for
for (int i = 1; i < n; ++i){
#pragma omp critical
{
if (x[i] > Max){
Max = x[i];Num = i;
}}
}
return Num;
}
```

ПРИКЛАДИ ВИКОРИСТАННЯ КРИТИЧНИХ СЕКЦІЙ

```
int max4(float *x, int n){
float Max = x[0];
int Num = 0;
#pragma omp parallel for
for (int i = 1; i < n; ++i){
if (x[i] > Max){
#pragma omp critical
{
if (x[i] > Max){
Max = x[i]; Num = i;
} //if
}// critical
}// if
}// for
return Num;
}
```

max1: 0.025; max2:0.006;
Max3:1.96, max4: 0.006
Max5:0.0066

```
int max5(float *x, int n){
int mn[16];
int nt = omp_get_max_threads();
int h = n / nt;
#pragma omp parallel for
for (int p = 0; p < nt; p++){
int b = p * h, e = b + h;
if (p == nt) e = n;
int nlm = b; float lm = x[b];
for (int i = b + 1; i < e; i++){
if (x[i] > lm){
lm = x[i];nlm = i;
}}
mn[p] = nlm;
}
int gmn = mn[0]; float gm = x[gmn];
for (int p = 1; p < nt; p++){
if (gm < x[mn[p]]){
gmn = mn[p];gm = x[gmn];
}}return gmn;}
```

ПРИКЛАДИ ВИКОРИСТАННЯ КРИТИЧНИХ СЕКЦІЙ

```
int OMPMax5 (int *x, int n){
int m, nm;
int max [16], nmax[16];
int nThr = omp_get_max_threads ();
int h = (n + nThr - 1)/nThr;
#pragma omp parallel for
for (int i = 0; i < nThr ; ++i){
int j0 = i * h; int jn = j0 + h;
if (i == nThr - 1) jn = n;
int curm = x [j0];
for (int j = j0; j < jn; ++j) {
if (x [j] > curm) curm = x [j];
}
max [i] = curm;
}
```

```
m = max [0];
for (int i = 1; i < nThr; i++)
{
if (max [i] > m) m = max[i];
}
return m;

}
```

ПРИКЛАДИ ВИКОРИСТАННЯ КРИТИЧНИХ СЕКЦІЙ

N = 2000000, 4 ядерний комп’ютер
_OPENMP = 200203

max1	(послідовний)	time = 0.0014
max2	помилковий варіант	
max3	if в КС	time = 0.33
max4	Можлива помилка	
max5	Мінімальна критична секція	time = 0.00062
max6	Без критичної секції	time = 0.00058

Прискорення: 2.4 рази

ПЕРЕВАГИ ТА НЕДОЛІКИ КРИТИЧНИХ СЕКЦІЙ

Перевага критичних секцій: Найефективніший метод захисту після *atomic*.

Недоліки критичних секцій.

1. Один потік не може багаторазово ввійти в критичну секцію без попереднього її закриття.
2. Немає можливості перевірити стан критичної секції.
3. Якщо критична секція використовується у функції й захищає дані, які передаються як параметр, неможливо перевірити, чи дійсно має сенс цей захист, наприклад

```
void fun (void * par){
        ...
        critical{
        // Модифікація й використання par
        }
        ...
}
…
#pragma omp parallel sections{
#pragma omp section{
fun (data1);
}
#pragma omp section{
fun (data2);
}
}
```

Тут функції *fun* передаються різні параметри, але одночасний доступ до цих даних захищено.

4. Не можна з критичної секції вийти, використовуючи goto, *break, continue,* тому що тоді критична секція нормально не завершиться.

«ЗАМКИ»

«Замки» аналогічні критичним секціям, але використовують функції бібліотеки замість директив.

Для використання «замка» необхідно:

- Задати початковий стан «замку» - до першого використання (аналог функція *InitializeCriticalSection*).
- Закрити «замок» на початку критичної секції (аналог функція *EnterCriticalSection*).
- Відкрити «замок» наприкінці критичної секції (аналог функція *LeaveCriticalSection*).
- Знищити «замок» (аналог функція *DeleteCriticalSection*).
- «Замок» має ім'я, тому можна використовувати декілька «замків».

ТИПИ ДАНИХ ТА ФУНКЦІЇ ДЛЯ «ЗАМКА»

typedef void * omp_lock_t;
typedef void * omp_nest_lock_t;
Ініціалізація «замків»:
- ***void omp_init_lock(omp_lock_t *lock);***
- **void omp_init_nest_lock (omp_nest_lock_t);**

Закриття «замків»
- ***void omp_set_lock(omp_lock_t *lock);***
- **void omp_ set_nest_lock (omp_nest_lock_t);**

Відкриття «замків»
- ***void omp_unset_lock(omp_lock_t *lock);***
- **void omp_ unset_nest_lock (omp_nest_lock_t);**

Знищення «замка»
- ***void omp_destroy_lock(omp_lock_t *lock);***
- **void omp_ destroy_nest_lock (omp_nest_lock_t);**

Перевірка стану «замка» (аналог функція *TryEnterCriticalSection*)
- ***int omp_test_lock(omp_lock_t *lock);***
- ***int omp_test_nest_lock(omp_nest_lock_t *lock);***

ЗАБЕЗПЕЧЕННЯ АВТОМАТИЧНОГО ВІДКРИТТЯ «ЗАМКА» ПІСЛЯ ЗАВЕРШЕННЯ КРИТИЧНОЇ СЕКЦІЇ (ВИЗНАЧЕННЯ КЛАСУ)

```
class omp_lock {
private: omp_lock_t *lock;
public:
// Конструктор. Виконує закриття «замка»
omp_lock (omp_lock_t &lock) {
this->lock = lock; omp_set_lock (this->lock);
}
// Деструктор. Виконує відкриття «замка»
~omp_ lock (){
    omp_unset_lock (lock);}
};
```

ЗАБЕЗПЕЧЕННЯ АВТОМАТИЧНОГО ВІДКРИТТЯ «ЗАМКА» ПІСЛЯ ЗАВЕРШЕННЯ КРИТИЧНОЇ СЕКЦІЇ (ВИКОРИСТАННЯ КЛАСУ)

```
omp_lock_t lock;
omp_ init_lock(& lock);
…
#pragma omp parallel
{
…
{
        omp_lock Lock (lock);
        // Критична секція
}
…
}
omp_ destroy_lock(& lock);
```

ПРИКЛАДИ ВИКОРИСТАННЯ «ЗАМКІВ». ЗВИЧАЙНІ “ЗАМКИ”

```
void Lock1 ()
{
    omp_init_lock(&simple_lock);
     #pragma omp parallel num_threads(2){
            int tid = omp_get_thread_num();
            int i;
            for (i = 0; i < 5; ++i) {
                    omp_set_lock(&simple_lock);
                    _tprintf(_TEXT("Потік %d-почав критичну ділянку\n"), tid);
                    _tprintf(_TEXT("Потік %d-скінчив критичну ділянку\n"),
                    tid);
                    omp_unset_lock(&simple_lock);
            }
    }
    omp_destroy_lock(&simple_lock);
}
```

ПРИКЛАДИ ВИКОРИСТАННЯ «ЗАМКІВ». ВКЛАДЕНІ “ЗАМКИ”

```
void Lock2 () {
 omp_init_nest_lock(&nest_lock);
 #pragma omp parallel num_threads(2) {
      int i;
       _tprintf (_TEXT("Потік %d - почав роботу\n"), omp_get_thread_num());
      omp_set_nest_lock(&nest_lock);
       _tprintf (_TEXT("Потік %d - захопив «замок»\n"), omp_get_thread_num());
      for (int i = 0; i < 4; ++i)  {
              omp_set_nest_lock(&nest_lock);
              _tprintf_s(_TEXT("Потік %d - почав критичну ділянку\n"),
              omp_get_thread_num());
              _tprintf (_TEXT("Потік %d - скінчив критичну ділянку\n"),
              omp_get_thread_num());
              omp_unset_nest_lock(&nest_lock);
       }
      omp_unset_nest_lock(&nest_lock);
}
omp_destroy_nest_lock(&nest_lock);
}
```

ПРИКЛАДИ ВИКОРИСТАННЯ «ЗАМКІВ». ЗАДАЧА ПРО ФІЛОСОФІВ

```
omp_lock_t forks[N];
void think(int i) {
	printf("%d thinking...\n", i); Sleep(1000);
}
void eat(int i) {
	int fork_first; fork_second;
	if (i == N - 1) {
		fork_first = 0; fork_second = i;
	}
	else {
		fork_first = i; fork_second = i + 1;
	}
	printf("%d ask fork (%d)\n", i, fork_first); omp_set_lock(&forks[fork_first]);
	printf("%d ask fork (%d)\n", i, fork_second); omp_set_lock(&forks[fork_second]);
	printf("%d EAT...\n", i); Sleep(1000);
	omp_unset_lock(&forks[fork_second]); omp_unset_lock(&forks[fork_first]);
}
void simulation(int i) {
	for (int j = 0; j < 10; j++) {
		think(i); eat(i);
}}
```

ПРИКЛАДИ ВИКОРИСТАННЯ «ЗАМКІВ». ЗАДАЧА ПРО ФІЛОСОФІВ. ГОЛОВНА ФУНКЦІЯ

```
int main() {
	omp_set_num_threads(N);
	for (int i = 0; i < N; i++)
		omp_init_lock(&forks[i]);


	#pragma omp parallel for
		for (int i = 0; i < N; i++) {
			simulation(i);
}
```

ПОРІВНЯННЯ КРИТИЧНИХ СЕКЦІЙ, СТВОРЮВАНИХ ЗА ДОПОМОГОЮ ДИРЕКТИВ І ФУНКЦІЙ

	critical	lock
Простота використання	+	-
Швидкість	+	-
Використання для рекурсії	-	+ (*nest_lock*)
Автоматичне закриття	-	+(class)

ВИСНОВКИ

1. Open MP використовує класи пам'яті, які визначені для мови C++, але має і додаткові класи пам'яті (private, shared).
2. Для усіх змінних, для яких не визначено додаткового класу пам'яті, він визначається по замовченню в залежності від міста об'яви.
3. Особливу увагу при програмуванні треба приділити shared змінних, які можуть змінюватися тільки в ексклюзивному режимі.
4. Для виключення помилок, пов'язаних з невірним використанням класу, який визначається по замовченню, рекомендується використовувати default (none).
5. Для забезпечення ексклюзивного доступу можна використовувати усі засоби синхронізації WINDOWS та додаткові засоби Open MP.
6. Розглянуті та порівняні внутрішні засоби синхронізації Open MP.
7. При програмуванні треба пам'ятати, що, незалежно від засобу синхронізації, витрати, пов'язані з їх використанням, значні!!!

ПИТАННЯ ДЛЯ САМОСТІЙНОГО ВИВЧЕННЯ

1. Паралельне обчислення числа π
2. Паралельне Обчислення простих чисел
3. Рекомендації з використанні технології OPEN MP

Учбовий посібник: Паралельне програмування

МАТЕРІАЛИ ДЛЯ ЕКСПРЕС-КОНТРОЛЮ

- Що означає параметр *default (none)*, у якому випадку його має сенс задавати?
- У якому випадку параметр *private* варто замінити параметром: *firstprivate*; *lastprivate*; *threadprivate;*
- Нехай необхідно накопичувати суму елементів. Для цього можна використовувати *reduction* або директиву *#pragma omp atomic*. Який із цих способів більше ефективний? Чому?
- Чим відрізняється критична секція Windows і OPEN MP?
- Зрівняєте 3 варіанти завдання критичної секції: *CRITICAL_SECTION* (Windows), *critical* (OPEN MP), *locked* (OPEN MP) за наступними критеріями:
  - швидкість;
  - можливість використання в рекурсивних функціях;
  - можливість використання в потоках різних процесів;
  - можливість задати порядок виконання критичних секцій;
  - можливість звільнення критичної секції, зайнятий іншим потоком.
- Як виключити можливість взаємного блокування при використанні декількох вкладених критичних секцій?
- Чи можна класичну задачу синхронізації процесів (задачу про філософів, які обідають вирішити за допомогою засобів синхронізації OPEN MP?
- Які обмеження на оброблювачі виключень для паралельних областей коду?