Змістовний модуль:

РОЗРОБКА ПАРАЛЕЛЬНИХ АЛГОРИТМІВ І ПРОГРАМ

Розділ: Технології розробки паралельних програм

Лекція 12. ТЕХНОЛОГІЯ OPEN MP. ЗАКЛЮЧЕННЯ

ПИТАННЯ ДЛЯ ВИВЧЕННЯ

1. Класи змінних для OPEN MP
2. Засоби синхронізації потоків одного процесу для Windows
3. Засоби синхронізації OPEN MP. atomic
4. Засоби синхронізації OPEN MP. Критичні секції
5. Засоби синхронізації OPEN MP. Замки
6. Приклад використання

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

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.

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.

Обчислити суми рядків матриці, якщо кожний елемент задається формулою

R = sqrt (I + 1) * sin (i) * tg (j)

Отримати наступні варіанти:

Послідовний варіант без використання допоміжних змінних.

Послідовний варіант з використанням допоміжних змінних.

Паралельний варіант без використання допоміжних змінних.

Паралельний з використанням допоміжних змінних.

ВИКОРИСТАННЯ КЕШУ (FALSE SHARING)

```
double work(int i, int j){
    double di = i, dj = j;
    return sqrt(di + 1) * sin(di + 1) * tan(dj + 1);
}
void SecFun1(double *A, int n, int m){
    for (int j = 0; j < n; j++){
        A[j] = 0;                          // double temp = 0;
        for (int i = 0; i < m; i++)
            A[j] += work(i, j);            // temp += work(i, j)
        A[j] = temp;
    }
}
```

ВИКОРИСТАННЯ КЕШУ (FALSE SHARING)

```
void ParFun1(double *A, int n, int m){
    #pragma omp parallel for
        for (int j = 0; j < n; j++){
                A[j] = 0; // double temp = 0;
                for (int i = 0; i < m; i++)
                        A[j] += work(i, j);
                 A[j] =  temp;
        }
}
```

Результаті для n = 4; m = 1024*1024*128:

Sec	35 vs 34.6	(1 %)
Par	15 vs 12	(20%)

Чому?

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;

ДИРЕКТИВИ MASTER, SINGLE, BARRIER

Загальний вид директив:

#pragma omp master {}

#pragma omp single {}

#pragma omp barrier

Приклад . Хай в паралельній області треба ввести дані виконати обчислення, вивести результат.

ДИРЕКТИВИ MASTER, SINGLE, BARRIER

Приклад 2. Хай в паралельній області треба ввести дані виконати обчислення, вивести результат.

```
#pragma omp parallel
{
    #pragma omp single
    {
        // Введення даних
    }
    #pragma omp barrier
    #pragma omp for
    for (){} // обробка даних
    #pragma omp single
    {
        // Виведення даних
    }}
```

ДИРЕКТИВА ATOMIC

Загальний вид директиви:

#pragma omp atomic

Оператор

Змінна тільки проста!!!.

x= x <Бінарна операція >expr;

x <Бінарна операція >= expr;

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

Де

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

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;
}
```

```
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 - 1) 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

```
float max5(float*x, int * nm, int n){
  float res [16];
  int num[16];
  int tc=omp_get_num_procs(), h = n / tc;
  #pragma omp parallel for
  for (int k = 0; k < tc; k++){
    int beg = k * h, end = beg + h;
    if (k == tc - 1)
      end = n;
    res[k] = x[beg];
    num[k] = beg;
    for (int j = beg + 1; j < end; j++){
      if (x[j] > res[k]){
        res[k] = x[j]; num[k] = j;
      }
    }
  }
  float r = res[0]; int nm1 = num[0];
  for (int k = 1; k < tc; k++){
    if (res[k] > r){
      r = res[k]; nm1 = num[k];
    }
  }
  *nm = nm1;
  return r;
}
```

7 версія – повернення тільки номеру максимуму.

Win32

max = 32767 nm = 38723 time = 0.000753919

max = 32767 nm = 823037 time = 0.000837735

max = 32767 nm = 525763 time = 0.112897

max = 32767 nm = 823037 time = 0.000493917

max = 32767 nm = 38723 time = 0.000390857

max = 32767 nm = 38723 time = 0.000378884

max = 32767 nm = 38723 time = 0.000367765

ПРИКЛАДИ ВИКОРИСТАННЯ КРИТИЧНИХ СЕКЦІЙ

```
float max5(float*x, int * nm, int n){
  float res [16];
  int num[16];
  int tc=omp_get_num_procs(), h = n / tc;
  #pragma omp parallel for
  for (int k = 0; k < tc; k++){
    int beg = k * h, end = beg + h;
    if (k == tc - 1)
      end = n;
    res[k] = x[beg];
    num[k] = beg;
    for (int j = beg + 1; j < end; j++){
      if (x[j] > res[k]){
        res[k] = x[j]; num[k] = j;
      }
    }
  }
  float r = res[0];
  *nm1 = num[0]; //int nm1 = num[0];
  for (int k = 1; k < tc; k++){
    if (res[k] > r){
      r = res[k];
      *nm1 = num[k]; // nm1 = num[k];
    }
  }
  //*nm = nm1;
  return r;
}
```

```
max = 32767 nm = 38723 time = 0.00109517
max = 32767 nm = 525763 time = 0.000830893
max = 32767 nm = 525763 time = 0.114361
max = 32767 nm = 525763 time = 0.000492634
max = 32767 nm = 38723 time = 0.00039043
max = 32767 nm = 38723 time = 0.000370759
```

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. Не можна з критичної секції вийти, використовуючи goto, *break, continue,* тому що тоді критична секція нормально не завершиться.
4. Не можна створити масиву критичних секцій (філософи).
4. Якщо критична секція використовується в функції й захищає дані, які передаються як параметр, неможливо перевірити, чи дійсно має сенс цей захист

ПЕРЕВАГИ ТА НЕДОЛІКИ КРИТИЧНИХ СЕКЦІЙ

```
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.

«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)

```
omp_lock_t l;
class c_omp_lock {
    private: omp_lock_t *lock;
    public:
    c_omp_lock(omp_lock_t *l) {
            lock = l;
            omp_set_lock(lock);
    }
    ~c_omp_lock(){
            omp_unset_lock(lock);
    }
};
```

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

```
omp_init_lock(&l);
#pragma omp parallel
{
    {       c_omp_lock lock(&l);
            cout << "thread " << omp_get_thread_num() << " "<< "string1 " <<
    "string2\n";
    }
    {       c_omp_lock lock(&l);
            cout << "thread " << omp_get_thread_num() << " " << "string3 " <<
    "string4\n";
    }
}
```

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]);
}
```

ПРИКЛАДИ ВИКОРИСТАННЯ «ЗАМКІВ». ЗАДАЧА ПРО ФІЛОСОФІВ

```
void simulation(int i) {
    for (int j = 0; j < 10; j++) {
        think(i); eat(i);
    }
}

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?