Змістовний модуль:

РОЗРОБКА ПАРАЛЕЛЬНИХ АЛГОРИТМІВ І ПРОГРАМ

Розділ: Технології розробки паралельних програм

Лекція 10. ТЕХНОЛОГІЯ OPEN MP. ПАРАЛЕЛІЗАЦІЯ ЦИКЛІВ. РОЗПОДІЛ НАВАНТАЖЕННЯ МІЖ ПОТОКАМИ

ПИТАННЯ ДЛЯ ВИВЧЕННЯ

1. Повторення
2. Організація паралельного виконання циклу за допомогою OPEN MP .
3. Обмеження на цикли для паралельного виконання
4. Паралельне виконання циклів і функція rand
5. Накопичення значень (параметр reduction)
6. Розподіл навантаження між потоками
7. Відключення синхронізації по виходу із циклу. Параметр nowait
8. Упорядкування ітерацій у циклі. Параметр ordered
9. Вкладення паралельних ділянок коду

ВКЛЮЧЕННЯ РЕЖИМУ ПІДТРИМКИ OPEN MP

Для включення цього режиму необхідно:

1. Створити проект. Проект може бути будь-якого типу, у тому числі консольний.
2. Додати до проекту *C++* файл, якщо обрано пустий проект.
3. У поле Властивостей: *Project→ Properties→ <Ім'я проектуProperty→*

Configuration Properties → C/C++ → Language → OPEN MP вибрати *Yes*.

Увага!!!

Якщо Ви забули включити режим підтримки OPEN MP, то програма, в якій використовуються можливості цього режиму при трансляції, не буде видавати помилок, а просто буде працювати в послідовному режимі, тому настійно рекомендуємо перевіряти успішність включення цього режиму!

ПЕРЕВІРКА УСПІШНОСТІ ВКЛЮЧЕННЯ РЕЖИМУ ПІДТРИМКИ OPEN MP

Режим включений → компілятор визначає макрос **_OPENMP**.

Структура макроса: yyyymm - ціле дане, старші 4 цифри якого задають рік, а молодші місяць версії OPEN MP.

Таким чином, якщо макрос _OPENMP визначений, то можна зробити висновок про успішність включення режиму підтримки OPEN MP, а значення цієї змінної фактично визначає версію OPEN MP.

Код для перевірки успішності включення й визначення версії OPEN MP:

```
#ifdef _OPENMP
    _tprintf (TEXT("OPEN MP is Support. Version:Year %d, Month = %d\n"),
            _OPENMP/100, _OPENMP%100);
#else
            printf ("_OPEN MP is not defined.\n");
#endif
```

Для VS2010 Year = 2002, Month = 3. Ось чому ця реалізація не повністю підтримує можливості останніх версій.

Для VS2010 + Intel Year = 2008, Month = 5.

КЛАСИФІКАЦІЯ ТА ЗАГАЛЬНИЙ ВИД ДИРЕКТИВ

Директиви діляться на:

- директиви визначення паралельних ділянок;
- директиви визначення класу пам'яті змінних;
- директиви для синхронізації.

Загальний вид директиви:

#pragma omp ім'я директиви *[*Додаткові параметри*]*.

Додаткові параметри залежать від конкретної директиви. Додаткові параметри не обов'язкові, задають додаткову інформацію для директиви.

Якщо параметри задаються, то відокремлюються друг від друга комами.

Якщо директива не міститься в одному рядку, для її продовження використовується символ \ (як для макросу *#define*).

Увага!!!

Якщо при завданні директиви опустити ключове слово *omp*, то директива просто ігнорується, і замість паралельного виконання одержимо код, що буде виконуватися послідовно.

ДИРЕКТИВИ ДЛЯ ВИЗНАЧЕННЯ ПАРАЛЕЛЬНИХ ДІЛЯНОК. ЗАГАЛЬНА ХАРАКТЕРИСТИКА

Містять директиви:

- ***Parallel, for, sections, section, task, simd***

Директива *parallel* використовується для ділянки програми, що повинна виконуватись багаторазово: стільки разів, скільки створене потоків. Задається на початку паралельної ділянки незалежно від завдання інших директив.

Директива *for* використовується для паралельного виконання тіла циклу з відомою кількістю повторень. Ітерація циклу виконується одним потоком. Якщо кількість потоків менше кількості повторень циклу, то кілька ітерацій циклу виконується одним потоком. Розподіл навантаження між потоками визначається параметрами.

Директива *sections* використовується для завдання декількох ділянок програми, які можна виконувати паралельно. Кількість ділянок зазвичай співпадає с кількістю ядер. Усередині повинна бути задана одна або кілька секцій (**директива *section***). Кожна секція може виконуватися паралельно. Фактично одна секція відповідає одній потоковій функції

Директива *task* аналогічна директиві ***section ,*** *але декілька задач може виконуватись одним потоком в залежності від кількості ядер. (версія 3.0. та вище)*

Директива ***simd*** – щоб показати, що паралельне виконання забезпечується використанням simd операцій

ДИРЕКТИВА *PARALLEL*

Визначає початок і кінець ділянки програми для багаторазового виконання.

Загальний вид директиви:

#pragma omp parallel [Параметри] {

    Блок операторів

}

Блок операторів виконується стільки разів, скільки потоків відповідає паралельній області.

Параметри:

- ***if*(Константний вираз);**
- ***num_threads*(Константний вираз);**
- *default*(shared | none);
- *private*(Список змінних);
- *firstprivate*(Список змінних);
- *shared*(Список змінних);
- *copyin*(Список змінних);
- *reduction* (Операція: Список змінних)
- proc_bind(master | close | spread) – для останніх версій (affinite)

ЗАГАЛЬНИЙ ВИД ФРАГМЕНТА ПРОГРАМИ З ПАРАЛЕЛЬНОЮ ДІЛЯНКОЮ

```
// Ділянка програми до паралельної секції
…
#pragma omp parallel
{
        Код для паралельного виконання
}
// Ділянка програми після паралельної секції
```

ПРИКЛАД НАЙПРОСТІШОГО ВИКОРИСТАННЯ ДИРЕКТИВИ

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

Код еквівалентний псевдокоду:

```
for (int i = 0; i < nthread; ++i) count++;
```

ВКЛЮЧЕННЯ – ВИКЛЮЧЕННЯ ПАРАЛЕЛЬНОГО ВИКОНАННЯ. ПАРАМЕТР *IF*

Вимикання. *Properties→ C/C++→Language* проекту виключити режим підтримки паралельного виконання (Open MP - off).

Недолік. Паралельне виконання відключається для всієї програми.

Як відключити тільки для ділянки програми???

Умовне відключення режиму паралельного виконання:

***if* (Вираз цілого типу)**

Вираз цілого типу = *true*, → паралельне виконання;

Вираз цілого типу = *false* → паралельного виконання немає.

#pragma omp parallel if (0)

паралельне виконання, задане директивою, вимикається.

***#pragma omp parallel if (1)*,**

паралельне виконання, задане директивою, вмикається.

За замовченням *if (1).*

Даний вираз може використовуватися при керування циклами, наприклад. якщо треба заборонити паралельне виконання циклу, якщо кількість повторень менше заданого значення.

СПОСОБИ ВИЗНАЧЕННЯ КІЛЬКОСТІ ПОТОКІВ

1. По замовченню.
2. Використання параметру *num_threads.*
3. Використання функцій бібліотеки OPEN MP.
4. Використання змінних середовища OMP_NUM_THREAD і OMP_DYNAMIC.

По замовченню – число потоків дорівнює кількості ядер процесора

ВИКОРИСТАННЯ ПАРАМЕТРУ NUM_THREADS

Загальний вид параметра:

num_threads (вираз цілого типу).

Значення виразу визначає кількість потоків для даної паралельної ділянки.

Приклад. Задати *кількість потоків* = 16, і перевірити правильність її завдання.

```
int count = 0;
#pragma omp parallel if (1) num_threads (16)
 {
      #pragma omp atomic
      count++;
 }
 printf("Number of threads: %d\n", count);
```

Програма повинна видати відповідь *Number of threads*:16.

Увага!

Встановлена кількість потоків діє тільки на задану директиву ***parallel***.

ВИКОРИСТАННЯ ФУНКЦІЙ БІБЛІОТЕКИ OPEN MP

Ім'я	Вхідні дані	Вихідні Дані	Коментар
omp_set_num_threads	int	-	Встановлює кількість потоків. *nthreads-var [0]*
omp_get_num_procs() *omp_get_num_threads*	-	int	Кількість доступних ядер Кількість потоків для даної паралельної області
omp_get_thread_num	-	int	Номер поточного потоку
omp_set_dynamic *omp_get_dynamic*	int	-	Якщо a != 0, то кількість потоків визначається як по замовченню

ГАРАНТОВАНЕ ВСТАНОВЛЕННЯ КІЛЬКОСТІ ПОТОКІВ

```
// Читаємо поточний режим
int dynamic = omp_get_dynamic();
// Встановлюємо потрібний режим, якщо
      необхідно
if (dynamic)
      omp_set_dynamic(0);
omp_set_num_threads(2);
…
// Повертаємо режим, який був с самого початку
omp_set_dynamic(dynamic);
```

ВИКОРИСТАННЯ ЗМІННИХ СЕРЕДОВИЩА OMP_NUM_THREAD І OMP_DYNAMIC

Встановлення змінної:

My computer → Properties → Advanced → Environment Variables → System variables і задаємо змінну зі своїм значенням (value).

Дія для OMP_NUM_THREAD:

If (value <0 || value >max)

реакція залежить від реалізації;

else

значення за замовчуванням = value;

Дія для OMP_DYNAMIC:

OMP_DYNAMIC=1 →nthreads = кількості ядер;

OMP_DYNAMIC=0 →nthreads залежить від установок;

ВИКОРИСТАННЯ ЗМІННИХ СЕРЕДОВИЩА. ПРИКЛАД

Нехай задане значення змінної середовища *OMP_NUM_THREADS* рівним 3.

Програма	Результат(кількість ядер=2)
#pragma omp parallel *{* *printf ("Hello\n");* *}*	Hello Hello Hello

ЗАГАЛЬНИЙ АЛГОРИТМ ВИЗНАЧЕННЯ КІЛЬКОСТІ ПОТОКІВ

```
if (omp_set_dynamic(1) задана || OMP_DYNAMIC ==1)
        кількість потоків дорівнює кількості ядер;
else {
        if (параметр num_threads (n))                          //(parallel)
            кількість потоків дорівнює n;
        else {
            if (omp_set_num_threads (m))
                        кількість потоків дорівнює m;
            else {
                        if (OMP_NUM_THREADS == k)
                                    кількість потоків дорівнює k;
                        else кількість потоків дорівнює = кількості ядер;
            }
        }
}
```

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛУ. ПОТОКОВА ФУНКЦІЯ

z[i] = x [i] + y [i] ***(int, 1000000)***

```
typedef struct {
        int Start, Finish, Step;
        // Вхідні дані;                    int *x, *y;
        // Вихідні дані                    int *z;
}DATAS, *PDATAS;
DWORD WINAPI ThreadFun (PVOID Par){
    PDATAS pDatas = (PDATAS) Par;
                                          int *x = pDatas->x, *y = pDatas->y, *z = pDatas->z;
    INT Start= pDatas ->Start, Finish= pDatas ->Finish, Step = pDatas ->Step;
    for (int i = Start; i < Finish; i+=Step){
        //          Тіло циклу            z[i] = x[i] + y[i];
    }
    return 0;
}
```

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛУ. ГОЛОВНА ПРОГРАМА

```
HANDLE hThread [16];
DATAS Datas [16 ];
nTthreads  = get_num_threads();
If (nTthreads > 16) nTthreads = 16;
Int V1 = …, V2 = …, nFor = (V2 - V1)/ Step;   V1 = 0,V2 = 1000000,nFor = V2;
int H = (nFor + nTthreads - 1) / nTthreads,     Step = 1;
for (i = 0; i < nTthreads; i++)
{
    Datas [i].Start = V1 + i * H;
    Datas [i].Finish = Datas [i].Start + H;
    Datas [i].Step = Step;
    if (i != nTthreads - 1)
    hThread [i] = CreateThread (0, 0, ThreadFun, &Datas [i], 0, 0);
}
ThreadFun (&Datas [nTthreads - 1]);
WaitForMultipleObjects (nTthreads - 1, hThread, true, INFINITE);
```

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛУ. OPEN MP

Директива: *#pragma omp for*

Загальний вид директиви *for:*

#pragma omp for [Параметри] .
for (…){}

Структура програми з паралельним циклом:

```
#pragma omp parallel[Параметри для parallel]
{   …
    #pragma omp for[Параметри для for]
    for (....)
    {…}
    …
}
```

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)

ordered

nowait

lastprivate(Список змінних)

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛУ. ПРИКЛАД (4 ЯДРА)

```
#pragma omp parallel
{
printf("Begin Thread num %d\n",
            omp_get_thread_num());
#pragma omp for
for (int i = 0; i < 10; i++){
            printf("i = %d Thread num = %d\n", i,
            omp_get_thread_num());
}
printf("End Thread num %d\n",
omp_get_thread_num());
}
```

Begin Thread num 0
Begin Thread num 2
i = 6 thread num = 2
i = 7 thread num = 2
i = 0 thread num = 0
i = 1 thread num = 0
i = 2 thread num = 0
Begin Thread num 1
i = 3 thread num = 1
i = 4 thread num = 1
i = 5 thread num = 1
Begin Thread num 3
i = 8 thread num = 3
i = 9 thread num = 3
End Thread num 0
End Thread num 3
End Thread num 2
End Thread num 1

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛУ. СПРОЩЕНИЙ ВАРІАНТ. ПРИКЛАД

Скорочена форму визначення паралельних циклів:

#pragma omp parallel for[Додаткова інформація]
for (....) {…}

Приклад використання. Обчислити z[i] = x[i] y [i] без паралельного виконання, за допомогою AVX та потоків*

Enable Parallel Code Generation (Yes /Qpar)

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛУ. СПРОЩЕНИЙ ВАРІАНТ

```
void Fun1(const float *a, const float *b, float *c, int n){
    for (int i = 0; i< n; i++)
        c[i] = a[i] * b[i];
}
void Fun2(const float *a, const  float *b, float *c, int n){
    __m256 *pa = (__m256 *)a, *pb = (__m256 *)b, *pc = (__m256 *)c;
    for (int i = 0; i < n / 8; i++){
        pc [i] = _mm256_mul_ps (pa [i], pb [i]);
    }
}
void Fun3(const float *a, const  float *b, float *c, int n){
    #pragma omp parallel for
        for (int i = 0; i < n ; i++)
            c[i] = a[i] * b[i];
} time (0.0026, 0.0018, 0.0020)        SSE      AVX      Threads
```

ПОМИЛКИ, ЯКІ ЧАСТО ДОПУСКАЮТЬ ПРИ ВИКОРИСТАННІ FOR

```
#pragma omp for

for (int i = 0; i < n; i++)
        z[i] = x[i]*y[i];
```

ПОМИЛКИ, ЯКІ ЧАСТО ДОПУСКАЮТЬ ПРИ ВИКОРИСТАННІ FOR

```
#pragma omp for

for (int i = 0; i < n; i++)
        z[i] = x[i]*y[i];
```

```
#pragma omp parallel
{
#pragma omp for
for (int i = 0; i < n; i++)
        z[i] = x[i]*y[i];
}
```

ПОМИЛКИ, ЯКІ ЧАСТО ДОПУСКАЮТЬ ПРИ ВИКОРИСТАННІ FOR

```
#pragma omp parallel
{
        for (int i = 0; i < n; i++)
                z[i] = x[i]*y[i];
}
```

ПОМИЛКИ, ЯКІ ЧАСТО ДОПУСКАЮТЬ ПРИ ВИКОРИСТАННІ FOR

```
#pragma omp parallel
{
        for (int i = 0; i < n; i++)
                z[i] = x[i]*y[i];
}
```

```
#pragma omp parallel for

        for (int i = 0; i < n; i++)
                z[i] = x[i]*y[i];
```

ПОМИЛКИ, ЯКІ ЧАСТО ДОПУСКАЮТЬ ПРИ ВИКОРИСТАННІ FOR

```
#pragma omp parallel
{
        #pragma omp parallel for
        for (int i = 0; i < n; i++)
                z[i] = x[i]*y[i];
}
```

ПОМИЛКИ, ЯКІ ЧАСТО ДОПУСКАЮТЬ ПРИ ВИКОРИСТАННІ FOR

```
#pragma omp parallel
{
        #pragma omp parallel for
        for (int i = 0; i < n; i++)
                z[i] = x[i]*y[i];
}
```

```
#pragma omp parallel    for

        for (int i = 0; i < n; i++)
                z[i] = x[i]*y[i];
```

Багато таких прикладів:

https://www.viva64.com/ru/a/0054

ОБМЕЖЕННЯ НА ЦИКЛИ ДЛЯ ПАРАЛЕЛЬНОГО ВИКОНАННЯ

- Допускається тільки заголовок типу ***for***, цикли не можуть мати заголовки типу ***while, do***.
- У заголовку повинні бути задані всі 3 вирази: ***v1*** - початкове значення параметра циклу; ***v2*** - вираз для перевірки необхідності виконання циклу; ***v3*** - вираз для зміни параметра циклу. Це пов'язане з тим, що необхідно задати вираз для визначення кількості циклів.
- ***v1* повинне мати вигляд: *Змінна = Вираз***, де *Змінна* повинна бути змінної цілого типу зі знаком або без, може бути покажчиком на ці типи. Однак версія 2 допускає тільки знакові змінні. *Вираз* повинен бути цілого типу, його значення не може залежати від параметра циклу.
- ***v2* повинне використовувати знаки відносин**: <, <=, >, >=, інші знаки відносин, а також їхня відсутність не допускається. Задається у вигляді *Змінна Знак відносини Вираз*.
- ***v3* повинне збільшувати або зменшувати параметр** циклу за допомогою операцій: ++, --, +=, -=, +, -. Після знака дорівнює повинен бути *вираз*, що не залежить від параметра циклу.
- У тілі циклу **параметр циклу змінюватися не може**.
- Після завершення циклу **параметр циклу не визначений**.
- Не можна в тілі циклу використовувати ***break*, *continue***.

Розгляньте ці обмеження та спробуйте їх пояснити!!!

ЗАЛЕЖНОСТІ І ЦИКЛИ

Увага! Відсутність залежностей повинна бути забезпечена програмістом!

Якщо цикл вміщує залежну та незалежну частину:

```
for (int i = 0; i < n; ++i){
    //залежна частина
     // незалежна частина
}
```

→

```
for (int i= 1; i <n;i++) {
        a[i]= 2*a[i-1]+b[i];
        b [i] =a[i] + c [i-1];

}
```

Правильне рішення

```
for (int i= 1; i <n;i++) {
     a [i] =2*a[i-1]+b [i];
#pragma omp parallel for
for (int i= 1; i <n;i++)
      b [i] = a [i] + c [i-1];
```

ЗАЛЕЖНОСТІ І ЦИКЛИ

Приклад 2
Чи можна паралельно виконувати цикл?

```
for (i = 0; i < n; ++i) {
        x = fun1 (a [2 * i]);
        a [2 * I + 1] = fun2 (x * x, c [2 * i] + 1);
}
```

Виправлення:

```
#pragma omp parallel for
for (i = 0; i < n; ++i) {
    int x;
    x = fun1 (a [2 * i]);
    a [2 * i + 1] = fun2 (x * x, c [2 * i] + 1);
}
```

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛІВ І ФУНКЦІЯ RAND

Код для ініціалізації матриці:

```
void InitMatrix(int** matr, int n, int m, int module){
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < n; ++i){
            for (int j = 0; j < m; ++j){
                matr[i][j] = rand()%module;
            }
        }
    }
}
…
InitMatrix(matr, 4, 4, 20);
```

Який результат ви очикуєте ???

ПАРАЛЕЛЬНЕ ВИКОНАННЯ ЦИКЛІВ І ФУНКЦІЯ RAND

Код для ініціалізації матриці:

```
#include <intrin.h>
void InitMatrix1(int** matr, int n, int m, int module){
    #pragma omp parallel
    {
        srand(__rdtsc() ^ omp_get_thread_num());
        #pragma omp for
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < m; ++j)
                matr[i][j] = rand() % module;
    }
}
```

Правильний варіант!!!

НАКОПИЧЕННЯ ЗНАЧЕНЬ

s += *a[i]* ; *p* *= *a[i]*;

reduction

Загальний вид параметра ***reduction***:

reduction **(Операція: Список змінних)**

Змінні тільки прості, без індексів, не показники, не посилання

Знаки операції	Операція	Початкове значення
+, -	Складання, віднімання	0
*	Множення	1
&	Побітове множення	~0 (1 у всіх розрядах)
|	Побітове додавання	0
^	Додавання по модулі. 2	0
&&	Логічне множення	1
||	Логічне додавання	0

ПРИКЛАД ЦИКЛІВ З НАКОПИЧЕННЯМ

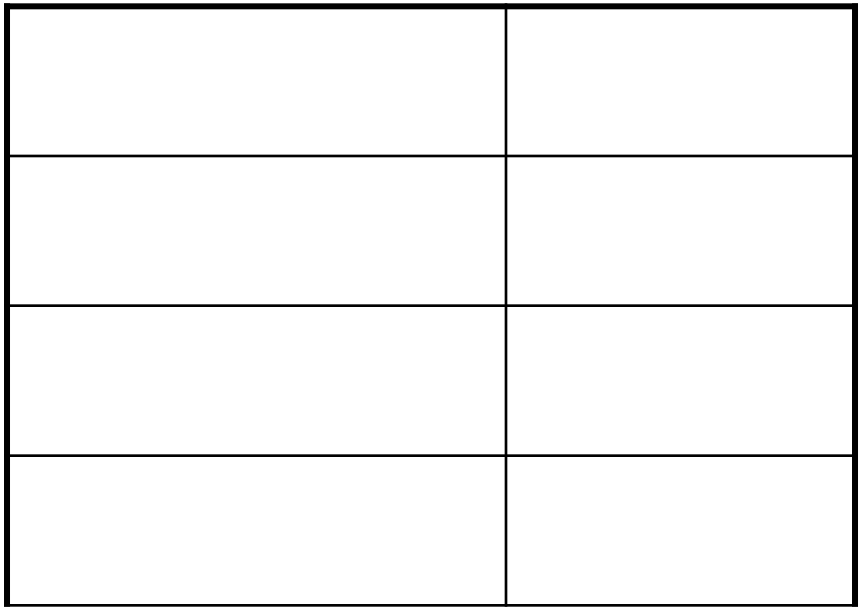
Реалізувати обчислення скалярного добутку

Розглянути варіанти:

Звичайна функція

Використання atomic

Використання reduction



ПРИКЛАД ЦИКЛІВ З НАКОПИЧЕННЯМ

Реалізувати обчислення скалярного добутку

```
float SecSums(float *x, float *y, int n){
        float s = 0;
        for (int i = 0; i < n; i++) s+= x [i] * y [i];
        return s;
}
float OpenMPSums1(float *x, float *y, int n){
        float s = 0;
        #pragma omp parallel for
                for (int i = 0; i < n; i ++)
                #pragma omp atomic
                        s+= x [i] * y [i];

        return s;
}
float OpenMPSums2(float *x, float *y, int n){
        float s = 0;
        #pragma omp parallel for reduction (+ : s)
        for (int i = 0; i < n; i ++) s+= x [i] * y [i];
        return s;
}
```

Ім'я функції	Час (ms)
SecSums	5.5
OpenMPSums1	80.75
OpenMPSums2	3.67

РОЗПОДІЛ НАВАНТАЖЕННЯ МІЖ ПОТОКАМИ (ПО ЗАМОВЧЕННЮ)

```
a = 3;
#pragma omp parallel for num_threads (4)
        for (int i = 0; i < a * a; i++) { int b = i + i;
             printf ("thread # = %d\ti = %d\tb= %d\n",
            omp_get_thread_num (), i, b);
        }
```

thread # = 2	i = 5	b= 10	thread #0 : 0, 1, 2
thread # = 1	i = 3	b= 6	thread #1 : 3, 4
thread # = 2	i = 6	b= 12	thread #2 : 5, 6
thread # = 1	i = 4	b= 8	thread #3 : 7, 8
thread # = 0	i = 0	b= 0	
thread # = 0	i = 1	b= 2	
thread # = 3	i = 7	b= 14	
thread # = 0	i = 2	b= 4	
thread # = 3	i = 8	b= 16	

КЕРУВАННЯ РОЗПОДІЛОМ НАВАНТАЖЕННЯ

Параметр *schedule*

Загальний вид параметра:

***schedule*(Спосіб[, Розмір]),**

де:

Спосіб – задає спосіб розподілу навантаження;

Розмір - задає кількість ітерацій циклу для одного потоку.

Спосіб може приймати значення:

***static*, *dynamic*, *guided*, *auto, runtime*.**

КЕРУВАННЯ РОЗПОДІЛОМ НАВАНТАЖЕННЯ. Static

***schedule*(static) - default**

***schedule*(static, 4)**

thread # = 1 i = 4 b= 8

thread # = 1 i = 5 b= 10

thread # = 2 i = 8 b= 16

thread # = 0 i = 0 b= 0

thread # = 1 i = 6 b= 12

thread # = 0 i = 1 b= 2

thread # = 1 i = 7 b= 14

thread # = 0 i = 2 b= 4

thread # = 0 i = 3 b= 6

КЕРУВАННЯ РОЗПОДІЛОМ НАВАНТАЖЕННЯ. Static

***schedule*(static) - default**

***schedule*(static, 4)**

thread # = 1 i = 4 b= 8

thread # = 1 i = 5 b= 10

thread # = 2 i = 8 b= 16

thread # = 0 i = 0 b= 0

thread # = 1 i = 6 b= 12

thread # = 0 i = 1 b= 2

thread # = 1 i = 7 b= 14

thread # = 0 i = 2 b= 4

thread # = 0 i = 3 b= 6

КЕРУВАННЯ РОЗПОДІЛОМ НАВАНТАЖЕННЯ. Static

***schedule*(dynamic)** **_schedule_(dynamic, 4)**

***schedule*(dynamic)**		***schedule*(dynamic, 4)**	
thr # = 1 i = 0	b= 0	thr# = 3 i = 0	b= 0
thr# = 1 i = 2	b= 4	thr# = 3 i = 1	b= 2
thr# = 0 i = 3	b= 6	thr# = 3 i = 2	b= 4
thr# = 2 i = 1	b= 2	thr# = 2 i = 4	b= 8
thr# = 0 i = 5	b= 10	thr# = 3 i = 3	b= 6
thr# = 1 i = 4	b= 8	thr# = 2 i = 5	b= 10
thr# = 2 i = 6	b= 12	thr# = 2 i = 6	b= 12
thr# = 1 i = 8	b= 16	thr# = 2 i = 7	b= 14
thr # = 0 i = 7	b= 14	thr # = 3 i = 8	b= 16

КЕРУВАННЯ РОЗПОДІЛОМ НАВАНТАЖЕННЯ. Static

***schedule*(guide)** ***schedule*(guide, 4)**

***schedule*(guide)**			***schedule*(guide, 4)**		
# = 0	i = 0	b= 0	thr# = 0	i = 0	b= 0
thr# = 2	i = 5	b= 10	thr# = 0	i = 1	b= 2
thr# = 0	i = 1	b= 2	thr# = 0	i = 2	b= 4
thr# = 0	i = 2	b= 4	thr# = 0	i = 3	b= 6
thr# = 2	i = 6	b= 12	thr# = 0	i = 4	b= 8
thr# = 0	i = 7	b= 14	thr# = 3	i = 8	b= 16
thr# = 2	i = 8	b= 16	thr# = 0	i = 5	b= 10
thr# = 3	i = 3	b= 6	thr# = 0	i = 6	b= 12
thr # = 3	i = 4	b= 8	thr # = 0	i = 7	b= 14

РЕКОМЕНДАЦІЇ З ВИКОРИСТАННЯ РЕЖИМІВ КЕРУВАННЯ НАВАНТАЖЕННЯМ

- **Якщо ітерації виконуються приблизно однаковий час**, то можна обирати статичний спосіб розподілу навантаження без завдання поля розміру. Якщо кількість ітерацій на один потік може бути дуже малою, використовується цей режим з полем розміру.
- **Якщо час виконання ітерацій зменшується (збільшується) зі зміною номера ітерації**, то краще вибрати динамічний режим розподілу навантаження. Поле розміру необхідно обирати з урахуванням гранулярності.
- **Зменшення кількості ітерацій для наступного виконання** (режим *guided*), найчастіше ефективніше, ніж *static*, але це потребує додаткової перевірки для конкретної задачі.

ДОДАТКОВІ ПАРАМЕТРИ ДЛЯ ЦИКЛІВ. ПАРАМЕТР COLLAPSE

Хай треба виконати цикл:

```
for (int i = 0; i < 5; i++)
        for (int j = 0; j < 1000000; j++)
            z [i] = f (x [i] , y [j]);
```

Хай кожна ітерація зовнішнього циклу виконується 1 час. Хай в наявності 4 логічних ядра

```
#pragma omp parallel for
for (int i = 0; i < 5; i++)
        for (int j = 0; j < 1000000; j++)
            z [i] = f (x [i] , y [j]);
```

Час виконання 2 часа

```
for (int i = 0; i < 5; i++){
{
        #pragma omp parallel for
         for (int j = 0; j < 1000000; j++)
                z [i] = f (x [i] , y [j]);

}
```

Накладні витрати збільшуються в 5 разів

```
#pragma omp parallel for collapse (2)
for (int i = 0; i < 5; i++)
        for (int j = 0; j < 1000000; j++)
            z [i] = f (x [i] , y [j]);
```

Кількість ітерацій на один потік :5 * 1000000/4

В 2 версії немає. Що можна зробити?

1 Змінити порядок циклів.

2 Замінити 2 цикла одним `for (int i = 0; i < 5000000; i++){`

```
        int ii = i / 1000000,  jj = i% 1000000;
        z [ii] = f (x [ii] , y [ji]; }
```

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] = 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.

Приклад:

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

Статичні змінні. Визначаються поза функціями або всередині 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){
                …  ;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;
}
```

ВИСНОВКИ

1. Необхідними умовами паралельного виконання циклу є незалежність його ітерацій, тобто результат не залежить від порядку їх виконання.
2. Open MP “вміє” паралельно виконувати тільки цикли для який може визначити кількість повторення до виконання циклу.
3. Різні способи керування розподілом навантаження дозволяють найбільш ефективно врахувати кількість ядер процесору.
4. Вибір способу розподілу навантаження залежить від особливостей тіла циклу, а саме часу виконання окремих його ітерацій.
5. Використання для паралельного виконання циклу Windows потоків замість Open MP не дає переваги в часі виконання, але значно трудніше та код залежить від наявності та відсутності потоків.
6. Секції використовуються, якщо замість циклу треба паралельно виконати окремі ділянки кожу.
7. Далі будуть розглянуті класи пам'яті в Open MP

ПИТАННЯ ДЛЯ САМОСТІЙНОГО ВИВЧЕННЯ

1. Вкладені цикли, параметр *collapse*.
2. Спосіб розподілу навантаження *auto*.

Учбовий посібник: “Паралельне програмування”

МАТЕРІАЛИ ДЛЯ ЕКСПРЕС-КОНТРОЛЮ

- В якому випадку можна забезпечити паралельне виконання циклу без директиви for?
- Чому Open MP “вміє” паралельно виконувати тільки цикли з відомою кількістю повторень?
- Що буде, якщо цикл з залежними ітераціями виконувати паралельно за допомогою Open MP?
- Чи є помилка в коді:

  #pragma omp for

  for (int i = 0; i < n; ++i){...} ?
- В якому випадку рекомендується використовувати статичний спосіб розподілу навантаження, динамічний без розміру та з розміром, guided?
- Яким чином забезпечити паралельне виконання внутрішніх циклів?
- Яким чином можна виконувати паралельно функції за допомогою Open MP?
- Які недоліки використання секцій для організації паралельних обчислень?