Лабораторна робота №2
Розподіл роботи

Паралельне програмування за допомогою OpenMP. Засоби розподілу роботи між потоками.

Мета: Вивчити засоби розподілу роботи між потоками і навчитися їх застосовувати.

1.1. Низькорівневе розпаралелювання

Всі потоки в паралельній області нумеруються послідовними цілими числами від 0 до N-1, де N — кількість потоків, що виконують дану область.

Можна програмувати на найнижчому рівні, розподіляючи роботу за допомогою функцій **omp_get_thread_num()** і **omp_get_num_threads()**, що повертають номер потоку і загальну кількість породжених потоків в поточній паралельній області, відповідно.

Виклик функції **omp_get_thread_num()** дозволяє потоку отримати свій унікальний номер поточної паралельної області:

int omp_get_thread_num(void);

Виклик функції **omp_get_num_threads()** дозволяє потоку отримати кількість потоків поточної паралельної області:

int omp_get_num_threads(void);

Приклад демонструє роботу функцій **omp_get_num_threads()** і **omp_get_thread_num().** Потік, порядковий номер якого дорівнює 0, надрукує загальну кількість породжених потоків, а інші потоки надрукують свій порядковий номер.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
      int count, num;
      #pragma omp parallel
      {
            count=omp_get_num_threads();
            num=omp_get_thread_num();
            if (num == 0) printf("Усього потоків: %d\n", count);
            else printf("Потік номер %d\n", num);
      }
}
```

Використання функцій **omp_get_thread_num()** і **omp_get_num_threads()** дозволяє призначати кожному потоку свій шматок коду для виконання, і таким чином розподіляти роботу між потоками. Проте використання цього стилю програмування в OpenMP далеко не завжди виправдане – програміст в цьому випадку повинен явно організовувати синхронізацію доступу до загальних даних. Інші способи розподілу робіт в OpenMP забезпечують значну частину цієї роботи автоматично.

1.2. Паралельні цикли

Якщо в паралельній області зустрівся оператор циклу, то, згідно із загальним правилом, він буде виконаний всіма потоками поточної групи, тобто кожен потік виконає всі ітерації даного циклу. Для розподілу ітерацій циклу між різними потоками можна використовувати директиву **for**.

#pragma omp for [опція **[[,]** опція**]...]**

Можливі опції:

- **private**(список) – задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних із списку не визначене;
- **firstprivate**(список) – задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізувалися значеннями цих змінних в потоці-майстрові;
- **lastprivate**(список) – змінним, перерахованим в списку, привласнюється результат з останнього витка циклу;
- **reduction**(оператор:список) – задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; над локальними копіями змінних після завершення всіх ітерацій циклу виконується заданий оператор; оператор для мови Сі – +, *, - &, |, ^, &&;
- **schedule(type[, chunk])** – опція задає, яким чином ітерації циклу розподіляються між потоками;
- **collapse(n)** — опція вказує, що n послідовних тісновложених циклів асоціюється з даною директивою; для циклів створюється загальний простір ітерацій, який ділиться між потоками; якщо опція **collapse** не задана, то директива відноситься лише до одного безпосередньо наступного за нею циклу;
- **ordered** – опція, що говорить про те, що в циклі можуть зустрічатися директиви **ordered**; в цьому випадку визначається блок усередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть в послідовному циклі;
- **nowait** – в кінці паралельного циклу відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається лише тоді, коли всі вони досягнуть даної крапки; якщо в подібній затримці немає необхідності, опція **nowait** дозволяє потокам, що вже дійшли до кінця циклу, продовжити виконання без синхронізації з останніми.

На вигляд паралельних циклів накладаються досить жорсткі обмеження. Не можна використовувати побічний вихід з паралельного циклу. Розмір блоку ітерацій, вказаний в опції **schedule**, не повинен змінюватися в рамках циклу.

Формат паралельних циклів на мові С спрощено можна представити таким чином:

for([цілочисельний тип**] i =** інваріант циклу**;**
i {<,>,=,<=,>=} інваріант циклу**;**
i {+,-}= інваріант циклу**)**

Ці вимоги введені для того, щоб OpenMP міг при вході в цикл точно визначити число ітерацій.

Якщо директива паралельного виконання стоїть перед гніздом циклів, що завершуються одним оператором, то директива діє лише на самий зовнішній цикл.

Ітеративна змінна розподілюваного циклу по сенсу має бути локальною, тому у випадку, якщо вона специфікована загальною, то вона неявно робиться локальною при вході в цикл. Після завершення циклу значення ітеративної змінної циклу не визначене, якщо вона не вказана в опції **lastprivate**.

Приклад демонструє використання директиви **for**. У послідовній області ініціалізувалися три вихідні масиви A, B, C. У паралельної області дані масиви оголошені загальними. Допоміжні змінні i та n оголошені локальними. Кожен потік привласнить змінній n свій порядковий номер. Далі за допомогою директиви **for** визначається цикл, ітерації якого будуть розподілені між існуючими потоками. На кожній i-й ітерації даний цикл складе i-і елементи масивів A і B і результат запише в i-й елемент масиву C. Також на кожній ітерації буде надрукований номер потоку, що виконав дану ітерацію.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
     int A[10], B[10], C[10], i, n;
     for (i=0; i<10; i++){ A[i]=i; B[i]=2*i; C[i]=0; }
     #pragma omp parallel shared(A, B, C) private(i, n)
     {
          n=omp_get_thread_num();
          #pragma omp for
          for (i=0; i<10; i++)
          {
            C[i]=A[i]+B[i];
           printf("Нитка %d склала елементи з номером %d\n", n, i);
           }
     }
}
```

У опції **schedule** параметр **type** задає наступний тип розподілу ітерацій:

- **static** – блоково-циклічний розподіл ітерацій циклу; розмір блоку – **chunk**. Перший блок з **chunk** ітерацій виконує нульовий потік, другий блок — наступний і так далі до останнього потоку, потім розподіл знову починається з нульового потоку. Якщо значення **chunk** не вказане, то вся безліч ітерацій ділиться на безперервні шматки приблизно однакового розміру (конкретний чин залежить від реалізації), і отримані порції ітерацій розподіляються між потоками.
- **dynamic** – динамічний розподіл ітерацій з фіксованим розміром блоку: спочатку кожен потік отримує **chunk** ітерацій (за умовчанням **chunk**=1), потім потік, який закінчує виконання своєї порції ітерацій, отримує першу вільну порцію з **chunk** ітерацій. Потоки, що звільнилися, отримують нові порції ітерацій до тих пір, поки всі порції не будуть вичерпані. Остання порція може містити менше ітерацій, чим всі інші..
- **guided** – динамічний розподіл ітерацій, при якому розмір порції зменшується з деякого початкового значення до величини **chunk** (за умовчанням **chunk**=1) пропорційно кількості ще не розподілених ітерацій, що ділиться на кількість потоків, що виконують цикл. Розмір блоку, що спочатку виділяється, залежить від реалізації. У ряді випадків такий розподіл дозволяє акуратніше розділити роботу і збалансувати завантаження птоків. Кількість ітерацій в останній порції може виявитися менше значення **chunk**.
- **auto** – спосіб розподілу ітерацій вибирається компілятором і системою

виконання. Параметр **chunk** при цьому не задається.

- **runtime** – спосіб розподілу ітерацій вибирається під час роботи програми за значенням змінного середовища **Omp_schedule**. Параметр **chunk** при цьому не задається.

Приклад демонструє використання опції **schedule** з параметрами (**static**) (**static**, 1) (**static**, 2) (**dynamic**) (**dynamic**, 2) (**guided**) (**guided**, 2). У паралельній області виконується цикл, ітерації якого будуть розподілені між існуючими потоками. На кожній ітерації буде надруковано, який потік виконав дану ітерацію. У тіло циклу вставлена також затримка, що імітує деякі обчислення..

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
      int i;
#pragma omp parallel private(i)
      {
      #pragma omp for schedule (static)
      //#pragma omp for schedule (static, 1)
      //#pragma omp for schedule (static, 2)
      //#pragma omp for schedule (dynamic)
      //#pragma omp for schedule (dynamic, 2)
      //#pragma omp for schedule (guided)
      //#pragma omp for schedule (guided, 2)
         For (i=0; i<10; i++)

           {
            printf("Потік %d виконав
            ітерацію%d\n",omp_get_thread_num(), i);
            sleep(1);

          }
      }
 }
```

Результати виконання прикладу з різними типами розподілу ітерацій приведені в таблиці нижче. Стовпці відповідають різним типам розподілів, а рядки – номеру ітерації. У таблиці вказані номера потоків, що виконували відповідну ітерацію. У всіх випадках для виконання паралельного циклу використовувалися 4 потоки. Для динамічних способів розподілу ітерацій (**dynamic**, **guided**) конкретне ділення між потоками може відрізнятися від запуску до запуску.

Таблиця 1. Розподіл ітерацій по нитках.

i	static	static, 1	static, 2	dynamic	dynamic, 2	guided	guided, 2
0	0	0	0	0	0	0	0
1	0	1	0	1	0	2	0
2	0	2	1	2	1	1	1
3	1	3	1	3	1	3	1
4	1	0	2	1	2	1	2
5	1	1	2	3	2	2	2
6	2	2	3	2	3	3	3
7	2	3	3	0	3	0	3
8	3	0	0	1	3	0	0
9	3	1	0	0	3	3	0

У таблиці 1 видно різниця між розподілом ітерацій при використанні різних варіантів. До найбільшого дисбалансу привели варіанти розподілу (**static**, 2), (**dynamic**, 2) і (**guided**, 2). У всіх цих випадках одному з потоків дістається на дві ітерації більше, ніж іншим. У інших випадках ця різниця декілька згладжується.

1.3. Паралельні секції

Директива **sections** (**sections** ... **end sections**) використовується для завдання кінцевого (неітеративного) паралелізму:

#pragma omp sections [опція **[[,]** опція**]...]**

Ця директива визначає набір незалежних секцій коду, кожна з яких виконується своєю ниткою.

Можливі опції:

- **private**(список) – задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних із списку не визначене;
- **firstprivate**(список) – задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізувалися значеннями цих змінних в потоці-майстрові;
- **lastprivate**(список) – змінним, перерахованим в списку, привласнюється результат з останнього витка циклу;
- **reduction**(оператор:список) – задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; над локальними копіями змінних після завершення всіх ітерацій циклу виконується заданий оператор; оператор для мови C – +, *, - &, |, ^, &&;

• **nowait** – в кінці паралельного циклу відбувається неявна бар'єрна синхронізація паралельно працюючих потоків: їх подальше виконання відбувається лише тоді, коли всі вони досягнуть даної точки; якщо в подібній затримці немає необхідності, опція **nowait** дозволяє потокам, що вже дійшли до кінця циклу, продовжити виконання без синхронізації з останніми.

Директива **section** задає ділянку коду усередині секції **sections** для виконання одним потоком.

#pragma omp section

Перед першою ділянкою коду в блоці **sections** директива **section** не обов'язкова. Які саме потоки будуть задіяні для виконання будь-якої секції, не специфікується. Якщо кількість потоків більше кількості секцій, то частина потоків для виконання даного блоку секцій не буде задіяна. Якщо кількість потоків менше кількості секцій, то деяким (або всім) потокам дістанеться більше за одну секцію.

Приклад ілюструє вживання директиви **sections**. Спочатку три потоки, на яких розподілилися три секції **section**, виведуть повідомлення зі своїм номером, а потім всі потоки надрукують однакове повідомлення зі своїм номером.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
      int n;
     #pragma omp parallel private(n)
      {
           n=omp_get_thread_num();
        #pragma omp sections
         {
            #pragma omp section
            {
               printf("Перша секція, процес %d\n", n);
            }
            #pragma omp section
            {
                printf("Друга секція, процес %d\n", n);
            }
            #pragma omp section
            {
                printf("Третя секція, процес %d\n", n);
            }

         }
        printf("Паралельна область, процес %d\n", n);
    }
}
```

Приклад демонструє використання опції **lastprivate**. У даному прикладі опція **lastprivate** використовується разом з директивою **sections**. Змінна n оголошена як **lastprivate** змінна. Три потоки, що виконують секції **section**, привласнюють своїй локальній копії n різні значення. Після виходу з області **sections** значення n з останньої

секції привласнюється локальним копіям у всіх нитках, тому всі нитки надрукують число 3. Це ж значення збережеться для змінної n і в послідовній області.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
     int n=0;
    #pragma omp parallel
     {
       #pragma omp sections lastprivate(n)
       {
          #pragma omp section
          {
             n=1;
          }
          #pragma omp section
          {
             n=2;
          }
          #pragma omp section
          {
             n=3;
             }
       }
       printf("Значення n на потоці %d: %d\n", omp_get_thread_num(), n);
     }
    printf("Значення n в послідовній області: %d\n", n);
}
```

Хід роботи

Створити програму яка повинна реалізувати наступні дії:

1.Створити квадратні матриці A та B розміром n * n, елементи яких заповнюються рандомно, n задає кількість рядків і кількість потоків, які виконуватимуть паралельну область програми. Змінні можуть задаватися в програмному коді або вводитися з клавіатури.

2. У паралельній області за допомогою директиви single/master вивести наступні дані: номер лабораторної роботи; назва лабораторної роботи; групу студента; ФІО студента; номер варіанту; завдання.

3.У кожній матриці окремо обчислити паралельним способом вираження відповідно до свого варіанту. Розподіл ітерацій між потоками виконати за допомогою директиви **for** з використанням опції **schedule**. Для кожної з матриць використати різні значення параметру **type** та розмірі блоку – **chunk.** Результати обробки записати в масиви C та D.

4.Вивести результати обробки матриць паралельним способом, вказавши при виводі розподіл ітерацій по потоках для різних значень опції **schedule**. Порівняти результати, отримані при різних значеннях параметру **type** та розмірі блоку – **chunk.**

№ варіанта	**Завдання**	**Значення параметрів type та chunk**
1	Знайти мінімальне значення кожного рядка матриці	(static, 4) та (dynamic, 4)
2	Знайти максимальне значення кожного рядка матриці	(dynamic, 6) та (guided, 6)
3	Порахувати кількість додатних елементів в кожному рядку матриці	(static, 2) та (guided, 2)
4	Порахувати кількість від'ємних елементів в кожному рядку матриці	(static, 6) та (dynamic, 4)
5	Порахувати кількість нульових елементів в кожному рядку матриці	(dynamic, 4) та (guided, 6)
6	Порахувати суму елементів в кожному рядку матриці	(static, 8) та (guided, 8)
7	Порахувати суму додатних елементів в кожному рядку матриці	(dynamic, 6) та (guided, 8)
8	Порахувати суму від'ємних елементів в кожному рядку матриці	(static, 1) та (dynamic, 4)

Контрольні питання

1. Чи можуть функції **omp_get_thread_num()** і **omp_get_num_threads()** повернути однакові значення на декількох нитках однієї паралельної області?
2. Чи можна розподілити між нитками ітерації циклу без використання директиви **for**?
3. Чи можна однією директивою розподілити між нитками ітерації відразу декількох циклів?
4. Чи можливо, що при статичному розподілі ітерацій циклу ниткам дістанеться різна кількість ітерацій?
5. Чи можуть при повторному запуску програми ітерації розподілюваного циклу

дістатися іншим ниткам? Якщо так, то при яких способах розподілу ітерацій?

6. Для чого може бути корисно вказувати параметр **chunk** при способі розподілу ітерацій **guided**?
7. Чи можна реалізувати паралельні секції без використання директив **sections** (**sections** ... **end sections**) і **section**?
8. Як при виході з паралельних секцій розіслати значення деякої локальної змінної всім ниткам, що виконують дану паралельну область?