Лабораторна робота №3

Паралельне програмування за допомогою OpenMP. Засоби синхронізації.

Мета: Вивчити засоби синхронізації доступу до даних і навчитися їх застосовувати.

1.1. Бар'єр

Найпоширеніший спосіб синхронізації в OpenP – бар'єр. Він оформляється за допомогою директиви **barrier**.

#pragma omp barrier

Потоки, що виконують поточну паралельну область, дійшовши до цієї директиви, зупиняються і чекають, поки всі потоки не дійдуть до цієї точки програми, після чого розблоковуються і продовжують працювати далі

Приклад демонструє вживання директиви **barrier**. Директива **barrier** використовується для впорядковування виводу від працюючих потоків. Видачі з різних потоків "Повідомлення 1" і "Повідомлення 2" можуть перемежатися в довільному порядку, а видача "Повідомлення 3" зі всіх потоків прийде строго після двох попередніх видач.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char
{
#pragma omp parallel
      {
         printf("Повідомлення 1\n");
         printf("Повідомлення 2\n");
           #pragma omp barrier
         printf("Повідомлення 3\n");
      }
}
```

1.2.Директива ordered

Директиви **ordered** (**ordered** ... **end ordered**) визначають блок усередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть в послідовному циклі.

#pragma omp ordered

Блок операторів відноситься до самого внутрішнього з охоплюючих циклів, а в паралельному циклі має бути задана опція **ordered**. Потік, що виконує першу ітерацію циклу, виконує операції даного блоку. Потік, що виконує будь-яку наступну ітерацію, повинен спочатку діждатися виконання всіх операцій блоку всіма потоками, що виконують попередні ітерації. Може використовуватися, наприклад, для впорядкування виводу від паралельних потоків.

Приклад ілюструє вживання директиви **ordered** і опції **ordered**. Цикл **for** помічений як **ordered**. Усередині тіла циклу йдуть дві видачі – одна поза блоком **ordered**, а друга –

усередині нього. В результаті перша видача виходить невпорядкованою, а друга йде в строгому порядку за збільшенням номеру ітерації.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
      int i, n;
#pragma omp parallel private (i, n)
      {
            n=omp_get_thread_num();
            #pragma omp for ordered
            for (i=0;<5; i++)
            {
                  printf("Потік %d, ітерація %d\n", n, i);
               #pragma omp ordered
                           {
                            printf("ordered: Нитка %d, ітерація %d\n", n, i);
                           }
            }
      }
}
```

1.3. Критичні секції

За допомогою директив **critical** (**critical** ... **end critical**) оформляється критична секція програми.

#pragma omp critical [(<ім'я_критичної_секції **>)]**

У кожен момент часу в критичній секції може знаходитися не більше одного потоку. Якщо критична секція вже виконується яким-небудь потоком, то всі інші потоки, що виконали директиву для секції з даним ім'ям, будуть заблоковані, поки потік, що увійшов, не закінчить виконання даної критичної секції. Як тільки працюючий потік вийде з критичної секції, один із заблокованих на вході потоків увійде до неї. Якщо на вході в критичну секцію стояло декілька потоків, то випадковим чином вибирається один з них, а інші заблоковані потоки продовжують чекати.

Всі неіменовані критичні секції умовно асоціюються з одним і тим же ім'ям. Всі критичні секції, що мають одне і теж ім'я, розглядаються єдиною секцією, навіть якщо знаходяться в різних паралельних областях. Побічні входи і виходи з критичної секції заборонені.

Приклад ілюструє вживання директиви **critical**. Змінна n оголошена поза паралельною областю, тому за умовчанням є загальною. Критична секція дозволяє розмежувати доступ до змінної n. Кожен потік по черзі привласнить n свій номер і потім надрукує набуте значення.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
      int n;
      #pragma omp parallel
```

```
        {
        #pragma omp critical
                {
                n=omp_get_thread_num();
                printf("потік %d\n", n);
                }
        }
}
```

Якби в прикладі не була вказана директива **critical**, результат виконання програми був би непередбачуваний. З директивою **critical** порядок виведення результатів може бути довільним, але це завжди буде набір одних і тих же чисел від 0 до **Omp_num_threads**-1. Звичайно, подібного ж результату можна було б добитися іншими способами, наприклад, оголосивши змінну n локальною, тоді кожна нитка працювала б зі своєю копією цієї змінної. Проте у виконанні цих фрагментів різниця істотна.

Якщо є критична секція, то в кожен момент часу фрагмент оброблятиметься лише одним потоком. Інші потоки, навіть якщо вони вже підійшли до даної точки програми і готові до роботи, чекатимуть своєї черги. Якщо критичної секції немає, то всі потоки можуть одночасно виконати дану ділянку коду. З одного боку, критичні секції надають зручний механізм для роботи із загальними змінними. Але з іншого боку, користуватися їми потрібно обачно, оскільки критичні секції додають послідовні ділянки коду в паралельну програму, що може понизити її ефективність.

1.4.Директива atomic

Частим випадком використання критичних секцій на практиці є оновлення загальних змінних. Наприклад, якщо змінна sum є загальною і оператор вигляду sum=sum+expr знаходиться в паралельній області програми, то при одночасному виконанні даного оператора декількома потоками можна отримати некоректний результат. Щоб уникнути такої ситуації можна скористатися механізмом критичних секцій або спеціально передбаченою для таких випадків директивою **atomic**.

#pragma omp atomic

Дана директива відноситься до оператора привласнення, що йде безпосередньо за нею, гарантуючи коректну роботу із загальною змінною, що стоїть в його лівій частині. На час виконання оператора блокується доступ до даної змінної всім запущеним в даний момент потокам, окрім потоку, що виконує операцію. Атомарною є лише робота із змінною в лівій частині оператора привласнення, при цьому обчислення в правій частині не зобов'язані бути атомарними.

Приклад ілюструє вживання директиви **atomic**. У даному прикладі робиться підрахунок загальної кількості породжених потоків. Для цього кожен потік збільшує на одиницю значення змінної count. Для того, щоб запобігти одночасній зміні декількома потоками значення змінної, що стоїть в лівій частині оператора привласнення, використовується директива **atomic**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
                        int count = 0;
```

```
#pragma omp parallel
        {
            #pragma omp atomic count++;
        }
        printf("Число ниток: %d\n", count);
}
```

1.5.Замки

Один з варіантів синхронізації в OpenMP реалізується через механізм замків (**locks**). Як замки використовуються загальні цілочисельні змінні (розмір має бути достатнім для зберігання адреси). Дані змінні повинні використовуватися лише як параметри примітивів синхронізації..

Замок може знаходитися в одному з трьох станів: неініціалізований, розблокований або заблокований. Розблокований замок може бути захоплений деяким потоком. При цьому він переходить в заблокований стан. Потік, що захопив замок, і лише він може його звільнити, після чого замок повертається в розблокований стан.

Є два типи замків: прості замки і множинні замки. Множинний замок може багато разів захоплюватися одним потоком перед його звільненням, тоді як простий замок може бути захоплений лише один раз. Для множинного замку вводиться поняття коефіцієнта захоплюванності (nesting count). Спочатку він встановлюється в нуль, при кожному наступному захопленні збільшується на одиницю, а при кожному звільненні зменшується на одиницю. Множинний замок вважається розблокованим, якщо його коефіцієнт захоплюванності дорівнює нулю.

Для ініціалізації простого або множинного замку використовуються відповідно функції **omp_init_lock()**і **omp_init_nest_lock()**.

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

Після виконання функції замок переводиться в розблокований стан. Для множинного замку коефіцієнт захоплюванності встановлюється в нуль.

Функції **omp_destroy_lock()** і **omp_destroy_nest_lock()** використовуються для переведення простого або множинного замку в неініціалізований стан.

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

Для захоплення замку використовуються функції **omp_set_lock()** і **omp_set_nest_lock()**.

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

Потік що викликав цю функцію чекає звільнення замку, а потім захоплює його. Замок при цьому переводиться в заблокований стан. Якщо множинний замок вже захоплений даним потоком, то потік не блокується, а коефіцієнт захоплюванності збільшується на одиницю.

Для звільнення замку використовуються функції **omp_unset_lock()** і **omp_unset_nest_lock()**.

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_lock_t *lock);
```

Виклик цієї функції звільняє простий замок, якщо він був захоплений потоком, що викликав. Для множинного замку зменшує на одиницю коефіцієнт з захоплюванності. Якщо коефіцієнт дорівнюватиме нулю, замок звільняється. Якщо після звільнення замку є потоки, заблоковані на операції, що захоплює даний замок, замок буде відразу ж захоплений одним з чекаючих потоків.

Приклад ілюструє вживання технології замків. Змінна lock використовується для блокування. У послідовної області виробляється ініціалізація даної змінної за допомогою функції **omp_init_lock()**. На початку паралельної області кожен потік привласнює змінній n свій порядковий номер. Після цього за допомогою функції **omp_set_lock()** один з потоків виставляє блокування, а інші потоки чекають, поки потік, що викликав цю функцію, не зніме блокування за допомогою функції **omp_unset_lock()**. Всі потоки по черзі виведуть повідомлення "Початок закритої секції..." і "Кінець закритої секції...", при цьому між двома повідомленнями від одного потоку не можуть зустрітися повідомлення від інших потоків. В кінці за допомогою функції **omp_destroy_lock()** відбувається звільнення змінної **lock**.

```
#include <stdio.h>
#include <omp.h>
omp_lock_t lock;
int main(int argc, char *argv[])
{
      int n;
      omp_init_lock(&lock);
      #pragma omp parallel private (n)
      {
            n=omp_get_thread_num();
            omp_set_lock(&lock);
            printf("Початок закритої секції, потік %d\n", n);
            sleep(5);
            printf("Кінець закритої секції, потік %d\n", n);
            omp_unset_lock(&lock);
      }
      omp_destroy_lock(&lock);
}
```

Для неблокуючої спроби захвату замку використовуються функції **omp_test_lock()** і **omp_test_nest_lock()**.

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
```

Дана функція пробує захопити вказаний замок. Якщо це вдалося, то для простого замку функція повертає 1, а для множинного замку – новий коефіцієнт захоплюванності. Якщо замок захопити не вдалося, в обох випадках повертається 0.

Приклад ілюструє вживання технології замків і використання функції **omp_test_lock()**. У даному прикладі змінна lock використовується для блокування. На початку виробляється ініціалізація даної змінної за допомогою функції **omp_init_lock()**. У паралельній області кожен потік привласнює змінній n свій порядковий номер. Після цього за допомогою функції **omp_test_lock()** потоки спробують виставити блокування. Один з потоків успішно виставить блокування, інші ж потоки надрукують повідомлення "Секція закрита...", припинять роботу на дві секунди за допомогою функції **sleep()**, а після

знову намагатимуться встановити блокування. Потік, який встановив блокування, повинен зняти його за допомогою функції **omp_unset_lock()**. Таким чином, код, що знаходиться між функціями установки і зняття блокування, буде виконаний кожним потоком по черзі. В даному випадку, всі потоки по черзі виведуть повідомлення "Початок закритої секції..." і "Кінець закритої секції...", але при цьому між двома повідомленнями від одного потоку можуть зустрітися повідомлення від інших потоків про невдалу спробу увійти до закритої секції. В кінці за допомогою функції **omp_destroy_lock()** відбувається звільнення змінної **lock**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
      omp_lock_t lock; int n;
  omp_init_lock(&lock);
#pragma omp parallel private (n)
      {
            n=omp_get_thread_num();
            while (!omp_test_lock (&lock))
            {
                  printf("Секція закрита, потік %d\n", n);
                  sleep(2);
            }
            printf("Початок закритої секції, потік %d\n", n);
            sleep(5);
            printf("Кінець закритої секції, потік %d\n", n);
            omp_unset_lock(&lock);
      }
      omp_destroy_lock(&lock);
}
```

Хід роботи

Створити програму яка повинна реалізувати наступні дії:

1.Створити матрицю A розміром m * n, елементи якої заповнюються рандомно, m задає кількість рядків і кількість потоків, які виконуватимуть паралельну область програми, n задає кількість стовпців. Змінні можуть задаватися в програмному коді або вводитися з клавіатури.

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

3. Обробити паралельним способом матрицю відповідно до свого варіанту. Кожен потік повинен обробляти свій рядок матриці. Результати обробки вивести у текстовий файл. При цьому перед виводом кожен потік повинен виставити блокування за допомогою механізму замків.

4. У текстовий файл всі потоки по черзі повинні вивести повідомлення "Початок закритої секції..." і "Кінець закритої секції...". Якщо при цьому між двома повідомленнями від одного потоку зустрінуться повідомлення від інших потоків про невдалу спробу увійти до закритої секції, вони також повинні бути записані у файл.

№ Варіанта	Завдання
1	Знайти мінімальне значення кожного рядка матриці
2	Знайти максимальне значення кожного рядка матриці
3	Порахувати кількість додатних елементів в кожному рядку матриці
4	Порахувати кількість від’ємних елементів в кожному рядку матриці
5	Порахувати кількість нульових елементів в кожному рядку матриці
6	Порахувати суму елементів в кожному рядку матриці
7	Порахувати суму додатних елементів в кожному рядку матриці
8	Порахувати суму від’ємних елементів в кожному рядку матриці

Контрольні питання

1. Що станеться, якщо бар'єр зустрінеться не у всіх нитках, виконуючих поточну паралельну область?
2. Чи можуть дві нитки одночасно знаходитися в різних критичних секціях?
3. У чому полягає різниця у використанні критичних секцій і директиви **atomic**?
4. У чому відмінність критичною секцій і замку.