Лабораторна робота №1

Тема: Основні конструкції OpenMP. Модель даних

Мета: Ознайомитися з технологією OpenMP та набути практичних навиків її використання.

Теоретичні відомості

1. Основні конструкції OpenMP

OpenMP – стандарт, який включає директиви компілятора , бібліотеки і системні змінні, які можуть бути використані для вказівки паралелізму в системах із спільною пам'яттю. Це технологія, яку можна розглядати як високорівневу надбудову над Pthreads (або аналогічними бібліотеками потоків). OpenMP передбачається SPMD – Модель (Single Program Multiple Data) паралельного програмування, у рамках якої для всіх паралельних потоків використовується той самий код.

OpenMP простий у використанні і включає лише два базових типи конструкцій: директиви і функції виконуючого середовища OpenMP, які підключаються як додаткова бібліотека omp. h. Директиви в програмах на мові С починаються з **# pragma omp**:

 # pragma omp директива [оператор_1 [, оператор_2, …]]

Об'єктом дії більшості директив є один оператор або блок перед яким розташована директива у вихідному тексті програми. У OpenMP такі оператори або блоки називаються асоційованими з директивою. Асоційований блок повинен мати одну точку входу на початку і одну точку виходу в кінці. Порядок операторів у описі директиви неістотний, в одній директиві більшість операторів можуть зустрічатися кілька разів. Після деяких операторів може слідувати список змінних, що розділяються комами.

Всі директиви OpenMP можна розділити на 3 категорії: визначення паралельної області, розподіл роботи, синхронізація. Кожна директива може мати декілька додаткових атрибутів – опцій (clause). Окремо специфікуються опції для призначення класів змінних, які можуть бути атрибутами різних директив.

У момент запуску програми породжується єдиний потік-майстер або «основний» потік, який починає виконання програми з першого оператора. Основний потік і лише він виконує всі послідовні області програми. При вході в паралельну область породжуються додаткові потоки.

Для виділення паралельних фрагментів програми слід використовувати директиву **parallel**:

#pragma omp parallel [опція **[[,]**опція**]...]<блок програми>**

Для блоку (як і для блоків всіх інших директив OpenMP) повинно виконуватися правило "один вхід – один вихід", тобто передача управління ззовні в блок і з блоку за межі блоку не допускається.

Можливі опції: **private** (список змінних) **–** оголошує змінні зі списку локальними; початкове значення локальних копій змінних із списку не визначене;

- **firstprivate** (список змінних) **–** оголошує змінні зі списку локальними і ініціює їх значеннями з блоку програми, що передує даній директиві;
- **lastprivate** (список змінних) **–** оголошує змінні зі списку локальними і призначає їм значення з того блоку програми, який був виконаний останнім;
- **if** (умова) – виконання паралельної області по умові. Входження в паралельну область здійснюється лише при виконанні деякої умови. Якщо умова не виконана, тодирективна не спрацьовує і продовжується обробка програми в колишньому режимі;

- **num_threads** (скалярний цілий вираз) - **з**адає кількість потоків; за замовчуванням вибирається останнє значення, встановлене за допомогою функції **omp_set_num_threads()** або значення змінної **omp_num_threads**;
- **nowait** – після виконання виділеної ділянки 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 ділянки, продовжити виконання без синхронізації з останніми;
- **default (private|firstprivate|shared|none)** – усім змінним у паралельній області, яким явно не призначений клас, буде призначений клас **private**, **firstprivate** або **shared** відповідно; **none** означає, що всім змінним в паралельній області клас має бути призначений явно;
- **shared** (список змінних) – задає список змінних, загальних для всіх потоків;
- **copyin** (список змінних) – задає список змінних, оголошених як **readprivate**, які при вході в паралельну область ініціалізувалися значеннями відповідних змінних в основному потоці
- **reduction** (оператор:список змінних) – задає оператор і список загальних змінних; для кожної змінної створюються локальні копії в кожному потоці; над локальними копіями змінних після виконання всіх операторів паралельної області виконується заданий оператор; оператори для мови С: – +, *, - &, |, ^, &&, ||;

При вході в паралельну область породжуються нові **omp_num_threads**-1 потоки, кожен потік отримує свій унікальний номер, причому потік, що породжує, отримує номер 0 і стає основним потоком групи. Інші потоки отримують номер з 1 до **omp_num_threads**-1. Кількість потоків, що виконують дану паралельну область, залишається незмінною до моменту виходу з області. При виході з паралельної області виробляється неявна синхронізація і знищуються всі потоки, окрім того, що породив.

Нижче наведений приклад використання першої паралельної програми:

```
#include
<omp.h>
#include
<stdio.h>
int main(int argc, char *argv[])
{
printf("Consistent area 1\n");
// Виділення паралельної області
#pragma omp parallel
{
  printf("Hello World !\n");
}/* Завершення паралельного фрагмента */
  printf("Consistent area 2\n");
}
```

Як приклад використання опції **reduction** розглянемо наступну програму:

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

```
            printf("Current value of count: %d\n", count);
        }
        printf("Number of threads: %d\n", count);
    }
```

Перед запуском програми кількість потоків, що виконують паралельну область, можна задати, визначивши значення змінної середовища **omp_num_threads**. Значення за замовчуванням змінної **omp_num_threads** залежить від реалізації. В програмі її можна змінити за допомогою виклику функції **omp_set_num_threads ():**

void omp_set_num_threads(int num);

Приклад демонструє вживання функції **omp_set_num_threads()** і опції **num_threads**. Перед першою паралельною областю викликом функції **omp_set_num_threads(2)** виставляється кількість потоків, рівна 2. Але до першої паралельної області застосовується опція **num_threads(3),** яка вказує, що дану область слід виконувати трьома потоками. Отже, повідомлення "Parallel area 1" буде виведено трьома потоками. До другої паралельної області опція **num_threads** не застосовується, тому діє значення встановлене функцією **omp_set_num_threads(2),** і повідомлення "Parallel area 2" буде виведено двома потоками.

```
#include
<stdio.h>
#include
<omp.h>
int main(int argc, char *argv[])
{
     omp_set_num_threads(2);
#pragma omp parallel
num_threads(3)
     {
          printf("Parallel area 1\n");
     }
#pragma omp parallel
     {
          printf("Parallel area 2\n");
     }
}
```

В деяких випадках система може динамічно змінювати кількість потоків, використовуваних для виконання паралельної області, наприклад, для оптимізації використання ресурсів системи. Це дозволено робити, якщо змінна середовища **omp_dynamic** встановлена в true. Змінну **omp_dynamic** можна встановити за допомогою функції **omp_set_dynamic():**

void omp_set_dynamic(int num);

Взнати значення змінної **omp_dynamic** можна за допомогою **omp_get_dynamic():**

int omp_get_dynamic(void);

Функція **omp_get_max_threads()** повертає максимально допустиме число потоків для використання в наступній паралельній області:

int omp_get_max_threads(void);

Паралельні області можуть бути вкладеними; за замовчуванням вкладена паралельна область виконується одним потоком. Змінити значення змінної **omp_nested** можна за допомогою виклику функції **omp_set_nested()**:

void omp_set_nested(int nested)

Функція **omp_set_nested()** дозволяє або забороняє вкладений паралелізм. Якщо вкладений паралелізм дозволений, то кожен потік, в якому зустрінеться опис паралельної області, породить для її виконання нову групу потоків. На мові С значення параметра задається 0 або 1. Сам потік, що породив, стане в новій групі основним потоком. Якщо система не підтримує вкладений паралелізм, дана функція не матиме ефекту. Отримати значення змінної **omp_nested** можна за допомогою **omp_get_nested()**:

int omp_get_nested(void);

Функція **omp_in_parallel()** повертає 1, якщо вона була викликана з активної паралельної області програми:

int omp_in_parallel(void);

Якщо в паралельній області яка-небудь ділянка коду має бути виконаний лише один раз, то його потрібно виділити директивами **single**:

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

Директива **master** використовується для визначення структурного блоку програми , який буде виконуватися виключно в головному потоці (паралельному потоці з нульовим номером) з усього набору паралельних потоків. Неявної синхронізації дана директива не передбачає:

#pragma omp master

Синхронізація типу **barrie**r встановлює режим очікування завершення роботи всіх запущених в програмі паралельних потоків при досягненні точки barrier. Для завдання синхронізації типу barrier в OpenMP в програмах , написаних мовою C/C++ , використовується прагма

 # pragma omp barrier

Наступний приклад демонструє вживання директиви **master** і **barrie**r. Змінна n є локальною, тобто кожен потік працює зі своїм екземпляром. Спочатку всі потоки привласнять змінній n значення 0. Потім головний потік (master) привласнить змінній n значення 5, і всі потоки надрукують значення 5. Потім головний потік привласнить змінній n значення 10, і знову всі потоки надрукують значення 10.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
     int n;
    #pragma omp parallel
    {
          n=0;
    #pragma omp master
          {
               n=5;
          }
      printf("First value n: %d\n", n);
    #pragma omp barrier
    #pragma omp master
          {
               n=10;
          }
    printf("Second value n: %d\n", n);
     }
}
```

2. Модель даних

Модель даних в OpenMP передбачає наявність як загальної для всіх потоків області

пам'яті, так і локальної для кожного потоку. В OpenMP змінні в паралельних областях програми розділяються на два основні класи:

shared (загальні; всі потоки бачать одну і ту ж змінну);

private (локальні, приватні; кожен потік бачить свій екземпляр даної змінної).

Загальна змінна завжди існує лише в одному екземплярі для всієї зони дії і доступна всім потокам під одним і тим же ім'ям. Оголошення локальної змінної викликає породження свого екземпляра даною змінною (того ж типуі розміру) для кожного потоку. Зміна потоком значення своєї локальної змінної ніяк не впливає на зміну значення цієї же локальної змінної в інших потоках.

Якщо декілька потоків одночасно записують значення загальної змінної без виконання синхронізації або якщо як мінімум один потік читає значення загальної змінної і як мінімум один потік записує значення цієї змінної без виконання синхронізації, то виникає ситуація так званої «гонки даних» (data race), при якій результат виконання програми непередбачуваний.

За замовчуванням, всі змінні, породжені поза паралельною областю, при вході в цю область залишаються загальними (**shared**). Виняток становлять змінні, що є лічильниками ітерацій в циклі, по очевидних причинах. Змінні, породжені усередині паралельної області, за замовчуванням є локальними (**private**). Явно призначити клас змінних за замовчуванням можна за допомогою опції **default**. Не рекомендується постійно покладатися на правило за замовчуванням, для більшої надійності краще завжди явно описувати класи використовуваних змінних, вказуючи в директивах OpenMP опції **private**, **shared**, **firstprivate**, **lastprivate, reduction**.

Наступний приклад демонструє використання опції **private**. У даному прикладі змінна n оголошена як локальна змінна в паралельній області. Це означає, що кожен потік працюватиме зі своєю копією змінної n, при цьому на початку паралельної області на кожному потоці змінна n не буде ініціалізована. В ході виконання програми значення змінної n буде виведено в чотирьох різних місцях. Перший раз значення n буде виведено в послідовній області, відразу після привласнення змінній n значення 1. Другий раз всі потоки виведуть значення своєї копії змінною n на початку паралельної області. Далі всі потоки виведуть свій порядковий номер, отриманий за допомогою функції **omp_get_thread_num()** і привласнений змінній n. Після завершення паралельної області буде ще раз виведено значення змінної n, яке виявиться рівним 1 (не змінилося під час виконання паралельної області).

```
#include
<stdio.h>
#include
<omp.h>
int main(int argc, char *argv[])
{
      int n=1;
      printf("In the sequential area (begin): %d\n", n);

     #pragma omp parallel private(n)
      {
        printf("The value of n on the thread (at the input): %d\n", n);
        n=omp_get_thread_num();
        printf("n value on thread (output): %d\n", n);
      }
```

```
    printf("In the sequential area (end):  %d\n", n);
}
```

Наступний приклад демонструє використання опції **shared**. Масив m оголошений загальним для всіх потоків. На початку послідовної області масив m заповнюється нулями і виводиться на друк. У паралельної області кожен потік знаходить елемент, номер якого збігається з порядковим номером потоку в загальному масиві, і привласнює цьому елементу значення. В послідовній області друкується змінений масив m.

```
#include
<stdio.h>
#include
<omp.h>
int main(int argc, char *argv[])
{
    int i, m[10];
    printf("Masiv m at the
    beginning:\n");
    for (i=0; i<10; i++)
    {
        m[i]=0;
        printf("%d\n", m[i]);
    }
#pragma omp parallel shared(m)
    {
        m[omp_get_thread_num()]=1;
    }
    printf("Array m at the end:\n");
    for (i=0; i<10; i++) printf("%d\n", m[i]);
}
```

Наступний приклад демонструє використання опції **firstprivate**. Змінна n оголошена як **firstprivate** в паралельній області. Значення n буде виведено в чотирьох різних місцях. Перший раз значення n буде виведено в послідовній області відразу після ініціалізації. Другий раз всі потоки виведуть значення своєї копії змінною n на початку паралельної області, і це значення дорівнюватиме 1. Далі, за допомогою функції **omp_get_thread_num()** всі потоки привласнять змінній n свій порядковий номер і ще раз виведуть значення n. У послідовній області буде ще раз виведено значення n, яке знову виявиться рівним 1.

```
#include
<stdio.h>
#include
<omp.h>
int main(int argc, char *argv[])
{
     int n=1;
     printf("Значення n на початку: %d\n", n);
     #pragma omp parallel firstprivate(n)
     {
     printf("Значення n на потоці (на вході): %d\n", n);
     n=omp_get_thread_num();
     printf("Значення n на потоці (на виході): %d\n", n);
     }
printf("Значення n в кінці: %d\n", n);
}
```

Хід роботи

Створити програму яка повинна реалізувати матрично-векторне множення, використовуючи вхідні дані відповідно до завдання (n – розмірність квадратної матриці). Варіанти 1-10 виконують розбиття матриці по горизонтальних смужках, 11-20 – по вертикальних.

Завдання 1. $a_{ij} = \sin(i) + \cos(j)$; $b_i = \sin(i)\cos(i)$; n=100;

Завдання 2. $a_{ij} = \sin(i + j)$; $b_i = \cos(i)$; n=90;

Завдання 3. $a_{ij} = j\sin(i)$; $b_i = i\sin(i)$; n=120;

Завдання 4. $a_{ij} = i\log(j)$; $b_i = \sqrt{i}$; n=80;

Завдання 5. $a_{ij} = j\tan(i)$; $b_i = i\sqrt{i\cos(i)}$; n=110;

Завдання 6. $a_{ij} = i\sin(j) - tg(i)$; $b_i = \sin(i^2)\sqrt{i}$; n=100;

Завдання 7. $a_{ij} = \frac{i\cos(2j)}{3}$; $b_i = \frac{\lg\sqrt{i+5}}{\tan(2i)+10}$; n=60;

Завдання 8. $a_{ij} = \frac{|\sin(i)|}{5(j+1)}$; $b_i = \frac{1}{\tan(i)+5}$; n=80;

Завдання 9. $a_{ij} = \frac{j^2+3j+2}{\sqrt{i+1}}$; $b_i = \frac{\ln\sqrt{i}\tan(2i)}{\sin(i)+3}$; n=50;

Завдання 10. $a_{ij} = \frac{0.3\sqrt{i}}{\cos(j)+5}$; $b_i = \frac{i}{2}\sqrt{\sin(i)}$; n=100.

Обробити паралельним та послідовним способами множення матриці на вектор відповідно до свого варіанту. Вивести результат виконання алгоритму паралельним та послідовним способами. Визначити час, який був затрачений на виконання програми для обох способів множення матриці на вектор. Розроблену програму виконати почергово на 1, 2, 4 та 8 ядерному процесорі. Побудувати графік залежності часу обчислення від кількості ядер. Навести оцінки паралельного прискорення (parallel speedup) та паралельної ефективності (parallel efficiency).

Примітка. Для профілювання OpenMP програми можна використовувати таймери. Функція, що повертає час у секундах, що минув з довільного моменту в минулому має наступний формат:

double omp_get_wtime (void);

Точка відліку залишається незмінною протягом усього часу виконання програми.

Приклад найпростішої С-програми, яка демонструє використання функції **omp_get_wtime ()**, наведено нижче:

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
double omp_get_wtime(void);
int main()
```

```
{  double t1, t2;
   t1 = omp_get_wtime();
   sleep(2);
   /* Sleep for 2 seconds */
    t2 = omp_get_wtime();
   printf("%e\n", t2-t1);
   return 0;
}
```

Функція, що повертає час у секундах, що минув між послідовними “тиками”:

double omp_get_wtick (void);

Цей час є мірою точності таймера.

Контрольні питання

1. Яке основне призначення інтерфейсу OpenMP?
2. Які базові типи конструкцій технології OpenMP Ви знаєте?
3. Директива parallel та її опції.
4. Як відбувається породження та завершення паралельних процесів(ниток) в OpenMP?
5. У яких випадках може бути необхідне використання опції **if** директиви **parallel**?
6. Чим відрізняються директиви **single** і **master**?
7. Чи може одна і та ж змінна виступати в одній частині програми як загальна, а в іншій частині – як локальна?
8. Що станеться, якщо декілька потоків одночасно звернуться до загальної змінної?
9. Яким чином при вході в паралельну область розіслати всім потокам значення деякої змінної?
10. Чи можна зберегти значення локальних копій загальних змінних після завершення паралельної області?