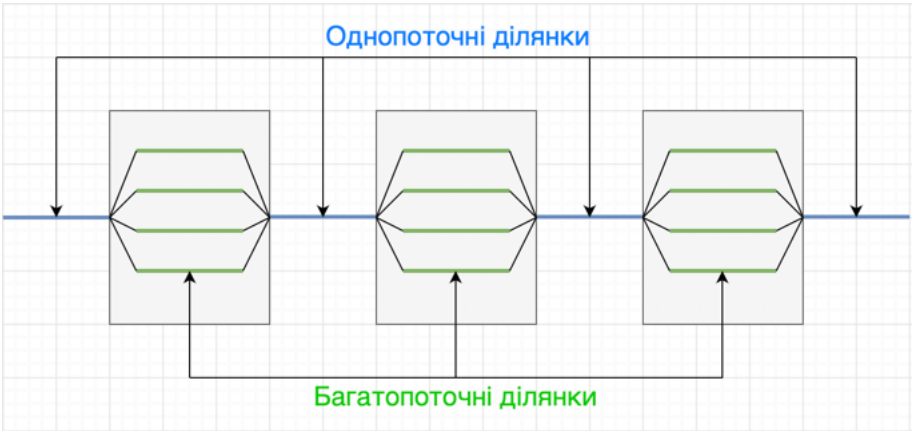
ЛЕКЦІЯ 6

БІБЛІОТЕКА OpenMP

Однією з найбільш популярних технологій та середовищ програмування для комп'ютерів зі спільною пам'яттю, що базуються на традиційних мовах програмування є технологія OpenMP. За основу береться послідовна програма, а для створення її паралельного представлення користувачеві (програмісту) представляється набір директивів, функцій та змінних оточення.

Під паралельною програмою в рамках OpenMP розуміється програма, для якої в спеціально вказуваних за допомогою директив місцях - паралельних фрагментах - програмний код, що виконується, може бути розділений на кілька окремих командних потоків (threads). У загальному вигляді програма подається у вигляді набору послідовних (однопоточних) та паралельних (багатопоточних) ділянок програмного коду.



Для того, щоб використовувати механізм OpenMP потрібно скомпілювати програму компілятором, що підтримує цю технологію з

вказанням відповідного ключа (наприклад, в icc/ifort використовується ключ компилятора `-openmp`, в gcc /gfortran `-fopenmp`, Sun Studio `-xopenmp`, в Visual C++ - `/openmp`, в PGI `-mp`)

Для перевірки того, що компілятор підтримує якусь версию OpenMP, достатньо написати директиви умов компіляції `#ifdef` або `#ifndef`. Найпростіші приклади умови компиляции в программах на мовах Си та Фортран.

Умови компіляції на мові С.

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
printf("OpenMP is supported!\n");
#endif
}
```

Умови компіляції на мові Fortran.

```
      program example1b
#ifdef _OPENMP
      print *, "OpenMP is supported!"
#endif
end
```

Модель паралельної програми. Розпаралелювання в OpenMP виконується явно за допомогою вставки у текст спеціальних директивів, а також викликів допоміжних функцій. При використанні OpenMP припускається SPMD-модель (Single Program Multiple Data) паралельного програмування, в рамках якої для всіх паралельних потоків використовується один і той самий код.

Розберемо деякі директиви та функції технології OpenMP.

Значна частина функціональності OpenMP реалізується за допомогою директивів компілятору. Вони повинні бути явно вставлені користувачем, що дозволить виконати програму у паралельному режимі.

Директиви в мові С оформлюються інструкціями процесору, зо починаються з **#pragma omp**. Ключове слово **omp** використовується для того, щоб уникнути співпадінь імен директивів OpenMP з іншими іменами.
Формати директивів в C/C++ виглядають наступним чином:

```
#pragma omp directive-name [опція[[,] опція]...]
```

Всі директиви OpenMP поділяються на 3 категорії: визначення паралельної області, розподіл роботи, синхронізація. Кожна директива може мати декілька виконуючих атрибутів. Окремо вказуються опції для призначення класів змінних, що можуть бути атрибутами різноманітних директивів.

Щоб використовувати функції бібліотеки OpenMP треба включити у програму заголовочний файл **omp.h**, якщо ви використовуєте тільки директиви OpenMP, цього робити не треба. Всі функції, що використовуються в OpenMP починаються з префіксу **_omp** .

Розберемо виконання програми та деякі функції для роботи з системним таймером, наприклад, а також найпростіші приклади програм.

Після отримання виконуємого файлу, необхідно запустити його на потрібній кількості процесорів. Для цього, як правило, треба задати необхідну кількість ниток, що виконують паралельні області програми, визначивши значення змінної середовища **OMP_NUM_THREADS** .

Наприклад в Linux в Bash це можна зробити за допомогою команди:

```
export OMP_NUM_THREADS=n
```

Після запуску починає працювати одна нитка, а всередині паралельних областей одна й та сама програма буде виконуватися всім набором ниток.

Вимірювання часу.

В OpenMP передбачені функції для роботи із системним таймером.

Функція **omp_get_wtime()** повертає для нитки, яка її викликала, астрономічний час у секундах (дійсне число подвійної точності), що пройшло з деякого моменту у минулому.

```
double omp_get_wtime(void);
```

Якщо деякий участок коду «обернути» викликами цієї функції, то різниця повертаємих значень покаже час, за який виконався цей участок коду. Гарантується, що момент часу, використовуємий в якості точки відліку не буде змінений за час існування процесу. Таймери різних ниток можуть бути не синхронізовані та видавати різні значення. Функція **omp_get_wtick()** повертає у використаній нитці розмірність таймеру, це порівняно з мірою точності таймеру.

```
double omp_get_wtick(void);
```

В наступному прикладі на мові С ілюструється робота з даними функціями для роботи з таймером в OpenMP.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
double start_time, end_time, tick;
start_time = omp_get_wtime();
end_time = omp_get_wtime();
tick = omp_get_wtick();
```

```
printf("Час на замір часу %lf\n", end_time-start_time);
printf("Точність таймера %lf\n", tick);
}
```

Директива ***parallel***

Паралельна область задається за допомогою директиви ***parallel***

```
#pragma omp parallel [опція[[,] опція]...]
```

Можливі опції:

- **if** (умова) – виконання паралельної області за деякої умови.
- **num_threads** (вираз цілих чисел) – явне задання кількості ниток, що будуть виконувати паралельну область. За замовчанням встановлюється останнє значення, встановлене функцією **omp_set_num_threads()**, або значення змінної **OMP_NUM_THREADS**
- **default(private|firstprivate|shared|none)** – всім змінним паралельної області, яким явно не призначений клас, будуть призначені класи **private**, **firstprivate** або **shared** відповіжно. **None** означає, що класи повинні бути задані явно.
- **Private (**список**)** – задає список змінних, для яких робиться локальна копія для кожного потоку.
- **Firstprivate (**список**)** – задає список змінних, для яких робиться локальна копія для кожної нитки. Локальні копії змінних ініціалізуються значеннями цих змінних в потоці-мастері.
- **Shared(**список**)** – Задає список змінних, що спільний для всіх ниток.

- **Copyin(**список**)** – задає список змінних, заявлених як **threadprivate**, що при вході в паралельну область ініціалізуються значеннями відповідних змінних в потоці-мастері.
- **Reduction**(оператор:список) – задає оператори та список спільних змінних, для кожної змінної створюється локальна копія в кожному потоці. Локальні копії ініціалізуються згідно типу оператору (адитивний-0, мультиплікативний-1).

При вході у паралельне середовище породжуються нові **OMP_NUM_THREADS-1** потоки, кожен потік отримає свій унікальний номер, при чому породжуємий потік має номер 0 та стає основною для групи. Інші потоки отримують в якості номеру цілі числа з 1 до **OMP_NUM_THREADS-1.** Кількість потоків, що задіяні у виконанні паралельної задачі, залишається незмінною до виходу з паралельної області. При виході з паралельної області виконується неявна синхронізація та знищуються всі потоки, окрім головного.

Всі побічні потоки мають один і той код, що відповідає паралельній код. Припускається, що в SMP-системі потоки будуть розподілені по різним процесам.

Наступний приклад демонструє використання директиви parallel.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
printf("hello world\n"); #pragma omp parallel
{
printf("parallel\n");
}
printf("bye world\n");
```

```
}
```

Наступний приклад демонструє виконання функції **reduction** . В цьому прикладі ми рахуємо кількість побічних потоків. Кожен потік ініціалізує спільну змінну **count** значенням 0. Далі, кожен потік збільшує значення своєї копії **count** на 1 та виводить отримане значення. На виході всі значення додаються та присвоюються глобальній змінній.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int count = 0;
#pragma omp parallel reduction (+: count)
{
count++;
printf("Поточне значення count: %d\n", count); }
printf("Число потоків: %d\n", count); }
```

Паралельні середовища та допоміжні функції.

Для того щоб змінювати значення змінної середовища **OMP_NUM_THREADS** можна викликати функцію **omp_set_num_threads()**

```
void omp_set_num_threads(int num)
```

Приклад використання цієї функції:

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

```
   omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
{
printf("Параллельна область 1\n");
   }
#pragma omp parallel
{
printf("Параллельна область 2\n");
} }
```

В деяких випадках система може динамічно змінювати кількість паралельних областей, що виконують паралельну область для оптимізації ресурсів. Це дозволено робити, якщо змінна середовища **OMP_DYNAMIC** встановлена на значення true.

Ось як вона встановлюється в bash Linux:

```
export OMP_DYNAMIC=true
```

В системах з динамічною зміною кількості потоків кількість потоків за замовчуванням не визначена. Змінну **OMP_DYNAMIC** можна встановити за допомогою функії **omp_set_dynamic()**. В аргументи передаються значення 0 або 1. Дізнатися значення **OMP_DYNAMIC** можна за допомогою функції **omp_get_dynamic().**

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
printf("Значення OMP_DYNAMIC: %d\n", omp_get_dynamic());
omp_set_dynamic(1);
printf("Значення OMP_DYNAMIC: %d\n", omp_get_dynamic());
#pragma omp parallel num_threads(128)
```

```
   {
#pragma omp master
      {
} }
}
printf("Паралельна область, %d потоків\n",
omp_get_num_threads());
```

Функція **omp_get_max_threads()** повертає максимальну допустиму кількість потоків для використання у наступній паралельній області.

Функція **omp_get_num_procs()** повертає кількість процесів, доступних для використання програмі користувача на момент виклику.

Паралельні області можуть бути вкладеними, виконанням вкладених паралельних областей можна за допомогою функції **omp_set_nested()**, що змінює значення змінної середовища **OMP_NESTED.**

Ця функція дозволяє або забороняє вкладений паралельзм та приймає аргументи 0 або 1.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
   omp_set_nested(1);
#pragma omp parallel private(n)
   {
      n=omp_get_thread_num();
#pragma omp parallel
```

```
      {
printf("Часть 1, нить %d - %d\n", n, omp_get_thread_num());
} }
   omp_set_nested(0);
#pragma omp parallel private(n)
   {
      n=omp_get_thread_num();
#pragma omp parallel
      {
} }
}
```

Функція **omp_in_parallel()** повертає 1, якщо вона визвана з активної паралельної області програми.

```
#include <stdio.h>
#include <omp.h>
void mode(void){
if(omp_in_parallel()) printf("Паралельная область\n");
else printf("Последовна область\n"); }
int main(int argc, char *argv[])
{
   mode();
#pragma omp parallel
   {
#pragma omp master
{
mode();
} }
}
```

Директива ***single***

Якщо в паралельній області який-небуть участок коду повинен бути виконаний лише один раз, то його треби виділити цією директивою.

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

Можливі опції:

- **Private (**список**) -** задає список змінних, для яких робиться локальна копія для кожного потоку.
- **Firstprivate (**список**) -** задає список змінних, для яких робиться локальна копія для кожної нитки. Локальні копії змінних ініціалізуються значеннями цих змінних в потоці-мастері.
- **Copyprivate (**список**) -** Після виконання нитки, що містить конструкцію single нові значення змінних будуть доступні для всіх однойменних приватних змінних, що описані на початку паралельної області.
- **Nowait –** після виконання виділеного участка відбувається неявна бар'єрна синхронізація паралельно працюючих потоків.

Приклад використання single та nowait

```
#include <stdio.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
{
printf("Сообщение 1\n");
#pragma omp single nowait
      {
printf("Один потік\n"); }
printf("Повідомлення 2\n"); }
}
```

Приклад використання single та copyprivate

```
#include <stdio.h>
```

```
#include <omp.h>
int main(int argc, char *argv[])
{
   int n;
#pragma omp parallel private(n)
{
n=omp_get_thread_num(); printf("Значення n (початок):
%d\n", n);
#pragma omp single copyprivate(n)
      {
n=100; }
printf("Значення n (кінец): %d\n", n); }
}
```

Директива ***master***

Директива ***master*** виділяє частину коду, що буде виконана тільки ниткою-мастером. Інші нитки пропускають цей участок та продовжують роботу оператору, розташованого пісня цієї директиви.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int n;
#pragma omp parallel private(n)
{
n=1;
#pragma omp master
      {
n=2; }
printf("Перше значення n: %d\n", n); #pragma omp barrier
#pragma omp master
      {
n=3; }
printf("Друге значення n: %d\n", n); }
}
```

Модель даних.

Модель даних в OpenMP така, що для всіх потоків програми існує спільна область пам’яті, а також існує локальна область пам’яті для кожного потоку.

Змінні в паралельних областях програми діляться на два класи. **`shared`** (Спільні; всі потоки бачать одну й ту саму змінну); **`private`** (Локальні, приватні; кожен потік бачить свій екземпляр цієї змінної).

Спільна змінна завжди існує лише в одному екземплярі для всієї програмної області та доступна ниткам під одним і тим самим ім’ям. Оголошення локальної змінної викликає породження нового екземпляра цієї змінної (того ж типу та розміру) для кожної нитки. Зміна ниткою значення своєї локальної змінної ніяк не впливає на зміну значень цієї змінною в інших локальних областях.

Якщо декілька змінних одночасно записують значення спільної змінної без виконання синхронізації, то виникає ситуація так званої гонки даних (розбирали на перших лекціях).

За замовченням, всі змінні, породжені за паралельною областю, при вході у цю область залишаються спільними. Виключення це змінні, що є лічильниками ітерацій у циклі. Змінні, що породжені у паралельній області за замовчуванням є приватними.

Приклад використання опції private:

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

```
{
int n=1;
printf("n в послідовній області (початок): %d\n", n);
#pragma omp parallel private(n)
{
printf("Значення n на потоці (на вході): %d\n", n);
/* Привласнимо змінній n номер поточного потоку */
n=omp_get_thread_num();
printf("Значення n на потоці (на виході): %d\n", n);
}
printf("n в послідовній області (кінець): %d\n", n); }
```

Та приклад використання опції shared:

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int i, m[10];
printf("Масив m на початку:\n");
/* Заповнюємо масив m нулями та надрукуємо його */
for (i=0; i<10; i++)
{
m[i]=0;
      printf("%d\n", m[i]);
   }
#pragma omp parallel shared(m)
   {
/* Привласнимо 1 елементу масива m, номер якого співпадає з
номером поточного потоку */
m[omp_get_thread_num()]=1;
}
/* Ще раз надрукуємо масив */
printf("Масив m в кінці:\n");
for (i=0; i<10; i++) printf("%d\n", m[i]);
}
```

У мові C++ статичні (static) змінні, що визначені в паралельній області програми є спільними (shared). Динамічно виділена пам’ять також є спільною, однак вказівник на неї може бути як спільним, так і локальним.

Зараз ми побачимо приклад використання опції firstprivate. Змінна n є змінною firstprivate у паралельній області.

```
#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);
/* Привласнимо змінній n номер поточного потоку */
n=omp_get_thread_num();
printf("Значення n на потоці (на виході): %d\n", n);
}
printf("Значення n в кінці: %d\n", n); }
```

Директива threadprivate вказує на те, що змінні зі списку повинні бути розмножені так, що кожна нитка має свою локальну копію.

Ця директива дозволяє зробити локальні копії статичних змінних, що за замовчуванням є спільними. Для коректного використання локальних копій глобальних об’єктів потрібно гарантувати, що вони використовуються в різних частинах програми одними й тими самими нитками. Якщо на локальні копії ссилаються у різних паралельних областях, то для збереження їх значення необхідно, щоб кількість ниток у всіх областях співпадало, а змінна

OMP_DYNAMIC була встановлена на значення false щ початку першої області до початку другої (якщо на локальну копію є ссилка у двох областях). Змінні threadprivate можуть використовуватися тількі в опціях директив copyin, coptprivate, schedule, num_threads, if. Приклад використання:

```
#include <stdio.h>
#include <omp.h>
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[])
{
int num;
   n=1;
#pragma omp parallel private (num)
{
num=omp_get_thread_num();
printf("Значення n на потоці %d (на вході): %d\n", num, n);
/* Привласнимо змінній n номер поточного потоку */
n=omp_get_thread_num();
printf("Значення n на потоці %d (на виході): %d\n", num,
n);
}
printf("Значення n (середина): %d\n", n);
#pragma omp parallel private (num)
{
num=omp_get_thread_num();
printf("Значення n на потоці %d (ще раз): %d\n", num, n);
} }
```

Якщо необхідно змінну threadprivete ініціалізувати значенням змінною, яку можна розмножити з головної нитки, то при вході у паралельну область можна використовувати опцію copyin.

```
#include <stdio.h>
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[])
{
   n=1;
#pragma omp parallel copyin(n)
{
printf("Значення n: %d\n", n);
} }
```