Práctica de Laboratorio
OpenMP: Ejemplos y ejercicios

1. Marco teórico

1.1. ¿Qué es OpenMP?

OpenMP (Open Multi-Processing) es una API (Interfaz de Programación de Aplicaciones) que permite la programación paralela en memoria compartida, utilizando múltiples hilos de ejecución. Se integra fácilmente con los lenguajes C, C++ y Fortran, y está especialmente diseñado para ejecutarse en sistemas con múltiples núcleos o procesadores.

OpenMP utiliza directivas del compilador, funciones de biblioteca y variables de entorno para controlar la paralelización. Su uso más común es la paralelización de bucles que se ejecutan de forma independiente entre iteraciones.

1.2. Ventajas del OpenMP

- Facilidad de uso: no requiere reescribir completamente el código.
- Portabilidad: disponible en la mayoría de compiladores modernos.
- Escalabilidad: se adapta a sistemas con distintos números de núcleos.
- Ideal para sistemas con memoria compartida (como PC multicore).

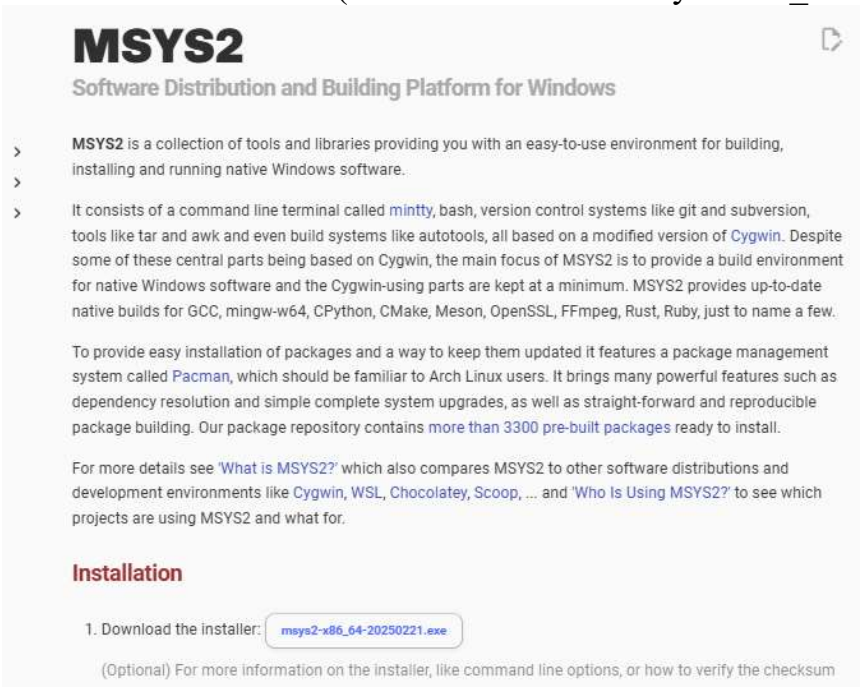
1.3. Condición de sincronización

- **#pragma omp parallel:** crea un bloque de código que se ejecuta en paralelo.
- **#pragma omp for:** distribuye las iteraciones de un bucle entre hilos.
- **#pragma omp critical:** protege una sección de código para que sea ejecutada por un solo hilo a la vez.
- **#pragma omp sections:** permite dividir tareas independientes entre hilos.
- **#pragma omp single:** indica que una sección del código será ejecutada solo por un hilo.

2. Instalar OpenMP

PASO 1: Descarga MSYS2

Ve a: https://www.msys2.org/
Haz clic en "Download installer" (el archivo se llama msys2-x86_64-....exe).



Paso 2: Instálalo

- Instala MSYS2 en una ruta sin espacios (por ejemplo, C:\msys64).
- Cuando termine, ejecuta MSYS2 desde el menú inicio (MSYS2 MinGW 64-bit).

Paso 3: Instala g++ con OpenMP

Dentro de la terminal de MSYS2, ejecuta estos comandos:

```
pacman -Syu   #Actualiza MSYS2 (puede cerrarse solo)
pacman -Su    #Ejecuta nuevamente tras reiniciar terminal

# Instala el compilador g++ con soporte para OpenMP:
pacman -S mingw-w64-x86_64-gcc
```

Paso 4: Usa el compilador

Abre "MSYS2 MinGW 64-bit" desde el menú inicio y compila:

```
cd /c/mis_proyectos_cpp
g++ -fopenmp hola.cpp -o hola.exe
```

Debes ubicar primero la carpeta en donde estará el archivo y luego compilarlo:

```
Usuario@DESKTOP-L2GAVL9 MINGW64 ~
$ cd /c/practica_5

Usuario@DESKTOP-L2GAVL9 MINGW64 /c/practica_5
```

```
Usuario@DESKTOP-L2GAVL9 MINGW64 /c/practica_5
$ g++ -fopenmp hola.cpp -o hola.exe
```

Y luego sería ejecutarlo:

```
./hola.exe
```

```
Usuario@DESKTOP-L2GAVL9 MINGW64 /c/practica_5
$ ./hola.exe
Hola desde el hilo Hola desde el hilo 2 de 4 hilos.Hola desde el hilo 3 de 4 hilos.

1 de 4 hilos.
Hola desde el hilo 0 de 4 hilos.
```

Y si quieres cambiar el número de threads:

```
export OMP_NUM_THREADS=4
./hola.exe
```

```
Usuario@DESKTOP-L2GAVL9 MINGW64 /c/practica_5
$ export OMP_NUM_THREADS=4

Usuario@DESKTOP-L2GAVL9 MINGW64 /c/practica_5
$ ./hola.exe
Hola desde el hilo 1 de 4 hilos.
Hola desde el hilo 3 de 4 hilos.
Hola desde el hilo 0 de 4 hilos.
Hola desde el hilo 2 de 4 hilos.
```

3. Ejemplo en c++: Suma de elementos en un arreglo

Generación de varios threads y modificación del número de threads Mediante: export OMP_NUM_THREADS=4.

```
#include <iostream>
#include <omp.h>

using namespace std;

#define N 24

int main() {
    int tid, nthr;      // Identificador del thread y número de hilos
    int A[N];

    // Inicializar el arreglo
    for (int i = 0; i < N; i++) {
        A[i] = 0;
    }

    // Sección paralela
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        nthr = omp_get_num_threads();

        cout << "Thread " << tid << " de " << nthr << " en marcha" << endl;

        A[tid] = tid + 10;

        cout << "El thread " << tid << " ha terminado" << endl;
    }
```

```
    // Mostrar resultados
    for (int i = 0; i < N; i++) {
        cout << "A[" << i << "] = " << A[i] << endl;
    }

    return 0;
}
```

Ejecución:

```
Usuario@DESKTOP-L2GAVL9 MINGW64 /c/practica_5
$ export OMP_NUM_THREADS=4

Usuario@DESKTOP-L2GAVL9 MINGW64 /c/practica_5
$ ./hola2.exe
Thread 2 de 4 en marcha
El thread 2 ha terminado
Thread 3 de 4 en marcha
El thread 3 ha terminado
Thread 1 de 4 en marcha
El thread 1 ha terminado
Thread 0 de 4 en marcha
El thread 0 ha terminado
A[0] = 10
A[1] = 11
A[2] = 12
A[3] = 13
A[4] = 0
A[5] = 0
A[6] = 0
A[7] = 0
A[8] = 0
A[9] = 0
A[10] = 0
A[11] = 0
A[12] = 0
A[13] = 0
A[14] = 0
A[15] = 0
A[16] = 0
A[17] = 0
A[18] = 0
A[19] = 0
A[20] = 0
A[21] = 0
A[22] = 0
A[23] = 0
```

Explicación:

Este programa utiliza programación paralela con OpenMP para que varios hilos (threads) trabajen simultáneamente, modificando diferentes posiciones de un arreglo. El objetivo es mostrar cómo cada hilo puede hacer un trabajo independiente y cómo identificar qué hilo hace qué.

Práctica

1. Definicion del numero de threads mediante la función omp_ser_num_threads().

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

#define N 24

int main() {
    int tid, nthr;
    int A[N];

    // Inicializamos el arreglo
    for (int i = 0; i < N; i++) {
        A[i] = 0;
    }

    // Preguntamos al usuario cuántos hilos desea
    cout << "\nIntroduce el número de threads ---> ";
    cin >> nthr;

    // Establecer el número de hilos si OpenMP está disponible
    #ifdef _OPENMP
    omp_set_num_threads(nthr);
    #endif

    // Sección paralela
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        cout << "Thread " << tid << " en marcha" << endl;

        A[tid] = tid + 10;

        cout << "El thread " << tid << " ha terminado" << endl;
    }

    // Mostrar resultados
    for (int i = 0; i < N; i++) {
        cout << "A[" << i << "] = " << A[i] << endl;
    }

    return 0;
}
```

Explicación:

Este programa usa OpenMP para ejecutar varias tareas en paralelo con hilos. Cada hilo escribe un valor distinto en un arreglo compartido, mostrando cómo se puede dividir el trabajo entre ellos. Sirve para demostrar el funcionamiento básico del paralelismo en C++ y cómo los hilos pueden trabajar al mismo tiempo sin interferirse.

2. Los tres casos juntos(pej: export con 16, teclado con 8, función con 6 threads)

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

#define N 24

int main() {
    int tid, nthr;
    int A[N];

    // Inicializamos el arreglo
    for (int i = 0; i < N; i++) A[i] = 0;

    // === 1. Número de hilos controlado por variable de entorno ===
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        cout << "Thread " << tid << " en marcha (región 1)" << endl;
        A[tid] = tid + 10;
        cout << "El thread " << tid << " ha terminado (región 1)" << endl;
    }

    for (int i = 0; i < N; i++)
        cout << "A[" << i << "] = " << A[i] << endl;

    cout << "\n>> Fin de la primera región paralela\n\n";

    // === 2. Número de hilos controlado por función ===
    cout << "Introduce el número de threads ---> ";
    cin >> nthr;

    #ifdef _OPENMP
    omp_set_num_threads(nthr);
    #endif

    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        cout << "Thread " << tid << " en marcha (región 2)" << endl;
        A[tid] = tid + 100;
        cout << "El thread " << tid << " ha terminado (región 2)" << endl;
    }

    for (int i = 0; i < N; i++)
        cout << "A[" << i << "] = " << A[i] << endl;

    cout << "\n>> Fin de la segunda región paralela\n\n";

    // === 3. Número de hilos controlado por cláusula num_threads ===
    #pragma omp parallel private(tid) num_threads(6)
    {
        tid = omp_get_thread_num();
        cout << "Thread " << tid << " en marcha (región 3)" << endl;
        A[tid] = tid + 1000;
        cout << "El thread " << tid << " ha terminado (región 3)" << endl;
    }
```

```
    for (int i = 0; i < N; i++)
        cout << "A[" << i << "] = " << A[i] << endl;

    cout << "\n>> Fin de la tercera región paralela\n";

    return 0;
}
```

Explicación:

En este caso, se define de forma explícita cuántos hilos (threads) se deben utilizar, directamente dentro de la directiva #pragma omp parallel, usando la cláusula num_threads(6).

Esto significa que, sin importar la configuración del sistema o lo que haya escrito el usuario, el programa ejecutará esta región paralela utilizando exactamente 6 hilos.

Este enfoque es útil cuando se quiere controlar con precisión cuántos hilos se usarán para una tarea, como por ejemplo dividir el trabajo entre 6 procesos paralelos, simular 6 usuarios, o hacer pruebas de rendimiento con una cantidad fija de hilos.