

**Universidad Nacional de San Antonio Abad del Cusco**  
**Departamento Académico de Informática**  
**ALGORITMOS PARALELOS Y DISTRIBUIDOS**  
**Práctica N° 2**

**Introducción a OpenMP**

Ivan Medrano Valencia

**1. OBJETIVO.**

- Conocer la programación paralela en arquitecturas de memoria compartida.
- Utilizar la API OpenMP para hacer programas paralelos.

**2. BASE TEORICA COMPLEMENTARIA.**

**2.1. Memoria compartida en una computadora.**

En el hardware de una computadora, la memoria compartida se refiere a un bloque de memoria de acceso aleatorio a la que se puede acceder por varias unidades de procesamiento diferentes. Tal es el caso de las computadoras *multicore* donde se tienen varios núcleos que comparten la misma memoria principal (ver Fig 1).

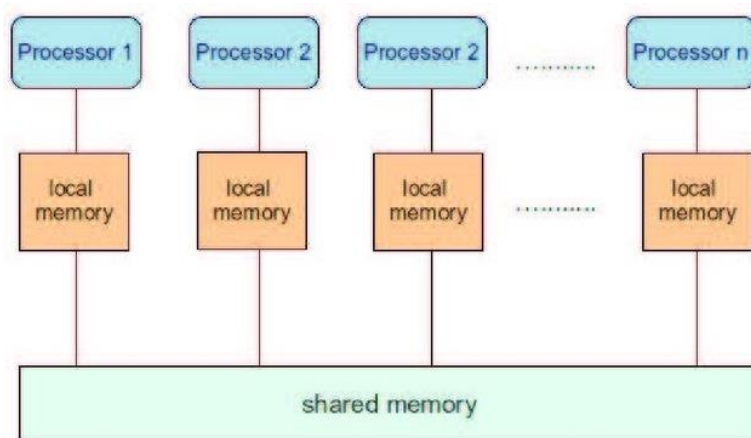


Fig. 1. Memoria local y memoria compartida

La memoria compartida es un medio de comunicación entre procesos / hilos, es decir; una manera de intercambiar datos entre programas o instrucciones que se ejecutan al mismo tiempo. Un proceso/hilo creará un espacio en la memoria RAM a la que otros procesos o hilos pueden tener acceso.

## 2.2. OpenMP (Open Multi-Processing).

Es una interfaz de programación de aplicaciones (API) multiproceso portable, para computadoras paralelas que tienen una memoria compartida.

OpenMP está formado por:

- Un conjunto de directivas del compilador, las cuales son ordenes abreviadas que instruyen al compilador para insertar órdenes en el código fuente y realizar una acción en particular.
- Una biblioteca de funciones, y
- Variables de entorno.

Que se utilizan para paralelizar programas escritos en lenguaje C, C++ y Fortran.

Para utilizar OpenMP es necesario contar con un compilador que incluya estas extensiones al lenguaje.

## 2.3. Arquitectura de OpenMP.

Para paralelizar un programa se tiene que hacer de forma explícita, es decir, el programador debe analizar e identificar qué partes del problema o programa se pueden realizar de forma concurrente y por tanto se pueda utilizar un conjunto de hilos que ayuden a resolver el problema.

OpenMP trabaja con la llamada arquitectura fork-join, donde a partir del proceso o hilo maestro se genera un número de hilos que se utilizarán para la solución en paralelo llamada región paralela y después se unirán para volver a tener un solo hilo o proceso maestro. El programador especifica en qué partes del programa se requiere ese número de hilos. Debido a esto, se dice que OpenMP combina código serial y paralelo (ver Fig. 2).

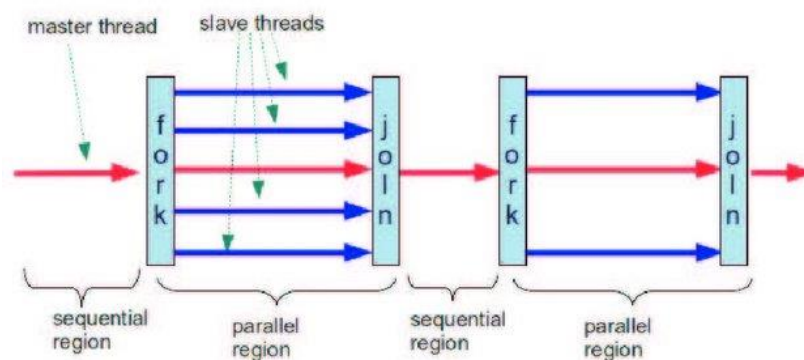


Fig. 2. Modelo fork/join de OpenMP

## 2.4. Directivas o pragmas.

Agregar una directiva o pragma en el código es colocar una línea como la que sigue:

***#pragma omp nombreDelConstructor <clausula o clausulas>***

Donde se puede observar, se tiene los llamados constructores y las cláusulas. Los constructores es el nombre de la directiva que se agrega y las cláusulas son

atributos dados a algunos constructores para un fin específico; una cláusula nunca se coloca si no hay antes un constructor.

Algunos constructores, cláusulas y funciones de biblioteca se irán explicando en el desarrollo de la práctica.

### 3. DESARROLLO DE LA PRÁCTICA

Para el desarrollo de la práctica se utilizará el compilador Visual C++ del Visual Studio versión 2013 o superior.

#### 3.1. ACTIVIDAD 1.

##### 3.1.1. El constructor *parallel*.

El primer constructor a revisar es el más importante, *parallel*, el cual permite crear regiones paralelas, es decir generar un número de hilos que ejecutarán ciertas instrucciones y cuando terminen su actividad se tendrá solo al maestro. (Ver Fig.3)

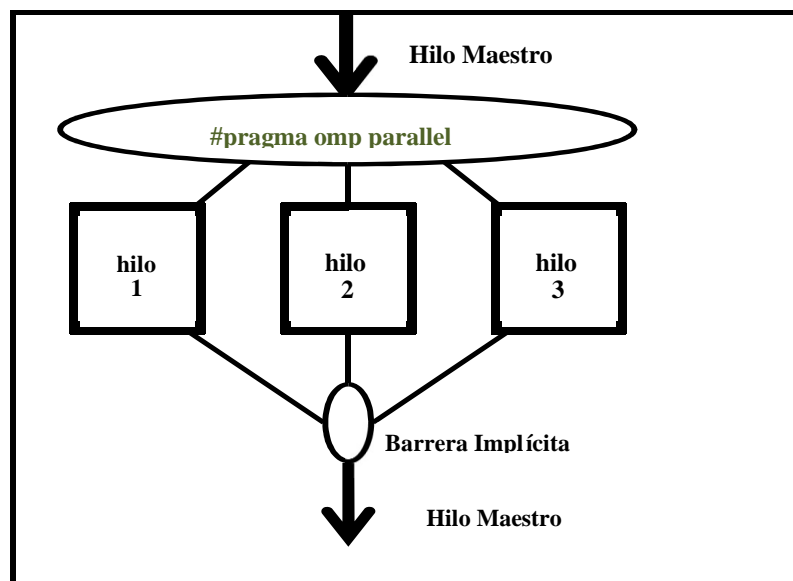


Fig. 3. Funcionamiento de la cláusula parallel.

Su sintaxis es como sigue:

```
#pragma omp parallel
{
    //Bloque de código paralelo
}
```

### 3.1.2. Ejercicio 1.-

Escribir el siguiente programa secuencial que muestra el mensaje “Hola Mundo...”.

```
#include <iostream>
int main()
{
    std::cout << "Hola Mundo...\n" << std::endl;
    std::cout << "Adios...\n" << std::endl;
    getchar();
    return 0;
}
```

La ejecución de este programa producirá la siguiente salida.

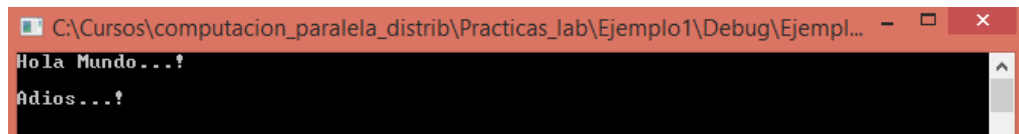


Fig. 4. Ejecución del programa secuencial

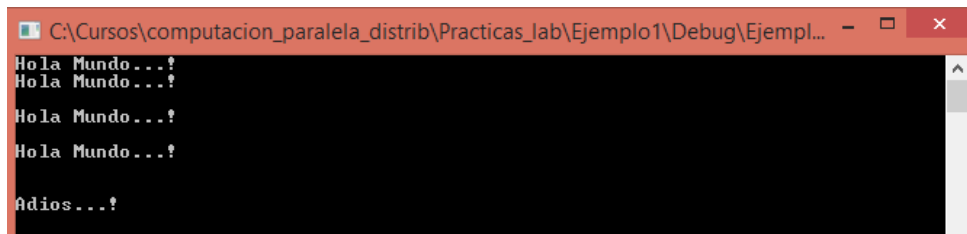
### 3.1.3. Ejercicio 2.-

Para paralelizar el programa del ejercicio 1, debemos agregar el constructor ***parallel***, como se muestra en el siguiente programa:

```
#include <iostream>
#include <omp.h>
int main()
{
    #pragma omp parallel
    {
        std::cout << "Hola Mundo...\n" << std::endl;
    }

    std::cout << "Adios...\n" << std::endl;
    getchar();
    return 0;
}
```

OpenMP ha detectado automáticamente 4 procesadores, por lo que ha creado 4 hilos y podemos ver el resultado de la Fig. 5.



```
C:\Cursos\computacion_paralela_distrib\Practicas_lab\Ejemplo1\Debug\Ejempl...
Hola Mundo...!
Hola Mundo...!
Hola Mundo...!
Hola Mundo...!
Adios...!
```

Fig. 5. Ejecución paralela

Pero para poder obtener este resultado cada vez que ejecutemos un programa paralelo con OpenMP, debemos configurar Visual C++ de la siguiente manera:

- Ingresar al menú “Proyecto” y luego a “Propiedades <nombre del proyecto>” tal como se muestra en la Fig. 6.

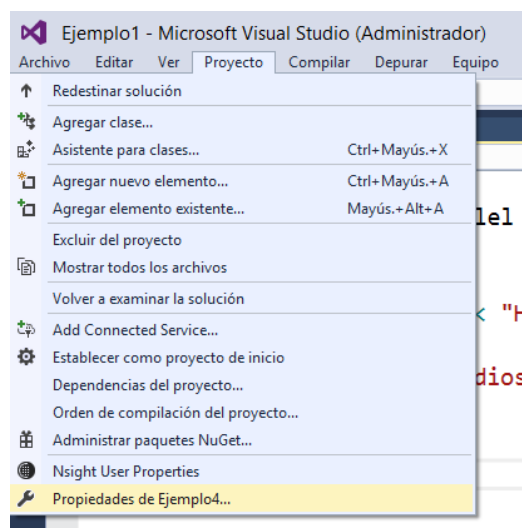


Fig. 6. Menú para configurar propiedades del proyecto.

Luego modificar “Idioma” como se muestra en la Fig. 7.

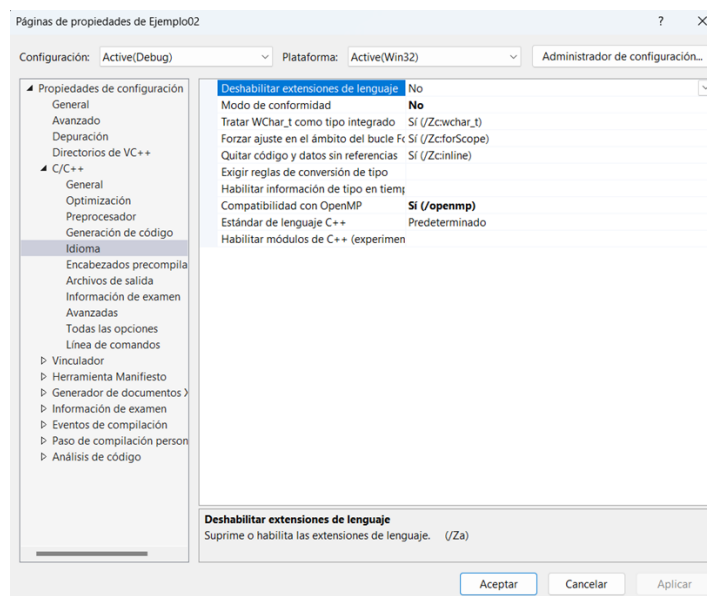


Fig. 7. Configurar Idioma.

### 3.2. ACTIVIDAD 2.

#### 3.2.1. La cláusula *num\_threads(n)*.

En cada región paralela hay un número de hilos generados por defecto y ese número es igual al de unidades de procesamiento que se tengan en la computadora paralela que se esté utilizando, en este caso el número de núcleos que tenga el procesador.

La cláusula *num\_threads(n)* permite establecer el número que se ejecutarán.

#### 3.2.2. Ejercicio 3.

Modificar el ejemplo del ejercicio 2, para que se ejecuten 5 hilos agregando la cláusula:

```
#pragma omp parallel num_threads(5)

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        std::cout << "Hola Mundo...\n" << std::endl;
    }

    std::cout << "Adios...\n" << std::endl;
    getchar();
    return 0;
}
```

### 3.3. ACTIVIDAD 3.

#### 3.3.1. Condición de Carrera.

En la programación paralela en computadoras que tienen memoria compartida puede presentarse la llamada condición de carrera (*race condition*) que ocurre cuando varios hilos tienen acceso a recursos compartidos sin control. El caso más común se da cuando en un programa varios hilos tienen acceso concurrente a una misma dirección de memoria (variable) y todos o algunos en algún momento intentan escribir en la misma localidad al mismo tiempo. Esto es un conflicto que genera salidas incorrectas o impredecibles del programa.

En OpenMP al trabajar con hilos se sabe que hay partes de la memoria que comparten entre ellos y otras no. Por lo que habrá variables que serán compartidas entre los hilos, (a las cuales todos los hilos tienen acceso y las pueden modificar) y habrá otras que serán propias o privadas de cada uno.

Dentro del código se dirá que cualquier variable que esté declarada fuera de la región paralela será compartida y cualquier variable declarada dentro de la región paralela será privada.

### 3.3.2. Ejercicio 4.

En el siguiente ejercicio, cada hilo ejecuta un ciclo 10 veces imprimiendo la variable  $i$ . En cada hilo que se ejecute la variable  $i$  tomará valores desde 0 hasta 9 porque la variable  $i$  es una variable propia de cada hilo.

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
#pragma omp parallel num_threads(1)
{
    std::cout << "Imprime ciclo...!\n" << std::endl;

    int i; //--variable local
    for (i = 0; i < 10; i++)
    {
        std::cout << "unsaac " << i << std::endl;
    }
    std::cout << "Fin de la impresión de ciclos...!\n" << std::endl;
    getchar();
    return 0;
}
```

### 3.3.3. Ejercicio 5.

En el siguiente ejercicio la definición de la variable  $i$  trasladamos antes de la región paralela, por lo tanto, ahora la variable  $i$  se convierte en global a la que todos los hilos tienen acceso.

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    int i;
#pragma omp parallel num_threads(3)
{
    for (i = 0; i < 10; i++)
    {
        std::cout << "unsaac " << i << std::endl;
    }
    std::cout << "Fin de la impresión de ciclos...!\n" << std::endl;
    getchar();
    return 0;
}
```

En el resultado de la ejecución del programa anterior se nota que la variable global  $i$  toma valores impredecibles.

### 3.4. ACTIVIDAD 4.

Existen dos cláusulas que pueden forzar a que una variable privada sea compartida y una compartida sea privada y son las siguientes:

#### 3.4.1. La cláusula `shared()`.

Las variables colocadas separadas por coma dentro del paréntesis serán compartidas entre todos los hilos de la región paralela. Sólo existe una copia, y todos los hilos acceden y modifican dicha copia.

#### 3.4.2. La cláusula `private()`.

Las variables colocadas separadas por coma dentro del paréntesis serán privadas. Se crean  $p$  copias, una por hilo, las cuales no se inicializan y no tienen un valor definido al final de la región paralela ya que se destruyen al finalizar la ejecución de los hilos.

#### 3.4.3. Ejercicio 6.

En el siguiente ejercicio se fuerza a que la variable  $i$  sea compartida

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    int i;
    #pragma omp parallel num_threads(3), shared(i)
    {
        std::cout << "Imprime ciclo...!\n" << std::endl;
        for (i = 0; i < 10; i++)
        {
            std::cout << "unsaac " << i << std::endl;
        }
    }
    std::cout << "Fin de la impresión de ciclos...!\n" << std::endl;
    getchar();
    return 0;
}
```

#### 3.4.4. Ejercicio 7.

En el siguiente ejercicio se fuerza a que la variable  $i$  sea privada

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    int i;

    #pragma omp parallel num_threads(3), private(i)
    {

        std::cout << "Imprime ciclo...!\n" << std::endl;

        for (i = 0; i < 10; i++)
        {
```



```

        std::cout << "unsaac " << i << std::endl;
    }
}
std::cout << "Fin de la impresión de ciclos...\n" << std::endl;
getchar();
return 0;
}

```

### 3.5. ACTIVIDAD 5.

Dentro de una región paralela hay un número de hilos generados y cada uno tiene asignado un identificador. Estos datos se pueden conocer durante la ejecución con la llamada a las funciones de la biblioteca *omp\_get\_num\_threads()* y *omp\_get\_thread\_num()* respectivamente.

#### 3.5.1. Ejercicio 8.

En el siguiente ejemplo para su buen funcionamiento se debe indicar que la variable *tid* sea privada dentro de la región paralela, ya que de no ser así todos los hilos escribirán en la dirección de memoria asignada a dicha variable sin un control (race condition), es decir “competirán” para ver quién llega antes y el resultado visualizado puede ser inconsistente e impredecible.

```

#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    int tid, nth;

    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        nth = omp_get_num_threads();

        printf("Este es el hilo %d de un total de %d\n", tid, nth);
    }

    getchar();
    return 0;
}

```

## 4. TAREA

Se requiere realizar la suma de dos arreglos unidimensionales de N elementos de forma paralela utilizando solo dos hilos. Para ello se utilizará un paralelismo de datos o descomposición de dominio, es decir, cada hilo trabajará con diferentes elementos de los arreglos a sumar, pero ambos utilizarán el mismo algoritmo para realizar la suma.



Fig. 9. Suma de vectores paralela

## 5. BIBLIOGRAFIA.

- Barbara Chapman, 2008 Using OpenMP. Masachuset Institute Technology.
- José E. Román. et. al. 2018. Ejercicios de Programación Paralela con OpenMP y MPI. Editorial de la Universidad Politécnica de Valencia.
- Barry Wilkinson. 2005. Parallel Programming. 2nd.Edition, Pearson Education USA.
- <https://www.openmp.org/>
- <https://mapecode.com/programacion-paralela-con-openmp/>
- <http://javierferrer.me/paralelizar-c-openmp-introduccion/>