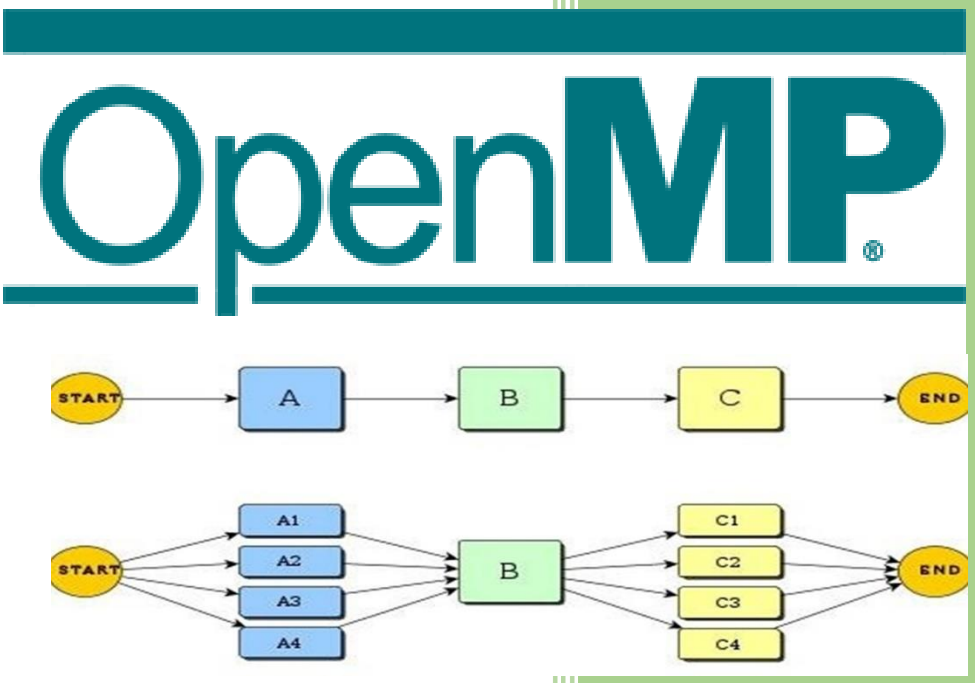
Estudio del API OpenMP



SAMEH ARMOUCHE

Y2951360Z

Ingeniería de computadores

Índice:

Contenido

Índice: 1
Introducción: 2
Directivas: 3
Clausulas: 5
Funciones: 6
Ejemplos: 7
Ej1: 7
Ej2: 8
Ej3: 9
EJ4: 10
EJ5: 11
Conclusión: 12
Referencias: 13

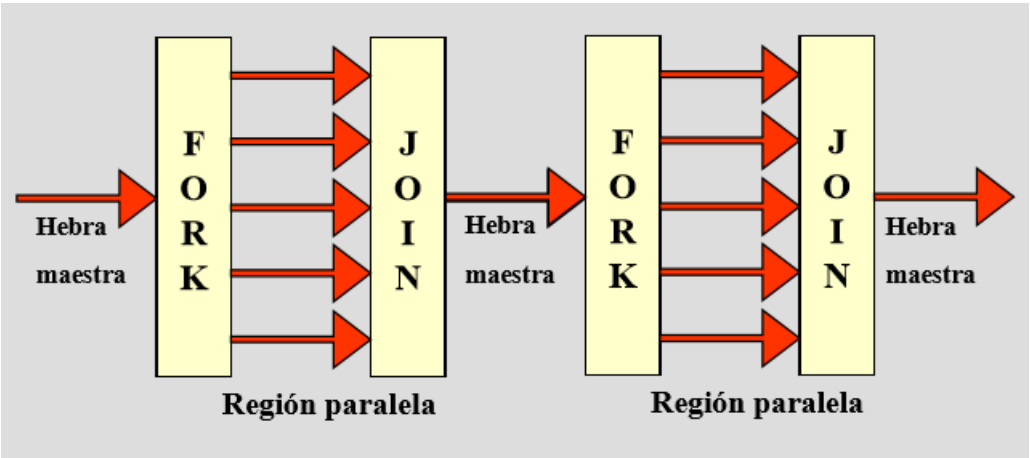
Introducción:

Es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join.

Está disponible en muchas arquitecturas, incluidas las plataformas de Unix y de Microsoft Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influyen el comportamiento en tiempo de ejecución.

Una aplicación construida con un modelo de programación paralela híbrido se puede ejecutar en un clúster de computadoras utilizando OpenMP y MPI, o a través de las extensiones de OpenMP para los sistemas de memoria distribuida.

OpenMP se basa en el modelo fork-join, paradigma que proviene de los sistemas Unix, donde una tarea muy pesada se divide en K hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un solo resultado (join):



Directivas:

- **Directivas #pragma de compilación:**
  Informan al compilador para optimizar código #pragma omp <directiva> {<cláusula>}* <\n>

For compartido en paralelo:

- **parallel:**
  Define una región paralela, que es un código que será ejecutado por múltiples hilos en paralelo.
- **for:**
  Hace que el trabajo realizado en un bucle for dentro de una región paralela se divida entre hilos.
- **sections:**
  Identifica secciones de código que se dividirán entre todos los hilos.
- **single:**
  Permite especificar que una sección de código debe ejecutarse en un solo hilo, no necesariamente el hilo maestro.
- **task:**
  Permite la ejecución asíncrona mediante tareas explícitas. El bloque de código dentro de la directiva task es ejecutado por un único hilo, y la tarea es instanciada por el hilo que se encuentra con la directiva. Por ello es recomendable utilizar esta directiva dentro de una directiva single, evitando crear (y ejecutar) la misma tarea tantas veces como hilos haya. Su formato es: #pragma omp task [cláusulas]. Cuando la tarea se instancia ésta es insertada en una cola de pendientes. Las tareas de la cola de pendientes son seleccionadas por la implementación de OpenMP en función de sus dependencias cumplidas y prioridad. Las tareas seleccionadas pasan a la cola de listas para ejecutar. Los hilos del equipo activo cogen tareas de dicha cola y las ejecutan.

For maestro y sincronización:

- **master:**
  Especifica que solo el hilo maestro debe ejecutar una sección del programa.
- **critical:**
  Especifica que el código solo se ejecuta en un hilo a la vez.

- **barrier:**
  Sincroniza todos los hilos en un equipo; todos los hilos se detienen en la barrera, hasta que todos los hilos ejecutan la barrera.
- **atomic:**
  Especifica que una ubicación de memoria se actualizará atómicamente.
- **flush:**
  Especifica que todos los hilos tienen la misma vista de memoria para todos los objetos compartidos.
- **ordered:**
  Especifica que el código bajo un ciclo for paralelo debe ejecutarse como un ciclo secuencial.

Clausulas:

Vamos a poner algunas de las clauslulas mas importante del openmp en c++ :

- **num_thread:**
  Especifica el número de hilos a ejecutar en paralelo.

- **omp_set_dynamic:**
  La llamada a la rutina omp_set_dynamic con el argumento 0 en C / C ++, deshabilita el ajuste dinámico del número de subprocesos en las implementaciones de OpenMP que lo admiten.

- **nowait:**
  Si hay múltiples bucles independientes dentro de una región paralela, puede usar la cláusula nowait para evitar la barrera implícita al final de la construcción del bucle.

- **SHARED (lista):**
  En el caso de la necesidad de definir variables compartidas este clausula declara compartidas las variables de la lista.

- **schedule(omp_sched_t tipo, int chunk_size):**
  Esto es útil si la carga procesal es un bucle do o un bucle for. Las iteraciones son asignadas a los threads basándose en el método definido en la cláusula.

- **Firstprivate:**
  Especifica que cada subproceso debe tener su propia instancia de una variable y que la variable debe inicializarse con el valor de la variable antes de la región de paralización, porque existe antes de la construcción paralela.

Funciones:

- **omp_get_max_threads():**
  Devuelve el número máximo de hilos de la actual región paralela.

- **omp_get_num_procs():**
  Cuántos procesadores (núcleos) hay disponibles en el dispositivo actual.

- **omp_get_num_threads():**
  Cuántos hilos forman el equipo de hilos activo actual.

- **omp_get_num_devices():**
  Cuántos dispositivos aceleradores hay.

- **omp_set_num_threads(int num_threads):**
  Configura el número de hilos máximo en un equipo de hilos.

- **omp_set_schedule(omp_sched_t kind, int chunk_size):**
  Configura el método de planificación (schedule) por defecto.

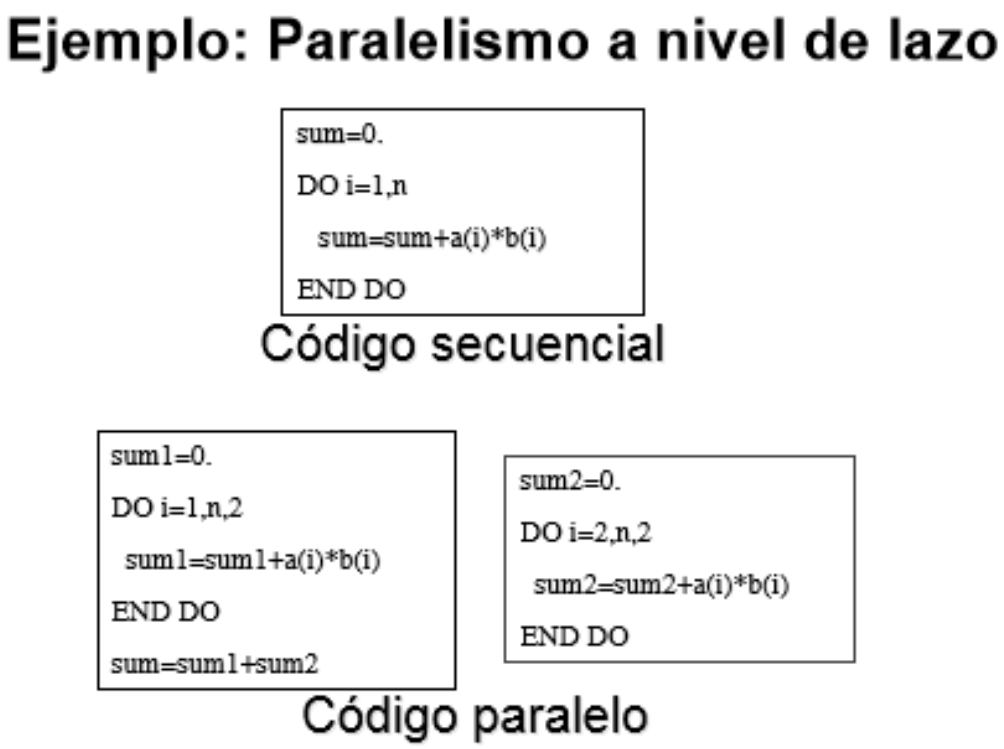
- **omp_get_wtick:**
  Obtener la precisión del cronómetro: cuántos segundos pasan entre dos ticks de reloj.

- **omp_get_wtime:**
  Obtener el tiempo del cronómetro.

Ejemplos:

Ej1:

Para empezar, vamos a poner un ejemplo sencillo en Fortran para entender más o menos el concepto del paralelismo.



En el ejemplo anterior, estamos haciendo una suma de la multiplicacion de componentes de vectores que son a y b.

Ej2:

Avanzando más, ya que tenemos que acercarlo a algún ejemplo concreto en c++:

parallel:

Esta directiva nos indica que la parte de código que la comprende puede ser ejecutada por varios hilos. Se crea una tarea implícita para cada hilo perteneciente al equipo de hilos creado por el parallel.

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Este código divide la inicialización de la tabla en múltiples subprocesos, que se ejecutan simultáneamente. Cada hilo inicializa una parte de la tabla.

Ej3:

El siguiente ejemplo muestra cómo paralelizar un bucle simple utilizando la construcción parallel loop. La variable de iteración de bucle es privada de forma predeterminada, por lo que no es necesario especificarla explícitamente en una cláusula privada:

```
void simple(int n, float *a, float *b)
{
    int i;

#pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Esta función almacena en el vector b la media de cada dos componentes del vector a todos los vectores en float para dar más potencia a la operación realizada.

EJ4:

En este fragmento de código se trata de explotar el potencial del paralelismo en una tarea de una serie geométrica que tiende al infinito para la ecuación:

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots = \sum_{n=1}^{\infty} \frac{1}{n}$$

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

#define LIMIT 1000000000000

int main(void)
{
        double serie = 0;
        long long i;
        #pragma omp parallel
        {
                #pragma omp for reduction(+:serie)
                for(i = 1;i <= LIMIT;i++)
                serie += 1.0/i;
        }

        printf("Valor de la Serie: %.5f\n",serie);
return 0;
}
```

EJ5:

Como hemos puesto en la sección de directivas es recomendable utilizar task dentro de una directiva single, vamos a poner un ejemplo donde se usan las dos directivas para entender el funcionamiento:

```
#define NUM_ITERS 1000000
#pragma omp parallel
{
    int32_t BS =  NUM_ITERS/omp_get_num_threads();
    #pragma omp single
    {
        for (int i = 0; i < NUM_ITERS; i+= BS) {
            int32_t upper = i+((NUM_ITERS-BS >= 0) ? BS : NUM_ITERS-BS);
            #pragma omp task firstprivate(i,upper)
            {
                for (int ii = i; ii < upper; ii++)
                    a[ii] += b[ii]*c[ii];
            }
        }
        #pragma omp taskwait
    }
}
```

Aquí se usan para evitar crear (y ejecutar) la misma tarea tantas veces como hilos haya.

Conclusión:

El mundo de openmp es mucho más extenso y es recomendable profundizar en el cual porque a la hora de observar el rendimiento de las tareas comparado con el secuencial la ganancia en muchas ocasiones es brutal.

Realmente este tutorial era muy resumido para animar a usar el paralelismo y entender mas o menos el concepto junto con la API openmp.

En la sección de referencias podéis consultar los enlaces puestos para practicar ejemplos experimentar vuestros ejemplos como funcionan a la hora de paralelizar y mucho más.

Referencias:

https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf

https://es.wikipedia.org/wiki/OpenMp

https://computing.llnl.gov/tutorials/openMP