Trabajo Optativo

Paralelización de programas con OpenMP

Miguel Ángel Sempere Vicente
Carlos Eduardo Fernández García

Sistemas Empotrados
Grado en Ingeniería Robótica
Escuela Politécnica Superior, Universidad de Alicante
Curso académico 2020-2021

Universitat d'Alacant
Universidad de Alicante

1. Introducción

El paralelismo es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente. El paralelismo está basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños que son resueltos posteriormente en paralelo.

Cabe remarcar que no es lo mismo hablar de concurrencia que de paralelismo, pese a tratarse de conceptos bastante relacionados. La concurrencia es la capacidad del microprocesador para realizar más de un proceso o tarea al mismo tiempo. La computación concurrente es una forma de computación en la que se realizan varios cálculos de manera concurrente y no de forma secuencial. En la computación concurrente, ya sea en un programa, en un computador o en una red, existen de forma separada un hilo de control para cada proceso. El CPU es capaz de procesar de forma simultánea tantos procesos como núcleos tenga y no tienen porqué estar relacionados entre sí; en el caso de un procesador single-core, puede realizar una única tarea y, cuando acabe, se empieza otra; si, por ejemplo, el procesador tiene cuatro núcleos, se pueden ejecutar hasta cuatro procesos al mismo tiempo.

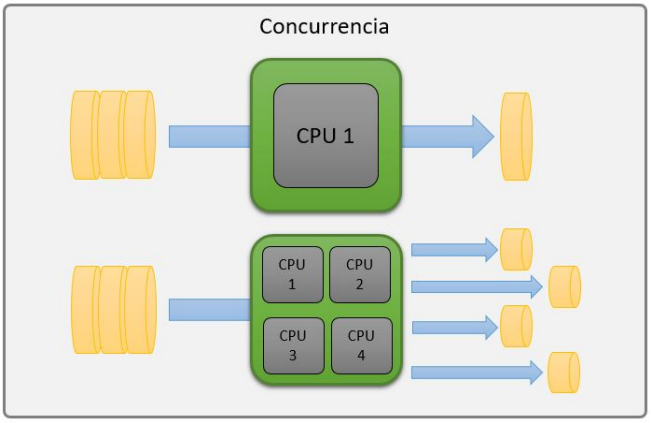

Figura 1. Esquema de ejecución concurrente

Por otro lado, el paralelismo se basa en la premisa de "divide y vencerás", consiste en tomar un problema único y utilizar la concurrencia para obtener la solución de forma más rápida. Se toma el problema principal y se divide en subproblemas más pequeños, al igual que en muchos algoritmos ampliamente utilizados en programación; después, cada fracción del problema inicial es procesada de forma concurrente, aprovechando así la máxima capacidad del microprocesador. En este caso los procesos a resolver sí que se encuentran relacionados ya que, en esencia, todos ellos son parte del mismo problema inicial; es por esto que en el paralelismo es necesario un paso final para unir todos los resultados obtenidos de cada subproceso y poder así arrojar el resultado final.

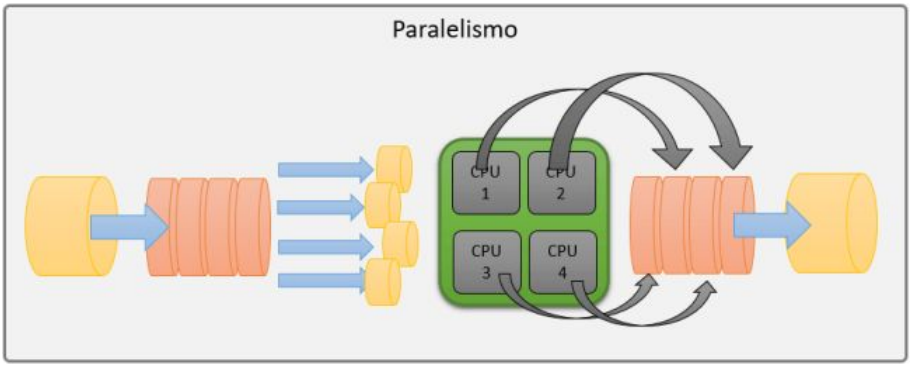

Figura 2. Esquema de ejecución en paralelo

El paralelismo permite la resolución de problemas de alto coste temporal y es ampliamente utilizado hoy en día en sectores como la economía, las matemáticas, la informática, etc. Existen varios tipos de paralelismo:

- nivel de bit
- nivel de instrucción
- nivel de datos
- nivel de tarea

2. Conceptos OpenMP

OpenMP es una API (interfaz de programación de aplicaciones) que permite programar aplicaciones multihilo simplificando en gran medida la tarea de escribir código de este tipo ya que permite añadir concurrencia a los programas escritos en lenguajes como C, C++ y Fortran. OpenMP está compuesto de directivas de compilador, una librería de rutinas, y variables de entorno que influyen en el comportamiento en tiempo de ejecución; además, se encuentra disponible para muchas arquitecturas. Se trata de un modelo SMP (*Shared Memory Programming*) portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, para plataformas que van desde los ordenadores de escritorio hasta supercomputadoras

La siguiente es una lista no exhaustiva de los principales compiladores que soportan OpenMP junto con la *flag* que lo habilita.

- GCC (incluidos gcc, g ++ y gfortran): -fopenmp
- LLVM: -fopenmp
- Compilador-suite de Intel (incluidos icc, icpc e ifort): -qopenmp
- IBM XL compilador-suite (incluyendo xlc, xlC y xlf): -xlsmp=omp
- PGI compiler-suite (incluyendo pgcc pgc ++ pgfortran): '-mp'

A continuación, se detallan los conceptos básicos de OpenMP, cuya estructura está dividida en directivas (constructores), cláusulas, sincronización, funciones y variables de entorno.

2.1. Directivas (constructores)

En este apartado se muestran las diferentes directivas o constructores de OpenMP, así como su descripción y un pequeño ejemplo de cada una de ellas. En OpenMP, las directivas son especificadas con la etiqueta #pragma del lenguaje C estándar.

- **Parallel**

Este constructor indica que el bloque de código que comprende puede ser ejecutado de manera simultánea por varios hilos. La sintaxis es la siguiente:

#pragma omp parallel [clause ...] **newline**

⋮

structured_block

Dónde **clause** puede adquirir una de los siguientes expresiones:

- **if ([parallel :]** scalar-expresion **)**
- **num_threads(**integrer-expresion**)**
- **defatult(none | shared | private)**
- **private(** list **)**
- **firstprivate (** list **)**
- **shared (** list **)**
- **copyin (** list **)**
- **reduction (** reduction-modifier **,**] reduction-identifier **:** list**)**
- **proc_bind (master | close | spread)**
- **allocate (**[allocator **:**] list**)**

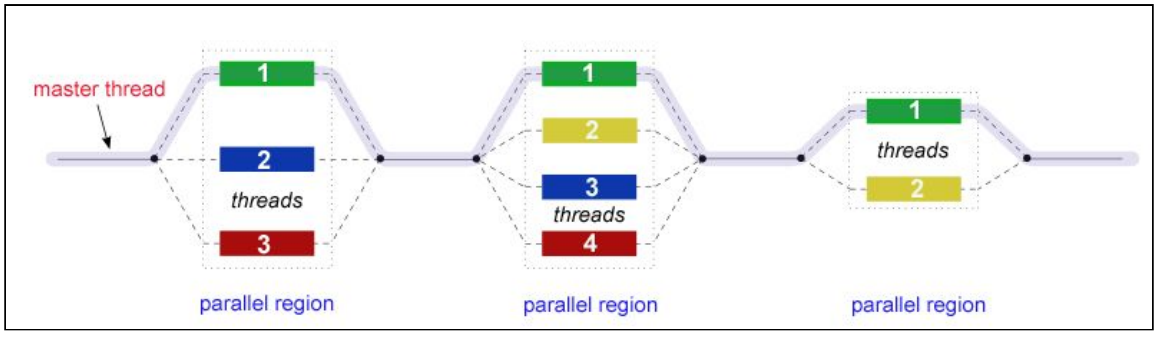

Figura 3. División en diferentes hilos del programa

En la figura 3 podemos ver como un hilo principal de ejecución se va dividiendo en varios hilos que se van solucionando en paralelo y no de manera secuencial como en un programa común.

- **For**

Este constructor permite dividir un bucle "for" en pedazos, también llamados "chunks", para que en distintos hilos, sean ejecutados paralelamente . La sintaxis es la siguiente:

#pragma omp for [clause ...] **newline**

⋮

for_loop

Donde **clause** puede adquirir una de los siguientes expresiones:

- **schedule (**type, [chunk]**)**
- **ordered**
- **private (**list**)**
- **firstprivate (**list**)**
- **lastprivate (**list**)**
- **shared (**list**)**
- **reduction (**operator: list**)**
- **collapse (**n**)**
- **nowait**

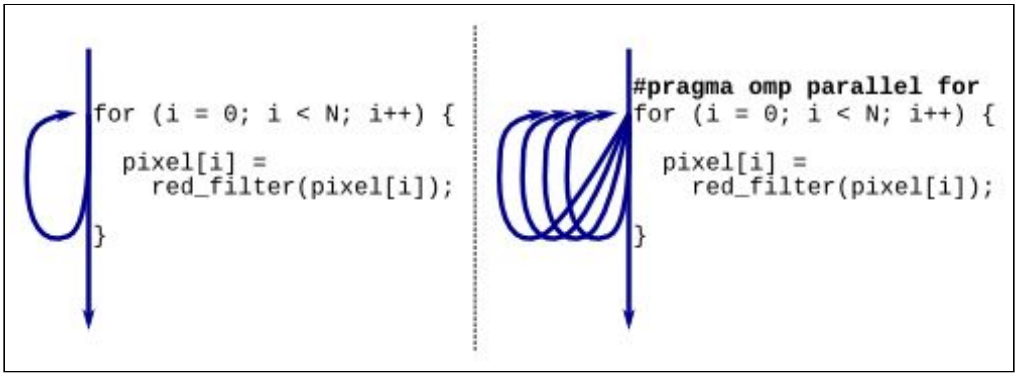

Figura 4. Hilos de programa de un bucle "for"

Como bien representa la Figura 4 por cada iteración del bucle "for" se genera un bucle por cada hilo de programa, esto permite una mayor velocidad de procesamiento comparado con los lenguajes estándares sin OpenMP (C++)

- **Sections**

Este constructor encapsula código en diferentes secciones que se dividirán entre los distintos hilos de los que se dispongan. En caso de que hayan más hilos que secciones, habrán hilos que no realizará ninguna tarea, si por el contrario hay más secciones que hilos, la ejecución del programa será distinta dependiendo de su implementación. La sintaxis de este constructor es la siguiente:

```
#pragma omp sections [clause ... ] newline
  ⋮
{
#pragma omp section newline
 structured_block

 #pragma omp section newline
  structured_block
  }
```

Dónde **clause** pueden ser las siguientes:

- **private (**list**)**
- **firstprivate (**list**)**
- **lastprivate (**list**)**
- **reduction (**operator: list**)**
- **nowait**

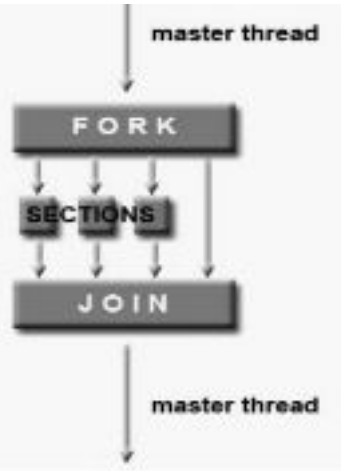

Figura 5. Encapsulación de código con “sections”

Este constructor crea un bloque estructurante para cada hilo del subproblema en el que se divide el problema, de manera que cada hilo de "secction" funciona como si fuera un problema completamente independiente.

- **Single**

Este constructor permite que una parte del código determinada sea ejecutada por un solo hilo, haciendo que el resto de los hilos del programa esperan a que este finalice. La sintaxis de este constructor es la siguiente:

#pragma omp single [clause ...] **newline**
⋮
structured_block

Las **clause** que tiene disponibles son las siguientes:

- **private (**list**)**
- **firstprivate (**list**)**
- **nowait**

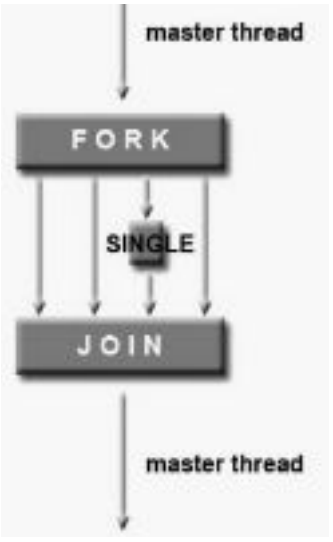

Figura 6. Esquema funcionamiento constructor "single"

El constructor “single” aporta un mayor control sobre la ejecución del programa al permitir que algunos fragmentos del código no sean divisibles en diferentes hilos.

- **Task**

Este constructor determina una tarea ya sea dentro o no de un constructor single. Si se encuentra dentro de un constructor single se paralelizará cada hilo a una tarea, pero si por el contrario se encuentra fuera de un constructor single, se ejecutará tantas veces como número de hilos hayan disponibles. Este constructor está orientado para su uso en algoritmos recursivos. La sintaxis del constructor es la siguiente:

#pragma omp task [clause ...] **newlie**
⋮
structured_block

Dónde **clause** se puede sustituir por las siguientes opciones:

- **if** (scalar expression)
- **final** (scalar expressions)
- **untied**
- **default** (shared | none)
- **mergeable**
- **private** (list)
- **firstprivate** (list)
- **shared** (list)

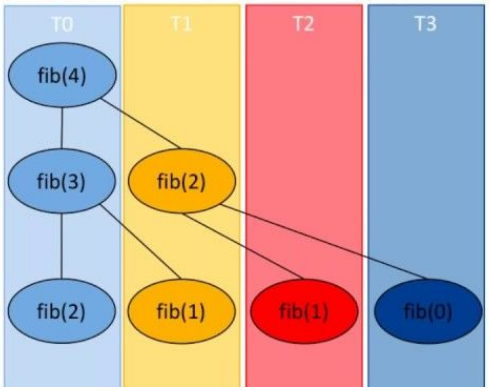

Figura 7. Ejemplo Fibonacci con varios “tasks”

El usuario puede evitar el cambio de hilo mediante tareas vinculadas, siempre y cuando los puntos de programación de las tareas estén bien definidos. Como se puede apreciar en la figura 7 mediante distintas tareas o "task" se puede obtener la sucesión de Fibonacci.

2.2. Cláusulas

Muchas de las directivas de OpenMp soportan el uso de las denominadas cláusulas, que se utilizan para especificar información adicional de la directiva que acompañan. Las cláusulas que incluye una directiva afectan únicamente al ámbito estático de la región, esto significa que, en una región paralela, sólo afecta al bloque de código que comprende (extensión estática) y no a todo el código que ejecuta (extensión dinámica). Las cláusulas de atributos de alcance de datos de OpenMP se utilizan para controlar el alcance de las variables incluidas, es decir, permiten controlar el entorno de datos durante la ejecución de construcciones paralelas. Las principales cláusulas son:

- **shared(var):** se utiliza para declarar una variable "var" cómo compartida para todos los hilos, es decir, existe una única copia de la variable y todos los threads del equipo acceden a la misma dirección de memoria de dicha variable para modificarla.
- **private(var):** se utiliza para declarar una variable "var" como privada en cada uno de los threads, se crean tantas copias como hilos y todas ellas se destruyen al finalizar la ejecución. Todas las variables privadas se encuentran NO-inicializadas tanto a la entrada como a la salida de la región paralela, es decir, no existe relación entre las mismas variables dentro y fuera de la región (y obviamente no tienen el mismo valor); para sobreescribir este comportamiento se utilizan las cláusulas firstprivate o lastprivate, detalladas a continuación.
- **firstprivate(var):** las variables afectadas por esta cláusula son inicializadas con el valor original que tenían antes de entrar en la región paralela.
- **lastprivate(var):** el hilo que ejecuta la última iteración o sección, actualiza el valor de las variables afectadas por esta cláusula, por lo que permite puede "extraer" sus valores si se desea.

En este ejemplo se observa una combinación de uso de las cláusulas first/lastprivate y private, que están estrechamente relacionadas.

```
main()
{
  A = 10;

#pragma omp parallel
{
  #pragma omp for private(i) firstprivate(A) lastprivate(B)...
  for (i=0; i<n; i++)
  {
      ....
      B = A + i;          /*-- A undefined, unless declared
                              firstprivate --*/
      ....
  }
  C = B;                  /*-- B undefined, unless declared
                              lastprivate --*/

} /*-- End of OpenMP parallel region --*/

}
```

Figura 8. Ejemplo de las cláusulas de tipo privado

- reduction(var): se utilizan para realizar operaciones de reducción en regiones paralelas; solo pueden afectar a variables compartidas y se usa cuando se está efectuando alguna operación de acumulación. Su principal cometido es evitar que las actualizaciones del valor de la variable acumulativa sean correctas y no se realicen de forma errónea debido a la ejecución paralela. En el ejemplo siguiente podemos observar un ejemplo de uso de esta cláusula afectando a la variable "sum".

```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

Variable SUM is a shared variable

Figura 9. Ejemplo de uso de la cláusula *reduction*

- if(expression): especifica si una región se ejecuta de forma paralela; si la expresión se evalúa como true (!0), el código se ejecutará de forma paralela, de lo contrario, lo hará en serie.

- default(none/shared): si esta cláusula adopta un valor "none", obliga a declarar explícitamente el ámbito de cada variable (buena práctica de programación para no olvidarse variables). Si su valor es "shared", al igual que por defecto en OpenMP, todas las variables son compartidas. Aunque en C/C++ no está disponible, en Fortran esta cláusula puede adoptar el valor "private" y, como su propio nombre indica, todas las variables declaradas son privadas al hilo.

- copyin(var): permite copiar en cada hilo el valor de la variable "var" en el hilo maestro al comienzo de la región paralela.

- num_threads(num): establece el número de hilos a utilizar en la región paralela; tiene la misma funcionalidad que la función omp_set_num_threads() o la variable de entorno OMP_NUM_THREADS, ambas detallan en apartados posteriores.

- nowait: esta cláusula sólo está soportada por algunas directivas de OpenMP, y se utiliza para minimizar la sincronización; si está presente, los hilos no se sincronizan al final de cada región.

2.3. Sincronización

La sincronización se utiliza para imponer limitaciones de orden y proteger el acceso a las variables compartidas. OpenMP proporciona diferentes mecanismos de sincronización entre hilos que se pueden dividir entre sincronización de alto y bajo nivel.

- **Sincronización de alto nivel**

- critical (exclusión mutua): mediante el uso de #pragma omp critical se define una parte del código que no puede ser ejecutada por más de un hilo a la vez, por lo que el resto de hilos esperan su turno. Esta sección crítica debe ser lo más breve posible,

además, pueden definirse distintas secciones críticas con diferentes nombres denominadas "named".

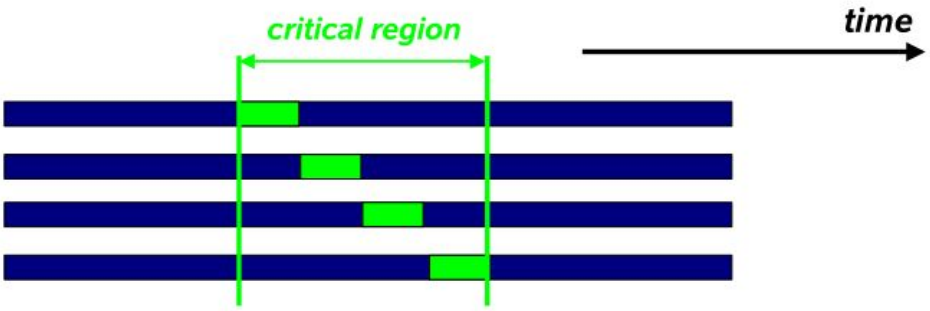

Figura 10. Esquema ejecución paralela con región *critical*

- atomic (exclusión mutua): es un caso particular de la sección critical en el que se lleva a cabo una operación acumulativa sencilla, por lo que tiene menor sobrecarga que la sincronización *critical* y por tanto mayor eficiencia.

- barrier (sincronización por eventos): permite definir barreras de sincronización global para todos los hilos de la región paralela, es decir, punto donde todos los threads se detienen a esperar y sólo continuarán cuando todos los hilos hayan alcanzado la barrera. En muchos casos, los constructores de OpenMP ya cuentan con una barrera al final de la región paralela (a no ser que sea de tipo *nowait*). Las barreras se utilizan cuando los datos están siendo actualizados de forma asíncrona, poniendo en riesgo la integridad de los mismos; sin embargo, no es aconsejable abusar de ellas, ya que tienden a ser muy costosas si se está trabajando con un número alto de procesadores.

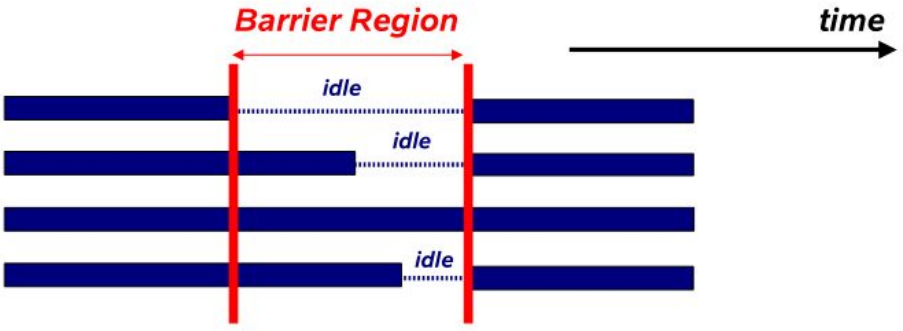

Figura 11. Esquema ejecución paralela con barreras

- ordered: la región de código que comprende es ejecutada en el orden en el que las iteraciones se ejecutarán de forma secuencial; es una forma de sincronización de hilos que resulta costosa.

- single (constructor): también es posible hacer uso del constructor *single* previamente analizado para realizar la sincronización de hilos; normalmente, es necesario colocar una barrera después de esta región. El uso de single es recomendable para la inicialización de entradas y salidas del programa.

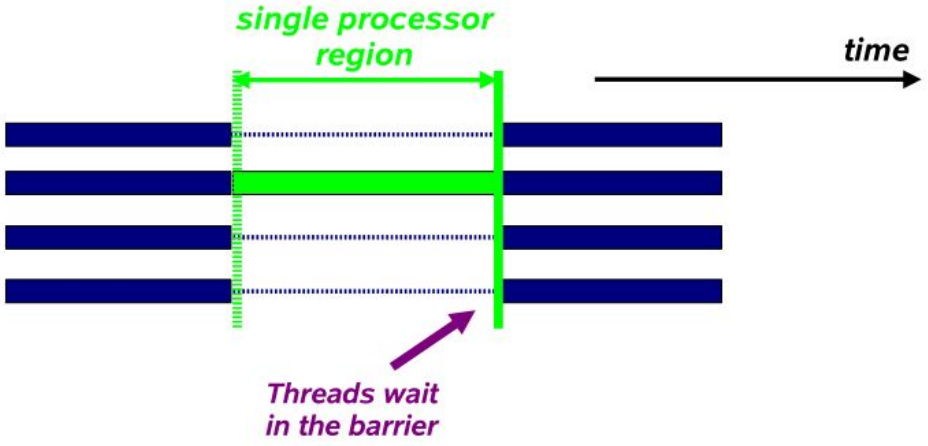

Figura 12. Esquema ejecución paralela con el constructor *single*

- **Sincronización de bajo nivel**

- flush(var): define un punto donde se garantiza que cada todos los subprocesos tienen la misma vista de memoria para todos los objetos compartidos. *Flush* fuerza la actualización de las variables especificadas como parámetro para que todos los hilos puedan ver el valor más reciente ellas.

```
double A;
A = compute();
flush(A);   // flush to memory to make sure other
            //  threads can pick up the right value
```

Figura 13. Ejemplo de sincronización con *flush*

- locks: se utilizan para proteger y "bloquear" recursos, esto permite a cada hilo hacer su trabajo de forma independiente, sin ser afectado por el resto. Una vez un thread ha finalizado la tarea, se libera el bloqueo para dejar el turno al siguiente hilo; OpenMP proporciona funciones para controlar la inicialización, eliminación, testeo, etc, de cada *lock*. Los bloqueos se pueden anidar y los hilos siempre acceden a la copia más actual del *lock*, por lo que no es necesario usar *flush* para actualizar la variable de bloqueo.

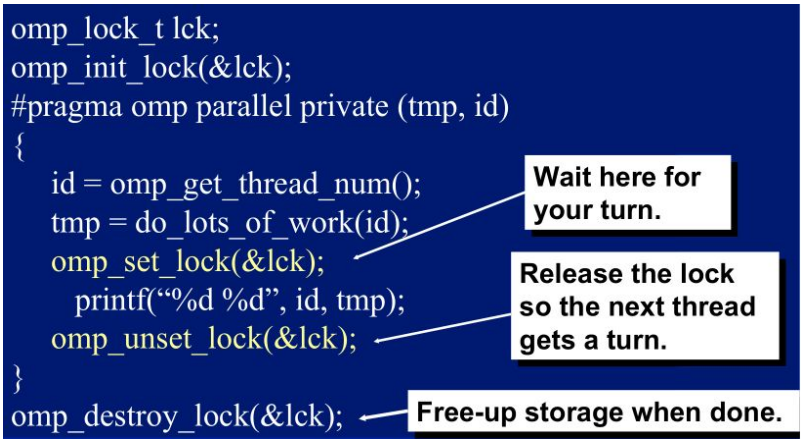

Figura 14. Ejemplo de sincronización usando *locks*

2.4. Funciones

OpenMP cuenta con una serie de funciones que permiten obtener información y configurar el entorno paralelo. Algunas de las principales funciones que se pueden encontrar en la librería de OpenMP son, entre muchas otras:

- omp_get_active_level: devuelve el número de regiones paralelas activas.
- omp_set_num_threads: configura el número máximo de hilos.
- omp_get_num_threads: devuelve el número de hilos activos del equipo actual.
- omp_get_thread_num: devuelve el identificador del hilo que invoca la función.
- omp_get_num_procs: devuelve el número de núcleos disponibles en el dispositivo.
- omp_in_parallel: chequea si el hilo se encuentra en una región paralela.
- omp_set_dynamic: activa/desactiva la regulación dinámica del tamaño de los equipos.

- omp_get_dynamic: chequea si la regulación dinámica de hilos se encuentra activa.
- omp_set_nested: activar o no regiones paralelas anidadas.
- omp_get_nested: chequea si las regiones paralelas anidadas están activadas.
- omp_get_wtime: devuelve el tiempo de ejecución del programa.
- omp_get_wtick: devuelve el tiempo entre ticks de reloj.

2.5. Variables de entorno

Los nombres de las variables de entorno de OpenMP, al igual que en la mayoría de los casos, están en mayúsculas y son "*case sensitive*", es decir, sensibles a mayúsculas y minúsculas. Aunque existen muchas funciones de tipo "*set*" para modificar distintos valores del entorno, OpenMP también permite hacerlo asignando los valores deseados a las variables correspondientes. A continuación, se muestran las diferentes variables de entorno junto con su tipo y la descripción de para qué sirven:

- OMP_DYNAMIC=bool: Especifica si el tiempo de ejecución de OpenMP puede ajustar el número de subprocesos de una región paralela.
- OMP_NUM_THREADS=num: Fija el número máximo de hilos simultáneos de la región paralela, a menos que se invalide por la función omp_set_num_threads() o la cláusula num_threads.
- OMP_SCHEDULE[=type[,size]]: Configura el tipo de schedule por defecto (solo en las directivas *for* y *parallel for*), es decir, como se programan las iteraciones del bucle en los procesadores.
- OMP_PROC_BIND=bool: Configura si se pueden mover los hilos entre procesadores.
- OMP_NESTED=bool: Activa el uso de regiones paralelas anidadas, a menos que el paralelismo anidado esté habilitado o deshabilitado con la función omp_set_nested().
- OMP_MAX_ACTIVE_LEVELS=num: Configura la máxima cantidad de regiones paralelas anidadas.
- OMP_THREAD_LIMIT=num: Configura el máximo número de hilos que se utilizan en todo OpenMP.

- OMP_WAIT_POLICY=(ACTIVE/PASSIVE): Configura el algoritmo de espera en las barreras (barrier).

- OMP_CANCELLATION=bool: Configura la posibilidad de utilizar la directiva #pragma omp cancel

- OMP_STACKSIZE=size: Configura el tamaño de la pila de cada hilo (no maestro).

- OMP_DISPLAY_ENV=bool: Muestra la versión de OpenMP y las variables de entorno.

3. Ejemplos de OpenMP

Para poder apreciar el potencial de OpenMP, se va a realizar una comparación de la ejecución de un programa simple en C++ empleando la librería estándar comparado con un programa en el mismo lenguaje empleando la librería de OpenMP.

C++

```
int valor_maximo
(vector<int>values){

  int maximo = values[0];
  int tamano = values.size();

  for (int i=0;i<tamano;i++){
     if (values[i] > max) {
        maximo = values[i];
     }
   }
   return maximo;
}
```

C++ OpenMP

```
int valor_maximo(vector<int>values){

   int maximo = values[0];
   int tamano = values.size();

   int chunks;
   int n_threads;

   #pragma omp parallel {
      #pragma omp single {
         n_threads = omp_get_num_threads();
         if (n_threads < tamano) {
            chunks = tamano / n_threads;
         }
         else {
            chunks = n_threads;
         }

         #pragma omp for schedule(static,chunks)
reduction(max:maximo){
            for (int i=0;i<tamano;i++) {
               if (values[i] > maximo) {
                  maximo = values [i];
               }
            }
         }
   }
   return maximo
}
```

Figura 15. Tabla comparativa C++/C++ con OpenMP

Ambos programas son una función a la que se le pasa un vector de enteros y se devuelve el mayor de sus valores. Los resultados del coste temporal de resolver el problema con las mismas condiciones para ambos códigos son las siguientes:

Tipo	**Tiempo** (s)
C++ estándar	7,4
C++ OpenMP	3,2

Figura 16. Resultados de las complejidades temporales

Por tanto se puede comprobar que el código empleando OpenMP, aunque sea más largo y lento de escribir, proporciona los resultados un **56,76%** más rápido que empleando C++ estándar.

Algunos ejemplos de códigos empleando OpenMP en los que se puede apreciar su sintaxis y comprender su funcionamiento son por ejemplo:

1. Impresión de "hello word"

Un programa muy utilizado para ilustrar la sintaxis básica de un lenguaje de programación cuando se está empezando es el "hola mundo". En la figura de debajo se puede ver su implementación empleando OpenMP

C++	C++ con OpenMP	Salida por pantalla OpenMP

#include <iostream>
 	
int main(){
 	
std::cout <<"hello world";
}
	#include "omp.h"
	
	void main(){
	
	#pragma omp parallel {
	
	int ID =
	omp_get_thread_num();
	printf ("hello(%d)", ID);
	printf ("world(%d), ID");
	}
	}
		hello(1) hello(0) worl(1)
		
		world(0)
		
		hello(3) hello(2) world(3)
		
		world(2)

Figura 17. Comparación de sintaxis y salida del código "hello world"

Los números que se encuentran dentro de los paréntesis de la "salida por pantalla" indican el hilo que se emplea.

2. Búsqueda de un elemento en un Array

En el siguiente fragmento de código se puede ver un código en C++ empleando OpenMP en el que se busca un valor determinado entre las diferentes posiciones que componen el vector.

C++	C++ con OpenMP

int (int vector[], int tam, int valor) {
for(int i=0;i<tam;i++){
int aux = vector[i];
if(aux == valor){
return i;
}
}
return -1;
}
	#pragma omp parallel private(i, id, p, load, begin, end) {
	
	p = omp_get_num_threads();
	id = omp_get_thread_num();
	load = N/p; begin = id*load; end = begin+load;
	for (i = begin; ((i<end) && keepon); i++) {
	if (a[i] == x) {
	keepon = 0;
	position = i;
	}
	#pragma omp flush(keepon)
	}
	}

Figura 18. Comparación de código C++/C++ con OpenMP

3. Multiplicación de un Array

En este ejemplo se muestra el código en C++ empleando OpenMP de la multiplicación de dos Arrays. Como se puede apreciar, la extensión del código es semejante, pero el código que emplea la librería OpenMP obtendrá la respuesta del problema más rápidamente.

C++	C++ con OpenMP

void mv (int m1[][], int m2[][], int mult[][], int row1, int col1, int row2, int col2) {
 	
int i, j, k;
 	
for(i = 0; i < row1; ++i) {
for(j = 0; j < col2; ++j){
mult[i][j] = 0;
}
for(i = 0; i < row1; ++i) {
for(j = 0; j < col2; ++j) {
for(k=0; k<col2; ++k){
mult[i][j] += m1[i][k] * m2[k][j];
}
}
}
}
}
	void mv(double *a, int fa,int ca,int lda,double *b,int fb,double *c,int fc){
	
	int i, j; double s;
	#pragma omp parallel {
	
	#pragma omp for private(i,j,s) schedule(static,BLOQUE){
	for (i = 0; i < fa; i++) {
	s=0;
	for(j=0;j<ca;j++){
	s+=a[i*lda+j]*b[j];
	}
	c[i]=s;
	}
	}
	}
	}

Figura 19. Comparación de código C++/C++ con OpenMP

Referencias

- https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- https://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf
- https://www.openmp.org/spec-html/5.0/openmp.html
- https://www.openmp.org//wp-content/uploads/openmp-examples-4.5.0.pdf
- https://www.openmp.org//wp-content/uploads/cspec20.pdf