Programación Multihebra con OpenMP

José Miguel Mantas Ruiz

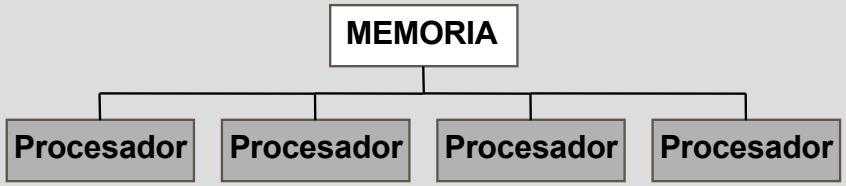
Depto. de Lenguajes y Sistemas Informáticos
Universidad de Granada

Contenidos

- Modelo de Programación e Introducción
- Paralelización de bucles
- Gestión de variables privadas
- Exclusión mutua
- Reducciones
- Paralelismo Funcional
- Sincronización en OpenMP
- Ejemplos
- Bibliografía y enlaces recomendados

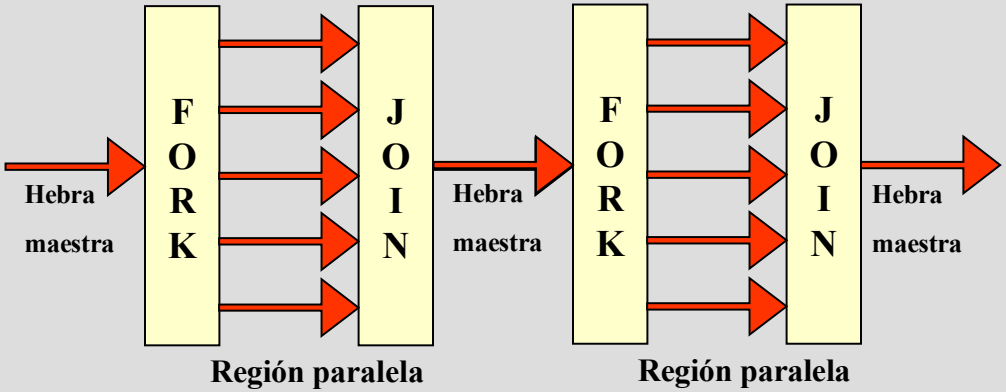
Modelo de Programación

- **Modelo de Programación de Memoria compartida**
  - Sincronización y Comunicac. mediante vars. compartidas



- **Paralelismo Fork/Join**

- **Soporta paralelización incremental**



El estándar OpenMP

- **Es una API, define sólo una interfaz.**
- **Expresa paralelismo multihilo en sistemas de memoria compartida.**
- **Componentes:**
  - Directivas ***#pragma*** de compilación
    - Informan al compilador para optimizar código

    #pragma omp <directiva> {<cláusula>}* <\n>
  - Funciones de librería
  - Variables de entorno

Paralelización de bucles

- **Un bucle es fácilmente paralelizable en OpenMP**
- **Reglas:**
  - No dependencia entre iteraciones
  - Prohibidas instrucciones **break, exit(), goto,...**
  - **Forma CANÓNICA**

for(i=*INICIO*; i { **<** , **<=** , **>=** , **>** } ***FINAL*;** { **i++** , **++i** , **i--** , **--i** , **i+= *inc*** , **i-= *inc*** , **i= i + *inc*** , **i= *inc* + i** , **i= i - *inc*** } **)**

Paralelización de bucles (2)

```
#pragma omp parallel for
  for ( i= primero; i< ultimo; i+= incr )
     celda[ i ]=  TRUE;
```

- **Hebra maestra crea hebras adicionales para cubrir las iteraciones del bucle**
- **El ámbito de cada directiva OpenMP es el del bloque que hay justo a continuación.**

  - Una sola instrucción es un bloque.
  - Una sentencia for define su propio bloque.
  - Un bloque convencional entre { }.

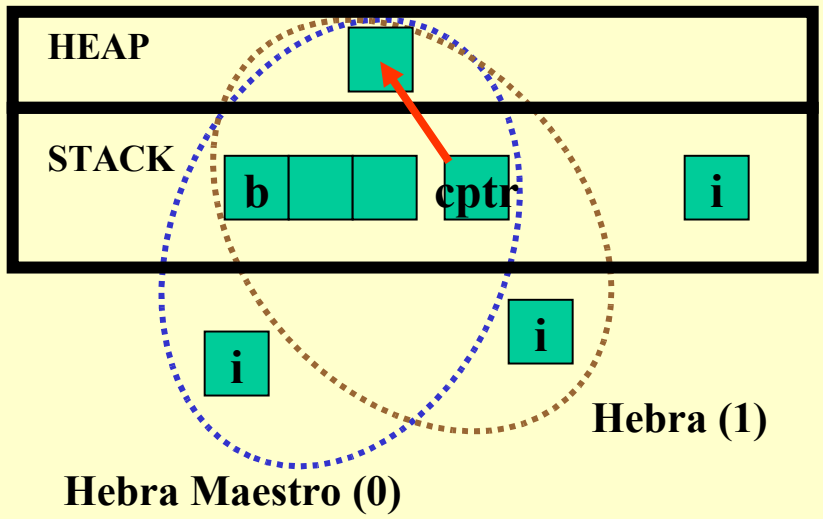
Paralelización de bucles(3)

- **Cada hebra tiene su propio contexto de ejecución**
  - Existen variables **compartidas** (misma dirección en el contexto de cada hebra) y variables **privadas**.
  - Por defecto, todas son ***COMPARTIDAS***, salvo el iterador del bucle.

```
#include <omp.h>

void main(void){
   int b[3];
   char* cptr;
   int i;
   cptr= malloc(sizeof(char));

#pragma omp parallel for
for( i= 0 ; i< 3 ; i++ )
            b[ i ]= i;
}
```



Número de hebras e identificación

- **Variables de entorno OMP_NUM_THREADS**
  - Número de hebras por defecto en secciones paralelas
- **void omp_set_num_threads(int num_hebras);**
  - Fija el número de hebras en secciones paralelas
- Para colocar el número de hebras igual al número de nodos del multiprocesador:

  int omp_get_num_procs(void);
  - Devuelve nº de procesadores físicos disponibles para el programa paralelo
- **void omp_get_thread_num(void);**
  - Devuelve número de hebra: 0,…,num_hebras-1

Gestión de variables privadas

- **Podemos especificar qué variables serán privadas y cuales compartidas**
  - Otra formulación del ejemplo anterior

```
#include <omp.h>

void main(void){
   int b[3], i;
   char* cptr;
   cptr= malloc(sizeof(char));

#pragma omp parallel for shared( b, cptr)
        for( i= 0 ; i< 3 ; i++ )
              b[ i ]= i;
}
```

Gestión de variables privadas (2)

- **Cláusula private:**

```
#pragma omp parallel for private( j )
    for( i= 0 ; i< m ; i++ )
        for( j= 0 ; j < n ; j++ )
                a [ i ] [ j ] = a[i][0] + a [ i ] [ j ] ;
```

  - Existe una copia de las variables i y j para cada hebra que ejecute el bloque precedente
  - El **valor de i j**:
    - Después del bucle es **indefinido** ya que no se conoce el valor a la salida de las variables privadas.
    - No se hereda el valor de las variables compartidas.

Gestión de variables privadas (3)

Asignación de valores coherentes a variables privadas

- **firstprivate**

```
x[0]= funcion_compleja();

#pragma omp parallel for firstprivate(x)
  for( i= 0 ; i< n ; i++ ) {
       x[ i]=  x[0] * x [i/2];
  }
```

- Las variables de la lista se inicializan de acuerdo con sus valores originales.

- **lastprivate**

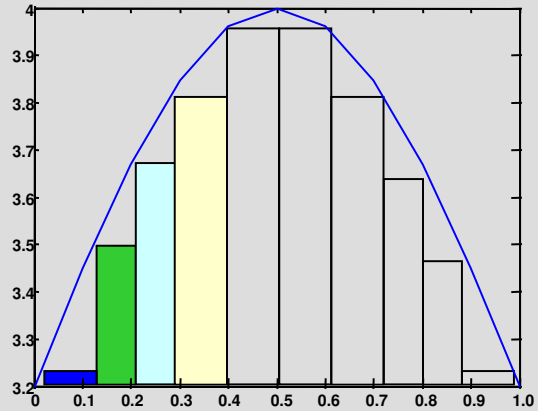
```
#pragma omp parallel for lastprivate(i)
  for( i= 2 ; i< n ; i++ ) {
         x[ i ]=  x[i-1] * x [i-2];
  }
  fibo_10 = x[i];
```

- El valor que se copia al objeto original es el valor en la última iteración del bucle.

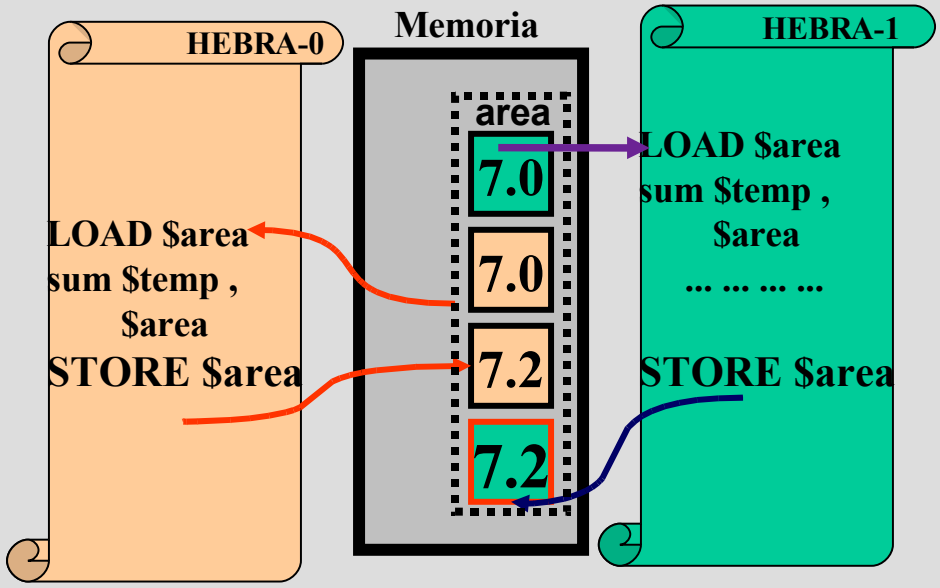
Exclusión mutua

Cálculo de pi: Condiciones de carrera

$$\pi = \int_0^1 \frac{4}{1+x^2}$$




```
   ...
double area, pi, x;
int i,n;
   ...
area= 0.0;
#pragma omp parallel for  private(x)
  for( i= 0 ; i< n ; i++ ) {
        x = (i+0.5)/n;
       area += 4.0/(1.0 + x*x);
  }
pi = area / n;
```



Exclusión mutua (2)

Definición de una sección crítica: ***#pragma omp critical [nombre]***

```
   ...
double area, pi, x;
int i,n;
   ...
area= 0.0;
#pragma omp parallel for private(x)
  for( i= 0 ; i< n ; i++ ) {
        x = (i+0.5)/n;
       #pragma omp critical
       area += 4.0/(1.0 + x*x);
  }
pi = area / n;
```

- **Ventajas:**
  - Secuencializan código (depuración).
  - Acceso seguro a memoria compartida.
- **Desventajas:**
  - Disminuyen eficiencia al reducir paralelismo.

• El ***nombre*** opcional de la sección crítica permite coexistir a regiones críticas diferentes. Los nombres actúan como identificadores globales. Todas las secciones críticas que no tienen nombre son tratadas como la misma.

Exclusión mutua(3)

Funciones para gestión de cerrojos

CERROJOS SIMPLES

- `void omp_init_lock( omp_lock_t * cerrojo );`
- `void omp_destroy_lock( omp_lock_t * cerrojo );`
- `void omp_set_lock( omp_lock_t * cerrojo );`
- `void omp_unset_lock( omp_lock_t * cerrojo );`
- `int omp_test_lock( omp_lock_t * cerrojo );`
- `int omp_get_nested( void );`

CERROJOS ANIDADOS

- `void omp_init_nest_lock( omp_nest_lock_t * cerrojo );`
- `void omp_destroy_nest_lock( omp_nest_lock_t * cerrojo );`
- `void omp_set_nest_lock( omp_nest_lock_t * cerrojo );`
- `void omp_unset_nest_lock( omp_nest_lock_t * cerrojo );`
- `int omp_test_nest_lock( omp_nest_lock_t * cerrojo );`

Exclusión mutua (4)

#pragma omp atomic

- Asegura que una posición específica de memoria debe ser modificada de forma atómica, sin permitir que múltiples threads intenten escribir en ella de forma simultánea.
- Proporciona una sección mini-critical.

- La sentencia debe tener una de las siguientes formas:
  - x <*operacion-binaria*> = <*expr*>
  - x++
  - ++x
  - x---
  - ---x

```
#include <omp.h>

int main(void)
{
int x= 0;
    #pragma omp parallel
      {#pragma omp atomic
            x= x+1;
      }
  return 0;
}
```

- Sólo atomiza la lectura y escritura de la variable. La evaluación de la expresión no es atómica ¿cuidado!

Reducciones en OpenMP

El ejemplo anterior se puede mejorar

```
   ...
double area, pi, x;
int i,n;
   ...
area= 0.0;

#pragma omp parallel for private(x)  reduction(+:area )
  for( i= 0 ; i< n ; i++ ) {
       x = (i+0.5)/n;
      area += 4.0/(1.0 + x*x);
  }
pi = area / n;
```

Paralelización condicional

Cláusula *if*

Si n no es suficientemente grande, los gastos de gestión de hebras pueden hacer que no exista ganancia paralela

```
            ...
#pragma omp parallel for private(x)  reduction(+:area )  if ( n > 5000 )
 for( i= 0 ; i< n ; i++ ) {
             x = (i+0.5)/n;
            area += 4.0/(1.0 + x*x);
  }
pi = area / n;
```

Planificación de bucles paralelos

- **¿Cómo se distribuyen las hebras el trabajo?**
  - Sea **n** el número de iteraciones totales.
  - Sea **t** el número de hebras.

- **Schedule (<tipo>[, <tamaño>])**
  - schedule(static), ⌈n/t⌉ iteraciones contiguas por hebra.
  - schedule(static, **K**), asignación de k iteraciones contiguas.
  - schedule(dynamic), una iteración cada vez.
  - schedule(dynamic, **K**), k iteraciones cada vez.
  - schedule(guided, **K**), descenso exponencial con k.
  - schedule(guided), descenso exponencial.
  - schedule(runtime), depende de la variable OMP_SHEDULE.
    - Ejemplo: setenv OMP_SCHEDULE “static,1”

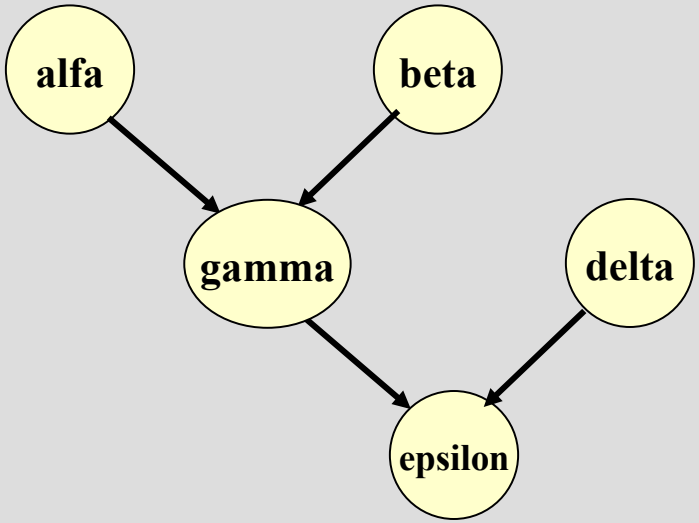
Planificación de bucles paralelos

Distribución de iteraciones de costo variable

```
            ...
#pragma omp parallel for private(j) schedule (dynamic,5)
  for( i= 0 ; i< 5000 ; i++ )
     for( j= 0 ; j< f(i); j++ )
            tarea_costo_variable(i,j);
```

Paralelismo funcional en OpenMP

Es posible asignar diferente código a a cada hebra




```
#include <omp.h>
 ...
#pragma omp parallel
{       #pragma omp sections
        { #pragma omp section
                v= alfa();
          #pragma omp section
                w= beta();
        }
       #pragma omp sections
        { #pragma omp section
                x= gamma(v,w);
         #pragma omp section
                y= delta();
        }
}
printf("Valor de epsilon= %6.6f\n", epsilon(x,y) );
```

En las regiones paralelas anidadas, se ejecuta sólo la primera hebra que llegue...

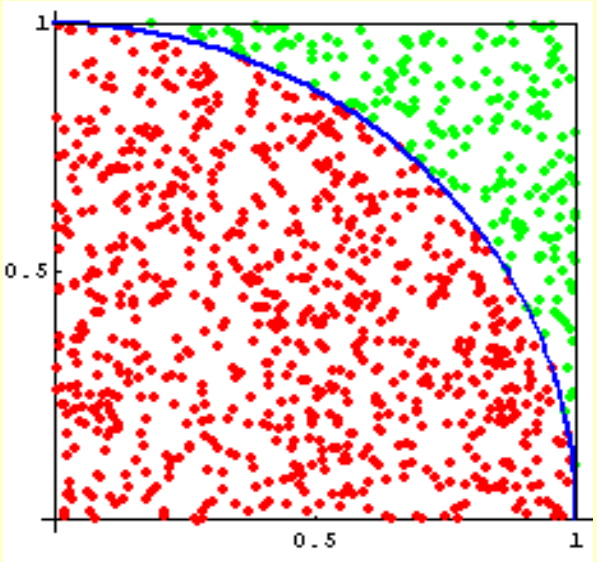
#pragma omp parallel

Cálculo MonteCarlo de π en OpenMP

```
...
#include "omp.h"
int main(int argc,char * argv[]) {
   int    n, i;
   double pi_est,x,y;
   int    in_circle = 0;
   int    local_in_circle,t;
   unsigned short xi[3];

   n=atoi(argv[1]);    omp_set_num_threads(atoi(argv[2]));

   #pragma omp parallel private(i,xi,t,x,y,local_in_circle)
   {
    local_in_circle=0;
    xi[0]=atoi(argv[3]); xi[1]=atoi(argv[4]); xi[2]=omp_get_thread_num();
    t=omp_get_num_threads();
    for (i = xi[2]; i < n; i+=t) { x = erand48(xi);   y = erand48(xi); if (x*x + y*y <= 1.0)
                                    local_in_circle++;}
    #pragma omp critical
      in_circle+=local_in_circle;
   }
   printf("Estimate of pi= %7.5f\n",4.0*((double) in_circle)/n);       return 0;}
```



Sincronización en OpenMP

#pragma omp flush [(<lista de variables>)]

- **Especifica un punto de sincronización donde se requiere que todas las hebras del grupo tengan una visión consistente de ciertos objetos en la memoria.**
- **En este punto se escriben en memoria variables que son visibles por las hebras.**
- **Si no se especifica lista de variables, se escriben en memoria todos los objetos compartidos por las hebras.**

```
/* ERROR - La directiva flush no puede ser
la sentencia justamente inmediata a una
sentencia if */
if (x!=0)
#pragma omp flush (x)
...
/* OK – La directiva flush es encerrada en
una instrucción */
if (x!=0) {
#pragma omp flush (x)
}
....
```

Sincronización en OpenMP

#pragma omp flush [(<lista de variables>)]

```
int main() {
int data, flag=0;
#pragma omp parallel sections num_threads(2)
 {
  #pragma omp section
   Lee(&data);
   #pragma omp flush(data)
   flag = 1;
   #pragma omp flush(flag)
  }
  #pragma omp section
  { while (!flag) {
     #pragma omp flush(flag)
        }
    #pragma omp flush(data)
     Procesa(&data);printf_s("%d\n", data);
      }
   }
}
```

```
void Lee(int *data)
{
   printf_s("read data\n");
   *data = 1;
}

void Procesa(int *data)
{
  printf_s("process data\n");
  (*data)++;
}
```

Sincronización en OpenMP

#pragma omp flush [(<lista de variables>)]

La directiva *flush* esta implícita en

barrier		
	A la entrada	A la salida
critical	*	*
ordered	*	*
parallel	*	*
for		*
sections		*
single		*
parallel for	*	*
parallel sections	*	*

No esta implícita si esta presente la cláusula nowait y en:

	A la entrada	A la salida
for	*	
master	*	*
sections	*	
single	*	

Sincronización en OpenMP

#pragma omp barrier

- Sincroniza todos los threads del equipo. Cuando un thread alcanza una directiva barrier, esperará en ese punto hasta que todos los threads del equipo lleguen hasta esta directiva.

- Cuando todos los threads han alcanzado dicho punto, continúan con la ejecución en paralelo del código que sigue a la barrera.

- Claúsula **nowait**: Las hebras de un bucle paralelo no se sincronizan al final del mismo y pueden continuar con la siguiente sentencia después del bucle sin esperar a que el resto finalice.

```
#pragma omp parallel shared (A, B, C) private(id)

        {id=omp_get_thread_num();
          A[id] = big_calc1(id);

      #pragma omp barrier

      #pragma omp for

         for(i=0;i<N;i++)

              {C[i]=big_calc3(I,A);}

      #pragma omp for nowait

          for(i=0;i<N;i++){ B[i]=big_calc2(C,
          i); }
          A[id] = big_calc3(id);

      }
```

Sincronización en OpenMP

#pragma omp threadprivate

- Se utiliza para permitir que varibles de ámbito global se conviertan en locales y persistentes a una hebra de ejecución a través de múltiples regiones paralelas.

```
#include <omp.h>
int  a, b, i, tid;  float x;

#pragma omp threadprivate(a, x)

main ()  {
/* Desactivo ajuste dinámico número de hebras*/
  omp_set_dynamic(0);

#pragma omp parallel private(b,tid)
  { tid = omp_get_thread_num();
    a = tid; b = tid;   x = 1.1 * tid +1.0;
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }

 #pragma omp parallel private(tid)
  {tid = omp_get_thread_num();
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }
}
```

Sincronización en OpenMP

#pragma omp ordered y claúsula ordered

- El código afectado se ejecuta en el orden en que las iteraciones hubieran sido ejecutadas en una ejecución secuencial del bucle.

- Puede aparecer sólo una vez en el contexto de una directiva for o parallel for.

- Sólo puede estar una hebra ejecutándose simultáneamente

```
static float a[1000], b[1000], c[1000];
void test(int first, int last)
{
  #pragma omp for schedule(static) ordered
   for (int i = first; i <= last; ++i) {
      // Cálculo
      if (i % 2)
      { #pragma omp ordered
          printf_s("test() iteration %d\n", i);} }
}
void test2(int iter)
{
  #pragma omp ordered
     printf_s("test2() iteration %d\n", iter);}
int main( )
{   int i;
   #pragma omp parallel
    {  test(1, 8);
       #pragma omp for ordered
        for (i = 0 ; i < 5 ; i++)
          test2(i);
    }
}
```

Sincronización en OpenMP

#pragma omp master

- El código afectado se ejecuta ejecutado sólo por hebra maestro del equipo.
- El resto de threads del equipo se saltan esta sección del código.

```
#include <omp.h>
#include <stdio.h>
int main( )
{
   int a[5], i;
   #pragma omp parallel
   {
       #pragma omp for
      for (i = 0; i < 5; i++)
         a[i] = i * i;
      // Maestro imprime resultados intermedios
      #pragma omp master
         for (i = 0; i < 5; i++)
            printf_s("a[%d] = %d\n", i, a[i]);
      #pragma omp barrier
      #pragma omp for
      for (i = 0; i < 5; i++)
         a[i] += i;
   }
}
```

Producto escalar de dos vectores

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define VECLEN 100

float a[VECLEN], b[VECLEN], sum;

float dotprod ()
{
int i,tid;

tid = omp_get_thread_num();
#pragma omp for reduction (+:sum)
  for (i=0; i < VECLEN; i++)
    sum = sum + (a[i]*b[i]);
}
```

```
int main (int argc, char *argv[]) {
int i;

for (i=0; i < VECLEN; i++)
  a[i] = b[i] = 1.0 * i;
sum = 0.0;

#pragma omp parallel
  dotprod();

printf("Sum = %f\n",sum);
}
```

Búsqueda en un array

```
#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)
    }
```

Bibliografía y enlaces

- **P*arallel Programming in C with MPI and OpenMP*. Michael J. Quinn. Mc Graw-Hill. Capítulo 17.**
- ***Parallel Programming in OpenMP*. Rohit Chandra, Dave Kohr, Leonardo Dagum, Ramesh Menon, Dror Maydan, Jeff McDonald. Morgan Kaufmann**
- ***Repositorio de código OpenMP***

  http://sourceforge.net/projects/ompscr/

- ***Página de intel* para bajarse compilador para Linux**

  http://www.intel.com/cd/software/products/asmo-na/eng/219771.htm
- **www.openmp.org**