SEMINARIO EARLY ADOPTERS (OpenMP)

1. Introducción a OpenMP

OpenMP 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** es decir, el hilo principal crea varios hilos para que realicen una tarea de forma paralela, y él se queda esperando que todos los hilos creados terminen antes de continuar. Está disponible en muchas arquietecturas 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.

2. Directiva `parallel`

Una región paralela es un bloque de código que será ejecutado por múltiples threads. Esto es definido en OpenMP mediante la directiva `parallel`. Cuando un thread ejecuta una región paralela, se crea un conjunto de threads para llevar a cabo las tareas definidas dentro de la región. El final de la región paralela determina un punto de sincronización a partir del cual sólo continuará ejecutándose el thread principal (id=0).

La estructura general de un programa que incluya una región paralela será la siguiente:

```
#include <omp.h>

main(){
int var1, var2, var3;

//código secuencial
…
// Comienzo de la sección paralela. Comienza la ejecución de un
//  conjunto de threads. Se especifica el alcance de las
//  variables

#pragma omp parallel private(var1,var2) shared (var3)
{
   //código paralelo ejecutado por todos los threads

   // todos los theads terminan y a partir de aquí sólo continua
el thread
   //principal
}
// código secuencial
…
}
```

3. Compilación con OpenMP

Para generar código paralelo hay que incluir la librería omp.h en los ficheros fuente, introducir en el código directivas de paralelismo y usar la opción de compilación

```
-fopenmp
```

Para compilar, por tanto, usamos la siguiente línea de comandos:

```
>c++ -O3 fichero.cpp -fopenmp
```

Habitualmente, los programas generados con OpenMP se ejecutan igual que los secuenciales pero utilizando tantos hilos como núcleos haya en el sistema para ejecutar las regiones paralelas. Opcionalmente, antes de ejecutar un programa, se puede indicar el número de threads que queramos que se utilicen. Para hacerlo se usa la variable de entorno `OMP_NUM_THREADS` del siguiente modo:

```
>export OMP_NUM_THREADS=2
```

De ese modo se fuerza a openmp a usar dos threads para ejecutar las regiones paralelas del programa. El valor por defecto de la variable anterior es el número de cores del sistema. También se puede establecer el número de threads para ejecutar el programa desde el propio código fuente usando la función `omp_set_num_threads(n)`, donde `n` es el número de threads.

A continuación, veamos un ejemplo con una región paralela en la que cada hilo mostrará el mensaje "Hola mundo" seguido de su identificador, y el hilo con identificador cero debe mostrar cuántos hilos se han creado para ejecutar la región paralela. Para ello vamos a utilizar dos funciones de openMP denominadas `omp_get_num_threads()` y `omp_get_thread_num()`. La primera devuelve el número de hilos lanzados en la región y la segunda el identificador del hilo que la ejecuta.

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

main(){
  int nthreads, tid;
  #pragma omp parallel private(tid)
  {
    tid=omp_get_thread_num();
    printf("Hola mundo desde el thread %d\n",tid);
    if (tid==0){
      nthreads=omp_get_num_threads();
      printf("Número de threads %d\n",nthreads);
    }
  }
}
```

4. Constructor de trabajo compartido: directiva `for`

La directiva `for` especifica que las iteraciones del primer bucle situado dentro de la región paralela serán ejecutadas en paralelo por el conjunto de threads. La variable índice del primer for debe ser de tipo entero.

```
#include <iostream>
#include <sys/time.h>
#include <omp.h>

#define DEBUG
using namespace std;

void generar(double *m,int n) {
  int i,j;

  for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
      m[i*n+j]=(double) (2*i+j);
    }
  }
}

void escribir(double *m,int n) {
  int i,j;

  for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
      cout << m[i*n+j]<< " ";
    }
    cout << endl;
  }
  cout << endl;
}

void mm(int t,double *a,double *b,double *c) {
  int i, j, k;
  double s;

  // Inicio multiplicación por filas
  #pragma omp parallel for private(i,j,k,s)
  for (i = 0; i < t; i++) {
    for(j=0;j<t;j++) {
      s=0.;
      for (k = 0; k < t; k++) {
        s = s+a[i*t+k]*b[k*t+j];
      }
      c[i*t+j]=s;
    }
  }
  // Fin multiplicación por filas
}
```

```
main()
{
  double *a,*b,*c;
  int n;
  struct timeval ti, tf;
  double tiempo;

  cin >> n;

  a=new double[n*n];
  b=new double[n*n];
  c=new double[n*n];

  generar(a,n);
  generar(b,n);

#ifdef DEBUG
  escribir(a,n);
  escribir(b,n);
#endif
  gettimeofday(&ti, NULL);   // Instante inicial

  mm(n,a,b,c);

  gettimeofday(&tf, NULL);   // Instante final
  tiempo=  (tf.tv_sec  -  ti.tv_sec)*1000000  +  (tf.tv_usec  -
ti.tv_usec);
  cout << "  En " << tiempo << " microsegundos" << endl;
#ifdef DEBUG
  escribir(c,n);
#endif
  delete [] a;
  delete [] b;
  delete [] c;
}
```

alternativamente podemos hacer la multiplicación por columnas usando el siguiente código como región paralela en lugar del incluido en el listado anterior.

```
// Inicio multiplicación por columnas
#pragma omp parallel for private(i,j,k,s)
for (i = 0; i < t; i++) {
    for(j=0;j<t;j++) {
      s=0.;
      for (k = 0; k < t; k++) {
         s = s+a[j*t+k]*b[k*t+i];
      }
      c[j*t+i]=s;
    }
  }
}
// Fin multiplicación por columnas
```

Ejercicio : Ejecutar el ejemplo anterior variando el número de threads (entre 1, versión secuencial, y el número de cores del sistema), y el tamaño de la matriz (con matrices pequeñas y grandes), y sacar conclusiones sobre la reducción en el tiempo de ejecución al usar OpenMP.

5. Constructor de trabajo compartido: directiva `sections`

La directiva `sections` es un constructor de trabajo compartido no iterativo. Sirve para especificar, dentro de una región paralela, secciones de código que serán ejecutadas por distintos threads. Cada sección individual será ejecutada por un solo thread.

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

#define N 1000

main(){
  int i;
  float a[N],b[N],c[N],d[N];

  // inicializaciones

  for(i=0; i<N; i++) {
     a[i]=i*1.5;
     b[i]=i+22.35;
  }

  #pragma omp parallel shared(a,b,c,d) private(i)
  {
    #pragma omp sections
    {
      #pragma omp section
      for(i=0;i<N;i++) {
        c[i]=a[i]+b[i];
        printf("Cálculo de la primera sección ejecutado por hilo
%d\n", omp_get_thread_num());
      }

      #pragma omp section
      for(i=0;i<N;i++) {
        d[i]=a[i]+b[i];
        printf("Cálculo de la primera sección ejecutado por hilo
%d\n", omp_get_thread_num());
      }
    }
  }
}
```

6. Constructor de sincronización: Directiva `critical`

La directiva `critical` especifica una región de código que debe ser ejecutado por sólo un thread en el mismo instante de tiempo. Si un thread está ejecutando la sección crítica y otro thread intenta ejecutarla al mismo tiempo, quedará bloqueado hasta que el primero finaliza la sección crítica. Se puede especificar un nombre de sección crítica para poder disponer de secciones distintas.

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

main(){
  int x=0;

  #pragma omp parallel shared(x)
  {

    #pragma omp critical
      x=x+1;
  }
}
```

Ejercicio : Diseñar un programa que realice 2000 incrementos de una variable compartida por dos threads donde cada uno realiza el incremento de 1000 unidades.