PRÁCTICA 3

Entrenamiento con ARM: OpenMP y OpenCV

David Martínez García

Alejandro Gea Belda

Parte 1 - Paralelismo en sistemas de memoria centralizada: Introducción a OpenMP

Tarea 1.1: Estudio del API OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación paralela en diferentes plataformas. La API define únicamente el estándar que hay que seguir para poder utilizar la interfaz. OpenMP está disponible en múltiples plataformas como Microsoft, Windows o Linux. En nuestro caso, su implementación está integrada en cualquier compilador de C y C + + de GNU con una versión 4.2 o superior.

Tarea 1.2: Estudio de OpenMP

1. Explique qué hace el código que se añade a continuación.

El código dado hace un par de operaciones sencillas. En primer lugar define tres vectores de tamaño 100 y asigna a los vectores a y b un valor a cada posición igual al índice del vector. Posteriormente, el programa suma en un bucle los vectores a y b, guardando el resultado en el vector c e imprimiendo por pantalla el resultado obtenido.

```
float a[N], b[N], c[N];

for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
```

```
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
```

En esencia, el programa hace estas operaciones básicas. Sin embargo, se utiliza OpenMP para que la ejecución del código sea simultánea en varios hilos de nuestro equipo.

2. Delimite las distintas regiones OpenMP en las que se está expresando paralelismo.

El uso de OpenMP modifica el código utilizando diferentes instrucciones que se identifican con el tipo #pragma que permiten modificar como se ejecuta nuestro programa. Una vez modificado el código, hay que asegurarse de que se compila añadiendo la opción -fopenmp para que el programa se ejecute correctamente.

La primera instrucción que se utiliza es `#pragma omp parallel shared (a ,b ,c, nthreads, chunk) private (i, tid)` y realiza varias instrucciones. La parte de `#pragma omp parallel` especifica que todo el código que hay dentro de la instrucción se puede ejecutar de forma paralela. La cláusula '**shared**' se utiliza para especificar qué variables van a ser compartidas por los diferentes hilos mientras que la cláusula '**private**' se utiliza para definir qué variables serán privadas en cada hilo. Esto implica que cada hilo

tendrá su propia copia de las variables definidas como private mientras que las variables definidas en shared serán comunes para todos los hilos..

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    ...
    ...
    ...
}
```

La siguiente instrucción `omp_get_thread_num()` devuelve el identificador del hilo que invoca la función. Si ese hilo es el número 0, el programa devuelve el número de hilos activos mediante la función `omp_get_num_threads()`.

La segunda instrucción `#pragma omp for schedule(dynamic, chunk)` realiza varias instrucciones. La parte `#pragma omp for` permite que un bucle for se pueda ejecutar mediante varios hilos en paralelo. El fragmento `schedule(dynamic, chunk)` es una opción que se usa para controlar cómo se dividen las iteraciones del bucle entre los hilos. En este caso, la opción '**dynamic**' especifica que las iteraciones se asignan de manera dinámica a los hilos, es decir, que las iteraciones se irán asignando automáticamente a medida que estén disponibles para su uso. El parámetro '**chunk**' especifica cuántas iteraciones se asignan a la vez a cada hilo que, en nuestro caso, son 10 iteraciones por hilo.

```
#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
```

3. Explique con detalle qué hace el modificador de OpenMP schedule. ¿Cuáles pueden ser sus argumentos? ¿Qué función tiene la variable chunk en el código? ¿A qué afecta?

El modificador schedule en OpenMP se utiliza para controlar cómo se distribuyen las iteraciones de un bucle entre los hilos en un equipo paralelo. Esta permite controlar la asignación de trabajo a los hilos y, en consecuencia, la forma en que se aprovechan los recursos del equipo. El modificador schedule tiene varios argumentos que te permiten especificar cómo se asignan las iteraciones del bucle a los hilos:

- static: Con `schedule(static, chunk)`, se divide el bucle en bloques de un tamaño fijo llamado "chunk" y se asigna a los hilos en un patrón estático y equitativo. Esto significa que las iteraciones se distribuyen de manera uniforme entre los hilos al principio, y cada hilo obtiene un conjunto de iteraciones contiguas.

- dynamic: Con `schedule(dynamic, chunk)`, las iteraciones se asignan de manera dinámica a los hilos a medida que los hilos terminan su trabajo. Cada hilo obtiene un

conjunto de iteraciones, y cuando un hilo termina su conjunto, recibe otro conjunto de iteraciones disponibles.

- guided: Con `schedule(guided, chunk)`, se asignan inicialmente bloques grandes de iteraciones a los hilos, pero a medida que los hilos completan su trabajo, reciben bloques más pequeños.

- auto: Con `schedule(auto)`, OpenMP permite al compilador determinar la estrategia de planificación más adecuada.

- runtime: Con `schedule(runtime, chunk)`, implica que la política de planificación y el tamaño de "chunk" se determinarán en tiempo de ejecución, generalmente mediante la configuración de variables de entorno o directivas de control específicas proporcionadas por la implementación de OpenMP.

4. ¿Qué función tiene el modificador de OpenMP dynamic en el código?

Como se ha explicado anteriormente, al utilizar el modificador '**dynamic**' en nuestro código hace que el bucle for que se encarga de sumar los vectores a y b se ejecute en hilos diferentes. Estos hilos van recibiendo conjuntos de instrucciones conforme resuelven las instrucciones recibidas anteriormente sin esperar a que todos los hilos terminen de ejecutar sus órdenes.

```
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
```

```
Thread 1 starting...
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
```

5. Investigue qué pasa si no declara como privadas las variables i y tid

Si no se declaran las variables i y tid como privadas en la región paralela, OpenMP tratará de manera predeterminada a estas variables como compartidas entre todos los hilos, lo que puede llevar a resultados incorrectos o comportamiento inesperado.

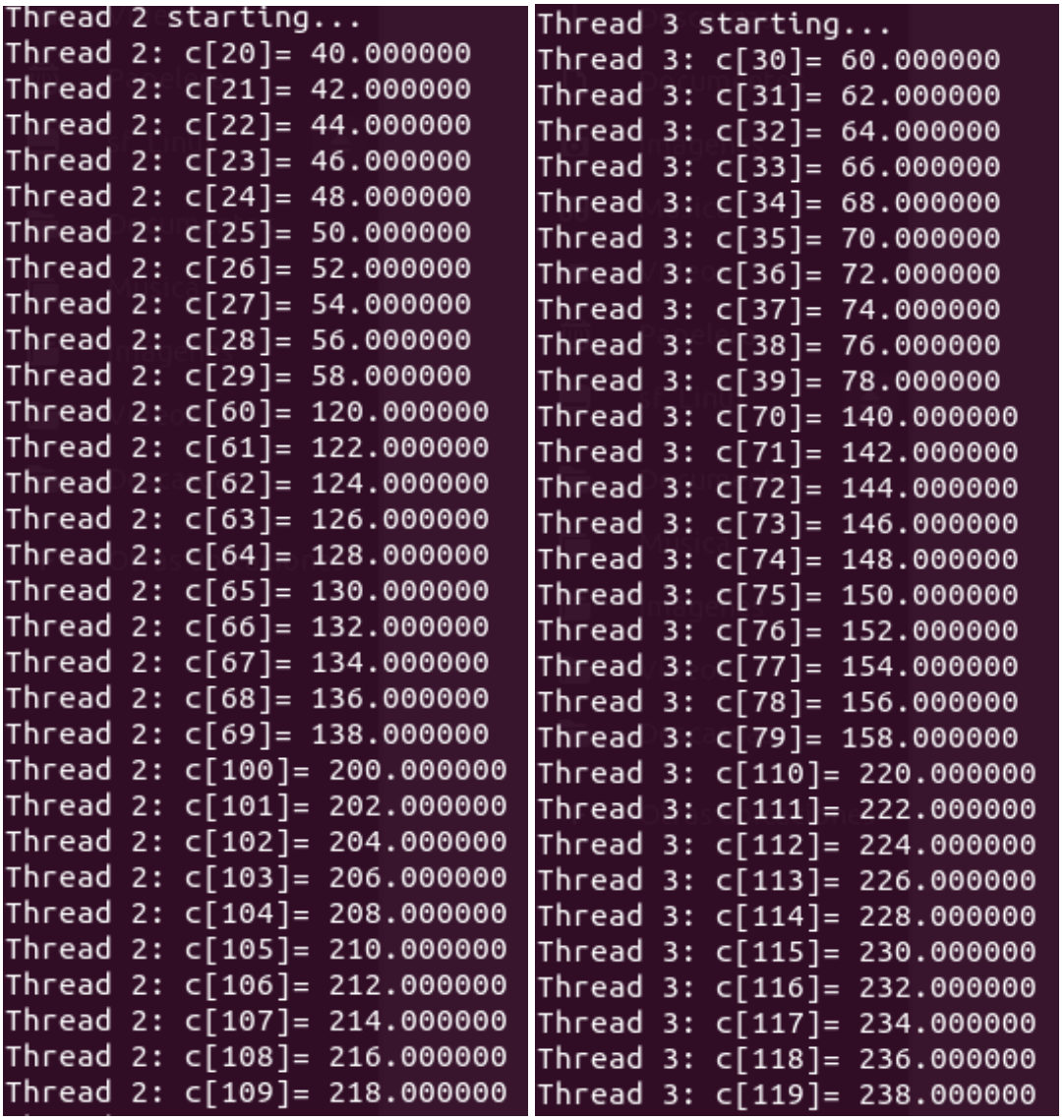
Dentro de una región paralela, OpenMP asume que las variables que no se declaran como privadas son compartidas, lo que significa que todos los hilos tendrán acceso y pueden modificar las mismas variables. Esto puede conducir a problemas de concurrencia, ya que múltiples hilos podrían estar intentando acceder o modificar las mismas variables i y tid al mismo tiempo.

Tarea J1. [Reto JEDI borde exterior]

Para emplear un planificador estático con OpenMP hay que cambiar el parámetro de nuestro pragma vector `#pragma omp for schedule(dynamic, chunk)` por el parámetro '**static**'. Para mejor visualización del código, se ha cambiado el tamaño de los vectores a 120 para que el reparto de la ejecución de nuestro código sea equitativo para los 4 procesadores que se están usando.

```
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
Thread 0: c[40]= 80.000000
Thread 0: c[41]= 82.000000
Thread 0: c[42]= 84.000000
Thread 0: c[43]= 86.000000
Thread 0: c[44]= 88.000000
Thread 0: c[45]= 90.000000
Thread 0: c[46]= 92.000000
Thread 0: c[47]= 94.000000
Thread 0: c[48]= 96.000000
Thread 0: c[49]= 98.000000
Thread 0: c[80]= 160.000000
Thread 0: c[81]= 162.000000
Thread 0: c[82]= 164.000000
Thread 0: c[83]= 166.000000
Thread 0: c[84]= 168.000000
Thread 0: c[85]= 170.000000
Thread 0: c[86]= 172.000000
Thread 0: c[87]= 174.000000
Thread 0: c[88]= 176.000000
Thread 0: c[89]= 178.000000
```

```
Thread 1 starting...
Thread 1: c[10]= 20.000000
Thread 1: c[11]= 22.000000
Thread 1: c[12]= 24.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 1: c[15]= 30.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 1: c[19]= 38.000000
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
Thread 1: c[90]= 180.000000
Thread 1: c[91]= 182.000000
Thread 1: c[92]= 184.000000
Thread 1: c[93]= 186.000000
Thread 1: c[94]= 188.000000
Thread 1: c[95]= 190.000000
Thread 1: c[96]= 192.000000
Thread 1: c[97]= 194.000000
Thread 1: c[98]= 196.000000
Thread 1: c[99]= 198.000000
```



En las imágenes se puede observar como, al utilizar el modificador static, los 4 hilos que están trabajando reciben 10 instrucciones cada uno y hasta que no terminan todos no vuelven a recibir instrucciones. Gracias a este modo de trabajar, podemos saber de antemano qué instrucciones exactas va a recibir cada hilo de nuestro procesador antes de ejecutar el código.

Tarea 1.3: Paralelización de una aplicación con OpenMP

Apartado 0:

Se ha modificado el programa pedido para que muestre por pantalla dos columnas de datos, en una se mostrará el tamaño de las matrices con las que se realizan las operaciones y en la otra se muestra el tiempo de ejecución del programa. Después se ha utilizado gnuplot para representar la gráfica resultante.

```
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ gcc -fopenmp -o matri
x matrix.c
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ matrix > matrix.steps
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ gnuplot matrix.gpi
```

Las gráficas obtenidas para las diferentes arquitecturas son las siguientes: