Práctica/laboratorio 3

Entrenamiento con ARM: OpenMP y OpenCV

Objetivos

Esta práctica persigue los siguientes objetivos:

- Aprender a **paralelizar** una aplicación sencilla con **OpenMP** en una máquina paralela de memoria centralizada mediante hilos/hebras (*threads*) usando el estilo de *variables compartidas*.
- Estudiar la API de OpenMP y aplicar distintas estrategias de paralelismo en su aplicación.
- Aprender a configurar, mantener y explorar una tarjeta de entrenamiento basada en Linux.
- De forma transversal se aprenderá el paquete matemático Numpy/SciPy/Matplotlib y OpenMP.

Parte 1 [6 puntos]

Paralelismo en sistemas de memoria centralizada: Introducción a OpenMP

Previo: El profesor de teoría/prácticas realizará una breve introducción a OpenMP. Pueden encontrar en el Campus Virtual el trabajo monográfico de OpenMP que presentaron sus compañer@s en cursos anteriores.

Tarea 1.1: Estudio del API OpenMP

Se deberá estudiar el API de OpenMP y su uso con GNU GCC (gcc, g++), comprobando el correcto funcionamiento de algunos de los ejemplos que hay disponibles en Internet (p.ej. ejemplo https://lsi.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php)

Tarea 1.2 Estudio de OpenMP

Dado el siguiente código paralelizado con OpenMP se pide:

1. Explique qué hace el código que se añade a continuación.
2. Delimite las distintas **regiones OpenMP** en las que se está expresando paralelismo.
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?
4. ¿Qué función tiene el modificador de OpenMP *`dynamic`* en el código?
5. Investigue qué pasa si no declara como privadas las variables `i` y `tid`.

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

#define CHUNKSIZE       10
#define N               100

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

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

    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
```

```
        }
        printf("Thread %d starting...\n",tid);

        #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]);
        }

    }
}
```

Tarea J1 [Reto JEDI borde exterior (+5 puntos)]: Repita la tarea 1.2 empleando un planificador estático y no dinámico. Puede hacerlo "a mano" o usando algún modificador de OpenMP.

Tarea 1.3 Paralelización de una aplicación con OpenMP

Diseñe un programa en C/C++ para multiplicar dos matrices cuadradas con elementos de tipo `double` (punto flotante de 64 bits) entre **0 y 1**. Después multiplique la matriz resultante por otra matriz de enteros entre 0 y 255 **elemento a elemento**. Es decir, diseñe un programa que haga esto:

$C = A \times B;$ Con A y B 2 matrices cuadradas de coeficientes `double` entre 0 y 1.
$R = K \cdot C;$ Con K una matriz aleatoria con coeficientes entre 0 y 255.

En la matriz resultante R, los coeficientes se calculan así:
$R_{ij} = K_{ij} \cdot C_{ij}$, para todas las $\forall\, i, j$.

Bien, ahora necesitamos que el programa **tarde** un tiempo **no despreciable** para que tenga sentido acometer un proceso de paralelización con OpenMP. Despreciable, en este contexto, significa que el tiempo de cómputo/cálculo útil del programa será **muy superior** al tiempo que emplee OpenMP cuando introduzca retardos debidos a: creación/destrucción/sincronización de hebras, interacción con el sistema operativo, cambios en el propio programa para acelerar la ejecución, etc. Como hemos visto en clase, si un programa tarda muy poco en ejecutarse quizá no tenga sentido el paralelizarlo.

0. Para lograr que el programa sea "más pesado" haga las siguientes pruebas. Muestre en una serie de gráficas cómo varía el tiempo paralelo de ejecución o $t_p(parámetro)$ en función de algún $parámetro$ que modifique la carga computacional del problema. Por ejemplo, pruebe a cambiar el **tamaño de la matriz**, el tipo de datos, etc. (ojo: si la matriz supera cierto tamaño debe declararse de forma dinámica, para que se almacene en el heap, o no cabrá en la sección .data).
   Por ejemplo: ¿Varía el tiempo si cambiamos el tipo de los elementos de las matrices de `double` a `float`, o a `int`/`char`? ¿Varían algo estar gráficas si ejecuta los programas en Intel/AMD (ordenadores del laboratorio) o ARM (RaspberryPi)?

Una vez realizado lo anterior, realice las siguientes tareas:

1. Paralelice el código con OpenMP y tome los siguientes tiempos de ejecución: código secuencial original (sin paralelizar), código paralelo ejecutado en mono-hebra y código con dos o más hebras. Apunte las ganancias en velocidad que se observan. Recuerde que la ganancia en velocidad o *speedup* es el cociente entre el tiempo en ejecución secuencial (monohebra) y el tiempo paralelo.

$$S_p(p) = \frac{t_{secuencial}}{t_{paralelo}(p)}$$

2. Represente en varias gráficas el tiempo paralelo, la ganancia en velocidad y la eficiencia de su código en función del número de hebras. ¿En cuántas hebras es más eficiente el código?

$$\eta(p) = \frac{S_p(p)}{p}$$

3. Compare los resultados obtenidos en el ARM de la Raspberry vs AMD/Intel del laboratorio. ¿Observa diferencias? ¿Qué puede concluir?

Tarea J2 [Reto JEDI borde exterior (+5 puntos)]: Si la matriz B es transpuesta, el código que produce el compilador para multiplicar las dos matrices A y B producirá menos fallos de caché, dado que maximizamos la localidad espacial de los datos en la cachés. Intente demostrar y medir si este hecho afecta a su problema. Nota: puede que tenga que escalar el problema para que el efecto sea apreciable.

Parte 2 [4 puntos]

Preparación y estudio del entorno de trabajo

En esta primera parte se familiarizará con la tarjeta RaspberryPi 3 Model B o Model B+ y aprenderá a instalar y configurar todo lo que necesite para realizar la práctica en el laboratorio o en casa.

Tarea 2.1: Acceso a Internet

Antes que nada, es siempre interesante y deseable tener acceso a Internet para mantener un sistema Linux (Raspbian en este caso) actualizado.

1.1 Identifique y muestre algunas características de las interfaces de red de las que dispone la tarjeta (apunte el tipo de interfaz, nombre en el sistema, MAC, ...)

Puede acceder a esta información de varias maneras:

`$ ls /sys/class/net` (mire qué información le muestra Linux)

```
$ cat /sys/class/net/<interfaz>/address
$ ifconfig  <interfaz>
```

Responda a las siguientes preguntas:

1. ¿Qué es la **MAC**? ¿Para qué sirve?
2. ¿Qué es y para qué sirve el parámetro **MTU** de cada interfaz de red? ¿Se puede cambiar? ¿Cómo nos afectaría?

Tarea 2.2: Exploración del sistema Linux (semiguiada)

El profesor de prácticas realizará un tutorial guiado donde explicará y/o esbozará todas las tareas que deberá acometer:

2.1 Explore el directorio `/proc` y responda a las siguientes cuestiones:

Nota: En muchos de estos casos, además de por inspección del directorio `/proc`, existen órdenes que muestran la misma información que la que pedimos en estos apartados. Si ése es el caso, muéstrelas. Por ejemplo, el tiempo que lleva encendida la máquina se puede saber con la orden `uptime` y leyendo un cierto archivo en `/proc`.

a) ¿Qué es el propio directorio `/proc`? ¿Qué información nos da? ¿Quién nos la da?
b) Características **completas** del microprocesador del que dispone según la información que encuentre en `/proc`. **Elabore además una lista con todos los conjuntos de extensiones al repertorio ISA base de su microprocesador y encuentre qué tipo de instrucciones aportan** (defina qué hacen estos conjuntos de instrucciones). ¿Dispone de extensiones vectoriales? ¿Cuáles? ¿De cuántos *cores* dispone?
c) ¿Puede realizar su procesador la función de un CRC de 32 bits (código de redundancia cíclico) por hardware? ¿Puede encontrar algún ejemplo de, en su caso, dicha instrucción o instrucciones?
d) Tiempo exacto que lleva encendida la tarjeta/equipo.
e) Versión de Linux y versión del compilador con la que fue compilado.
f) ¿Qué sistemas de archivos (*filesystems*) se soportan? Comente las características de, al menos, 5 de ellos.
g) ¿Qué muestra el fichero `/proc/stat` ? ¿Hay alguna orden que de la misma información pero de forma más legible?
h) ¿Qué muestra el fichero `/proc/cmdline` ?
i) ¿Qué muestra el fichero */proc/softirqs ?*
j) ¿Qué hay en los directorios dentro `/proc` que empiezan con un número? Qué información se muestra? Ponga algún ejemplo representativo.
k) ¿Cuántos sistemas de archivos de tipo ext3 o ext4 están montados? ¿Dónde? ¿Con qué archivo u orden puede acceder a esta información?
l) ¿Cuánta memoria tiene libre? Compruebe que encuentra la misma información que la orden `free`

m) ¿Cuáles son los módulos que tiene instalado el kernel? ¿Qué orden (comando) puede acceder a esta misma información de forma más legible?
n) ¿Qué módulos usan el módulo de *bluetooth*?

Entrega de la práctica:

La práctica tendrá una duración de **3 sesiones**.