PRÁCTICA 3: Parte individual
Estudio y compresión de OpenMP

INGENIERÍA DE LOS COMPUTADORES

UNIVERSIDAD DE ALICANTE

CURSO 2020/21

David Molina Martínez

Tarea 0:

La tarea 0 consiste en una preparación previa para la realización de la tarea 1, en este caso lo que se busca es que mediante la compilación y ejecución del código dado, se entienda lo que hacen sus distintas partes:

```
#include

#define N 1000

#define CHUNKSIZE 100

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

        int i, chunk;

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

        /* Inicializamos los vectores */

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

                a[i] = b[i] = i * 1.0;

        chunk = CHUNKSIZE;

        #pragma omp parallel shared(a,b,c,chunk) private(i) {

                #pragma omp for schedule(dynamic,chunk) nowait

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

                        c[i] = a[i] + b[i];

}

/* end of parallel region */ }
```

1.1 ¿Para qué sirve la variable chunk?

Define el tamaño mínimo del bloque utilizado.

1.2 Explique completamente el pragma:

Pragma es como tal lo que se tiene que poner para todos los directivos C++.

#pragma omp parallel shared(a,b,c,chunk) private(i)

Parallel crea un equipo de hilos y se convierte en el maestro del equipo. El maestro es un miembro del equipo y tiene el numero de hilo 0 del equipo. Se podría decir que es el capitán.

▪ ¿Por qué y para qué se usa shared(a,b,c,chunk) en este programa?

Declara las variables en una lista para cada uno de los hilos y estas se comparten en todos los hilos.

▪ ¿Por qué la variable i está etiquetada como private en el pragma?

Private lo que hace es crear una lista de variables privadas para cada hilo, por lo que en este caso está creando una variable privada para cada uno de los hilos.

1.3 ¿Para qué sirve schedule? ¿Qué otras posibilidades hay?

Describe cómo se dividen las iteraciones del bucle entre los subprocesos del equipo. La programación predeterminada depende de la implementación. Otras posibilidades son NO WAIT/ nowait que si se especifica hace que los hilos no se sincronicen al final de cada bucle paralelo, ORDERED que define un orden entre las iteraciones del bucle dando como resultado algo parecido a un programa lineal y COLLAPSE que especifica cuantos hilos s epueden procesar hasta realizar un colapso a mayor escala.

1.4 ¿Qué tiempos y otras medidas de rendimiento podemos medir en secciones de código paralelizadas con OpenMP?

Memoria usada, el tiempo de resolución de los hilos, de los candados simples y anidados, cantidad de threads, variaciones de coma flotante.

Tarea 1:

¿Qué es 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. Está disponible en muchas arquitecturas, 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.

Definido conjuntamente por proveedores de hardware y de software, OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, para plataformas que van desde las computadoras de escritorio hasta supercomputadoras. Una aplicación construida con un modelo de programación paralela híbrido se puede ejecutar en un cluster de computadoras utilizando OpenMP y MPI, o a través de las extensiones de OpenMP para los sistemas de memoria distribuida.

En OpenMP tenemos una sintaxis específica que determina el funcionamiento del código en el que la utilicemos:

```
# pragma omp <directiva> [cláusula [ , ...] ...]
```

Siendo las directivas las siguientes:

- **parallel**: Esta directiva nos indica que la parte de código que la comprende puede ser ejecutada por varios hilos. Se crea una tarea implícita para cada hilo perteneciente al equipo de hilos creado por el parallel.
- **for**: El equipo de hilos que se encuentra con el for ejecuta una o más fracciones de iteraciones como resultado de dividir el bucle delimitado por la directiva entre los hilos del equipo (el tamaño de cada partición dependerá de las cláusulas opcionales añadidas al for).
- **section y sections**: Indica secciones que pueden ejecutarse en paralelo pero por un único hilo.
- **single**: La parte de código que define esta directiva, solo se puede ejecutar un único hilo de todos los lanzados, y no tiene que ser obligatoriamente el hilo padre.
- **master**: La parte de código definida, solo se puede ejecutar por el hilo padre.
- **critical**: Solo un hilo puede estar en esta sección. Definen secciones críticas o condiciones de carrera.
- **atomic**: Se utiliza cuando la operación atañe a sólo una posición de memoria, y tiene que ser actualizada solo por un hilo simultáneamente. Operaciones tipo x++ o --x son las que usan esta cláusula.
- **flush**: Esta directiva resuelve la consistencia, al exportar a todos los hilos un valor modificado de una variable que ha realizado otro hilo en el procesamiento paralelo.

- **barrier**: Crea una barrera explícita en la que los hilos se quedan esperando hasta que todo el equipo la haya alcanzado.
- **task**: Permite la ejecución asíncrona mediante tareas explícitas. El bloque de código dentro de la directiva task es ejecutado por un único hilo, y la tarea es instanciada por el hilo que se encuentra con la directiva

Ejemplos de ejecución:

Debemos utilizar -fopenmp para poder compilar cualquier código que lo utilice.

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

int main(int argc, char *argv[])
{
    #pragma omp for
    for(int n=0; n<10; ++n)
    {
      printf(" %d", n);
    }
    printf(".\n");

    return 0;
}
```

```
alu@VDI-Ubuntu-EPS-2019:~/Descargas$ g++ -o o codigoEjemplo.cpp -fopenmp
alu@VDI-Ubuntu-EPS-2019:~/Descargas$ ./o
 0 1 2 3 4 5 6 7 8 9.
```

```c
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
     #pragma omp for
     for(int n=0; n<10; ++n) printf(" %d", n);
    }
    printf(".\n");

    return 0;
}
```

```
alu@VDI-Ubuntu-EPS-2019:~/Descargas$ g++ -o o codigoEjemplo.cpp -fopenmp
alu@VDI-Ubuntu-EPS-2019:~/Descargas$ ./o
 5 6 7 8 9 0 1 2 3 4.
```

La diferencia que ha habido entre estas 2 ejecuciones pese a que hacen lo mismo es clara, una al paralelizarse los eventos que suceden en el for, salen de forma descontinuada los números mientras que en la otra no.