Práctica 3
Paralelismo a nivel de hilos: Paralelización mediante OpenMP y programación asíncrona

Objetivos:

- Aprender a paralelizar una aplicación en una máquina paralela de memoria centralizada mediante hilos (*threads*) usando el estilo de *variables compartidas*.
- Estudiar la API de OpenMP y aplicar distintas estrategias de paralelismo en su aplicación.
- Estudiar la API de C++ de programación asíncrona para explotar el paralelismo funcional.
- Aplicar métodos y técnicas propios de esta asignatura para estimar las ganancias máximas y la eficiencia del proceso de paralelización.
- Aplicar todo lo anterior a un problema de complejidad y envergadura suficiente.

Desarrollo:
En esta práctica, tendréis que paralelizar, usando OpenMP y programación asíncrona, la solución a un problema dado para aprovechar los distintos núcleos de los que dispone cada ordenador de prácticas. Se paralelizará por tanto para un sistema multiprocesador (máquina paralela de memoria centralizada), en el que todos los núcleos de un mismo encapsulado ven la misma memoria, es decir, un puntero en un núcleo es el mismo puntero para el resto de los núcleos del microprocesador.

Tarea 1.1 Entrenamiento previo OpenMP:

Observa el siguiente programa en C donde se suman dos vectores de *floats* empleando OpenMP para paralelizar el cálculo.

```
#include <omp.h>
#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 */

}
```

Revisa la documentación de OpenMP (API, tutoriales, este enlace: https://computing.llnl.gov/tutorials/openMP/, etc...) y responda:

1.1.1 ¿Para qué sirve la variable **`chunk`**?
1.1.2 Explica **completamente** el *pragma* :

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

- ¿Por qué y para qué se usa **`shared(a,b,c,chunk)`** en este programa?
- ¿Por qué la variable `i` está etiquetada como **`private`** en el pragma?

1.1.3 ¿Para qué sirve **`schedule`**? ¿Qué otras posibilidades hay?
1.1.4 ¿Qué tiempos y otras medidas de rendimiento podemos medir en secciones de código paralelizadas con OpenMP?
NOTA: para compilar con openMP añade `-fopenmp` a las opciones de compilación.

Tarea 1.2: Entrenamiento previo std::async
Observa el siguiente programa en C++ donde se llama a dos funciones con `std::async`:

```
#include <iostream>
#include <future>
#include <chrono>
#include <thread>

int task(int id, int millis) {
    std::this_thread::sleep_for(std::chrono::milliseconds(millis));
    std::cout<<"Task "<<id<<" completed"<<std::endl;
    return id;
}

int main() {
    auto start = std::chrono::high_resolution_clock::now();
    std::future<int> task1 = std::async(std::launch::async, task, 1, 2000);
    std::future<int> task2 = std::async(std::launch::async, task, 2, 3000);

    task1.wait();
    int taskId = task2.get();

    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);

    std::cout<<"Completed in: "<<elapsed.count()<<"ms"<<std::endl;
}
```

Revisa la documentación de std::async y contesta a las siguientes preguntas:
1.2.1 ¿Para qué sirve el parámetro `std::launch::async`?
1.2.2 Calcula el tiempo que tarda el programa con `std::launch::async` y `std::launch::deferred`. ¿A qué se debe la diferencia de tiempos?
1.2.3 ¿Qué diferencia hay entre los métodos `wait` y `get` de `std::future`?
1.2.4 ¿Qué ventajas ofrece `std::async` frente a `std::thread`?

Tarea 2: Estudio del API OpenMP [Parte individual obligatoria (25% de la nota)]

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

Cada miembro del grupo* deberá realizar un pequeño tutorial a modo de "libro de recetas de paralelismo con OpenMP" con **distintos ejemplos de aplicación** del paralelismo que ofrece OpenMP en función de distintas **estructuras de software**.

Tarea 3: Uso de reduction y sections

Dado el siguiente código en C++:

```
#include <iostream>
#include <vector>
#include <random>
#include <omp.h>
#include <chrono>
#include <future>

std::random_device os_seed;
const int seed = 1;

std::mt19937 generator(seed);
std::uniform_int_distribution<> distribute(0, 1000);

double average(const std::vector<double> &v)
{
  double sum = 0.0f;
  for(int i=0;i<v.size();i++)
    sum += v[i];
  return sum/v.size();
}

double maximum(const std::vector<double> &v)
{
  double max = 0.0f;
  for(int i=0;i<v.size();i++)
    if (v[i]>max) max = v[i];
  return max;
}

double minimum(const std::vector<double> &v)
{
  double min = 1.0f;
  for(int i=0;i<v.size();i++)
    if (v[i]<min) min = v[i];
  return min;
}

int main(){

  int size = 100000000;
  std::vector<double> v(size);
  for(int i=0;i<v.size();i++)
    v[i] = distribute(generator)/1000.0;

  auto start=std::chrono::steady_clock::now();
  double min, max, avg;

  min = minimum(v);
  max = maximum(v);
  avg = average(v);

  auto end=std::chrono::steady_clock::now();
  std::chrono::duration<double> elapsed_seconds=end - start;

  std::cout<<"Elapsed: "<<elapsed_seconds.count()<<std::endl;
  std::cout<<"Min: "<<min<<" Max: "<<max<<" AVG: "<<avg<<std::endl;
```

```
}
```

Ignorando el tiempo de inicialización del vector y partiendo siempre del código inicial, responde a las siguientes preguntas:

3.1: ¿Cuánto tarda en ejecutarse el programa secuencial?

3.2: Utiliza la cláusula `reduction` de openMP para paralelizar los cálculos de las funciones `minimum`, `maximum` y `average`. ¿Cuánto tarda en ejecutarse el programa?

3.3: Utiliza la cláusula sections para que cada hilo ejecute una función. ¿Cuánto tarda en ejecutarse el programa?

3.4: Utiliza ahora `std::async` para lanzar cada función en una tarea en lugar de sections. ¿Cuánto tarda en ejecutarse el programa? NOTA: asegúrate de usar `std::ref` para pasar el vector por referencia en lugar de por valor para evitar una copia en cada llamada.

3.5: Combina `reduction` con `sections`. ¿Cuánto tarda en ejecutarse el programa? ¿Se observa mejora? ¿Por qué? NOTA: mira esta función omp_set_max_active_levels

3.6: Combina `reduction` de openMP con `std::async`. ¿Cuánto tarda ahora en ejecutarse? ¿Se observa mejora? ¿Por qué?

3.7: Calcula la ganancia de 3.2, 3.3, 3.4, 3.5 y 3.6 con respecto a 3.1. ¿Qué versión obtiene la mejor ganancia? ¿Por qué?

Tarea 4: Condiciones de carrera

Una condición de carrera se produce cuando varios hilos leen y/o escriben sobre la misma variable sin tener en cuenta el valor modificado por otro hilo. Por ejemplo:

```
#include <iostream>
#include <omp.h>
int main() {
    int max = 0;
    int min = 1000;
    #pragma omp parallel for
    for (int i=1000;i>=0;i--) {
        if (i > max) max = i;
        if (i < min) min = i;
    }
    std::cout<<"Max: "<<max<<" Min: "<<min<<std::endl;
}
```

Podemos observar que existe una dependencia entre la lectura de la variable `max` y su escritura. Cada hilo debe evaluar el valor actual de `max` y actualizarlo si toca (lo mismo ocurre con `min`). Varios hilos pueden leer el valor actual de `max` al mismo tiempo y actualizarlo a la vez de forma que el valor final será el del último hilo en lugar del mayor.

4.1: Prueba el código compilado con openMP. ¿Cuál es el resultado obtenido? ¿Es correcto? NOTA: usa `watch -n1 ./program` para ejecutar el programa cada segundo y comprueba que el resultado sea el mismo siempre.

4.2: Añade la cláusula `#pragma omp critical` sobre cada if. ¿Cuál es el resultado obtenido? ¿Es correcto ahora?

Observa el siguiente programa en C++ donde se inicializa un vector stl y se rellena con valores:

```
#include <vector>
#include <iostream>
#include <chrono>

int main() {
  int size = 10000000;
  // default initialization and add elements with push_back
  auto start = std::chrono::high_resolution_clock::now();

  std::vector<float> v1;

  for (int i = 0; i < size; i++)
    v1.push_back(i);

  auto end = std::chrono::high_resolution_clock::now();
  auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
  std::cout<<"Default initialization of "<<v1.size()<<" elements:
"<<elapsed.count()<<"ms"<<std::endl;

  // initialized with required size and add elements with direct access
  start = std::chrono::high_resolution_clock::now();

  std::vector<float> v2(size);
  for (int i = 0; i < size; i++)
    v2[i] = i;

  end = std::chrono::high_resolution_clock::now();
  elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
  std::cout<<"Initialization with size of "<<v2.size()<<" elements:
"<<elapsed.count()<<"ms"<<std::endl;
}
```

Responde a las siguientes preguntas:

4.3: ¿Cuál de las dos formas de inicializar el vector y rellenarlo es más eficiente? ¿Por qué?

4.4: ¿Podría ocurrir algún problema al paralelizar alguno de los bucles for? ¿Por qué?

4.5: Paraleliza con OpenMP ambos bucles sin modificar el código (sólo añadiendo pragma omp parallel for y/o pragma omp critical si es necesario). ¿Cuál es la ganancia que se obtiene? Explica los resultados obtenidos.

<u>Tarea 5:</u> Usa las implementaciones secuenciales de la Tarea 2.1.3 de la práctica 2 y paraleliza con OpenMP sobre una máquina paralela de memoria compartida:

- Cálculo de PI como la integral definida entre 0 y 1 de la derivada del arcotangente. Mirad el enlace: https://lsi2.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php?tuto=03_pi
- Cálculo de PI mediante el Método de Montecarlo. Véase el enlace https://www.geogebra.org/m/cF7RwK3H

Calcule los siguientes tiempos para cada programa:

- Tiempo secuencial del programa sin paralelizar.

- Tiempo paralelo del programa paralelizado ejecutado con diferente número de threads T=1,2,3,4,5,6 …Represente gráficamente estos tiempos y comente los resultados. ¿Se observa ganancia en velocidad? ¿Por qué?
- Compara los tiempos obtenidos con la versión de MPI. ¿Cuál es más rápida? ¿Por qué?

Notas generales a la práctica:

- La implementación realizada tendrá que poder ejecutarse bajo el sistema operativo Linux del laboratorio, y tomará 4 sesiones (8h)
- Es obligatorio entregar un Makefile con las reglas oportunas para compilar y limpiar los distintos programas implementados (make clean) de manera sencilla.
- Las/los estudiantes entregarán, además de la aplicación desarrollada, una memoria, estructurada según indicaciones del profesor, con la información obtenida.
- Entrega: se entregará tras 4 sesiones de prácticas.
- Los trabajos teóricos/prácticos realizados han de ser originales. La detección de copia o plagio supondrá la calificación de "0" en la prueba correspondiente. Se informará a la dirección de Departamento y de la EPS sobre esta incidencia. La reiteración en la conducta en esta u otra asignatura conllevará la notificación al vicerrectorado correspondiente de las faltas cometidas para que estudien el caso y sancionen según la legislación.