UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestciones

Tecnología de la Programación Paralela (TPP).

Ejercicios C++ Sesión 3

Introducción

Esta serie de ejercicios es continuación de la Sesión 2 y consta de ejercicios avanzados de C++.

Ejercicio 1: Concurrencia en C++ con `thread`

En este ejercicio se va a paralelizar el bucle principal del código del `ejercicio1.cpp`. Como una primera aproximación se creará un thread por cada iteración del bucle principal, es decir, cada thread creado se encargará de calcular la media y la desviación típica de un vector. Para ello, se van a seguir los siguientes pasos:

1. Se creará un vector llamado `task` dinámico (`new`) de threads. (Hay que acordarse de eliminarlo al final del programa.)
2. Dentro de la iteración del bucle se asignará la tarea específica al thread:

   ```
   task[v] = thread{ /* tarea */ };
   ```

   Esta tarea estará formada por todo el código de la iteración. La implementación será realizada mediante una expresión *lambda*. Para construir la *lambda* es conveniente tener en cuenta lo siguiente:

   - Las variables privadas a cada thread deberían ser declaradas dentro del cuerpo de instrucciones de la *lambda*.
   - Los vectores (`media`, `desvt`, `M` y `tam`) son variables compartidas, pero se asume que cada thread accede a "su" posición dentro de dicho vector, por lo tanto, el código será correcto tanto si se "capturan" por valor (`[=]`) como por referencia (`[&]`).
   - Existen dos opciones para pasar la variable `v` a la *lambda*: 1) como argumento del operador `operator()`, o 2) capturada por valor. Se sugiere utilizar la primera forma. En caso de utilizar la segunda forma, es necesario pasar, como segundo argumento del constructor de `thread` el valor `v`. Esto es:

     ```
     task[v] = thread{ Lambda, v };
     ```

3. Antes de terminar, no hay que olvidar sincronizar los threads.
4. Compilación:

   ```
   g++ -o ejercicio1 ejercicio1.cpp -std=c++11 ctimer.c -lpthread
   ```

Ejercicio 2: Concurrencia en C++ con `async`

Este ejercicio consiste en realizar una implementación basada en `async`. Para ello hay que tener en cuenta lo siguiente:

- En cada iteración del bucle principal se realizará una llamada a la función `async`. El primer argumento será la política de lanzamiento: `std::launch::async` o `std::launch::deferred`. El segundo argumento será la expresión *lambda* y el tercero el valor del argumento de la función *lambda*, es decir, lo mismo que se le pasa a la clase `thread`.
- Si se utiliza `std::launch::async` no hay mucho más que hacer, pero también se observará que las prestaciones obtenidas son pobres.
- Para utilizar la política `std::launch::deferred` es necesario crear un vector como el siguiente:

  ```
  vector<future<void>> task(n_vectores);
  ```

  para después realizar los lanzamientos de las tareas de este modo:

  ```
  task[v] = async( launch::deferred, Lambda, v );
  ```

- Después del bucle principal es necesario crear otro para llamar a la función `wait()` de la clase `future<void>` sobre cada tarea, esto es, `tarea[v].wait()`. (Obsérvese que es muy similar al `join()` de la clase `thread`).