Práctica 5

Optimización y Sincronización en OpenMP

Jordi Blasco Lozano
Computación de alto rendimiento
Grado en Inteligencia Artificial

Indice:

1. Desarrollo práctico guiado

Ejercicio 1:

El objetivo de este ejercicio será comprender como podemos aplicar la cláusula ‘reduction’, que función desempeña en nuestro código y que ventajas obtenemos.
Nuestro código ejecuta 3 bucles con 1000000 iteraciones, 1 de los bucles se ejecuta en modo secuencial, otro de los bucles se ejecuta en paralelo sin hacer uso de la cláusula ‘reduction’ y el ultimo bucle se ejecuta en paralelo también, pero haciendo uso de esta cláusula. Dentro de cada bucle se hace una suma de +i por cada iteración, el tiempo que tarda en ejecutarse cada tipo de bucle se guarda también. Al final calculamos el speed-up entre el secuencial y los bucles paralelos. He obtenido estos resultados:

```
root@5fdfe294afc7:/workdir/parte2# ./ejec_ejercicio1

Secuencial - Suma: 500000500000 Tiempo: 0.00369127 segundos
Paralela sin reduction - Suma: 60477617742 Tiempo: 0.00792436 segundos
Paralela con reduction - Suma: 500000500000 Tiempo: 0.00027795 segundos

Speed-up (secuencial / reduction): 13.2803
Speed-up (secuelcial / non_reduction): 0.465813
```

Como podemos observar la versión paralela sin reduction nos devuelve una suma invalida, esto se debe a las condiciones de carrera. Las condiciones de carrera ocurren cuando varios hilos intentan modificar una variable compartida de manera desincronizada.

Para paliar esta situación usamos la cláusula ‘reduction’. En nuestro código, en la versión paralela sin reduction, la variable 'suma' es compartida por todos los hilos sin sincronización esto provoca condiciones de carrera y un resultado final erróneo. Por otra parte, con reduction cada hilo mantiene una copia local de la variable 'suma' y al finalizar se combinan, garantizando un resultado correcto sin la sobreescritura simultánea.

En nuestra ejecución hemos obtenido que el bucle paralelo con reduction se ejecuta 13.28 veces más rápido que el bucle secuencial, esto supone una gran mejora de rendimiento. Al tener 16 hilos esta mejora es bastante considerable.

Ejercicio 2:

El objetivo de este ejercicio es comprender el papel de las técnicas de balanceo de carga y las ventajas que ofrecen unas sobre otras. Cuando se asignan tareas de diferente complejidad a distintos hilos, es común que algunos terminen más rápido y queden inactivos mientras esperan a que otros, con tareas más pesadas, finalicen. Este desequilibrio se puede mitigar utilizando la función omp_set_schedule() de OpenMP.

La función omp_set_schedule() permite configurar dos parámetros esenciales para la asignación de iteraciones entre hilos:

1. El tipo de tecnica de asignación (schedule), que puede ser, estática, dinámica o guiada.
2. El tamaño del chunk, que define cuántas iteraciones se asignan a un hilo en cada asignación.

omp_sched_static: Divide las iteraciones de manera fija y equitativa entre los hilos. Es ideal cuando cada iteración tiene un costo de procesamiento similar, ya que minimiza la sobrecarga.
omp_sched_dynamic: Asigna bloques de iteraciones de forma dinámica. Cada vez que un hilo termina su bloque, solicita el siguiente, lo que favorece a las aplicaciones con cargas de trabajo variables, aunque introduce una mayor sobrecarga de gestión.
omp_sched_guided: Inicialmente asigna bloques grandes y, conforme el trabajo avanza, reduce progresivamente el tamaño de estos bloques. Esto combina menor sobrecarga al inicio con flexibilidad para equilibrar la carga al final de la ejecución.

El programa que hemos compilado y ejecutado para poner en practica estas tecnicas usaba un bucle que se ejecutaba 1 000 000 en el cual se sumaba el cuadrado de la iteración a la suma global de esta forma ‘suma = suma + i * i’, este bucle se ejecuta 3 veces con las 3 distintas tecnicas de balanceo de carga y se cronometran. Hemos obtenido estos resultados finales:

```
root@1352b9d4af3e:/workdir/parte2# ./ejec_ejercicio2
Schedule: static - Suma: 16866193336416 Tiempo: 0.006377 segundos
Schedule: dynamic - Suma: 16866193336416 Tiempo: 0.000257 segundos
Schedule: guided - Suma: 16866193336416 Tiempo: 0.000299 segundos
```

Bajo estos resultados podemos concluir que en este caso las tecnicas dynamic y guided son superiores ya que, como la complejidad del problema se va incrementando en el orden i^2, las cargas que soportan los hilos son pesadas y muy variables lo que favorece a estas tecnicas. El tamaño del chunk que hemos usado esta bien, pero si lo hubieramos bajado a 100 se notaria más la diferencia de tiempo entre la static con las tecnicas dynamic y guided al usar menos iteraciones por hilo.

Ejercicio 3:

En este ejercicio veremos cómo usar las directivas ***atomic, critical y barrier***, como afectan a la sincronización y al rendimiento. Hemos implementado un código que paraleliza un bucle al cual se debe de acceder a la variable **suma** esta variable es compartida por los hilos por lo que va a generar situaciones de carrera y si no se usa la adecuada sincronización devolverá un resultado erróneo. Por esto ejecutaremos el bucle sin protección, y posteriormente con dos tipos de directivas de sincronización ***atomic*** y ***critical,*** para así, comparar los resultados. Los resultados que he obtenido al realizar el bucle de la **suma += 1** (1000000 veces) ha sido el siguiente:

```
Sin protección - Suma: 12928603 Tiempo: 0.979788
Con atomic - Suma: 100000000 Tiempo: 2.643827
Con critical - Suma: 100000000 Tiempo: 14.166273
```

Viendo este resultado podemos sacar varias conclusiones.

Los códigos sin protección con condiciones de carrera van a dar siempre resultados erróneos, para paliar esto observamos como usando ***atomic*** obtenemos un mejor tiempo de ejecución que usando ***critical***. Esto se debe a que critical funciona mejor si queremos proteger secciones criticas complejas (no es nuestro caso), ya que genera una sobrecarga significativa al hacer que los hilos tengan que esperar su turno. Por otro lado si usamos atomic vemos una mejora de rendimiento ya que es más eficiente al realizar operaciones simples.

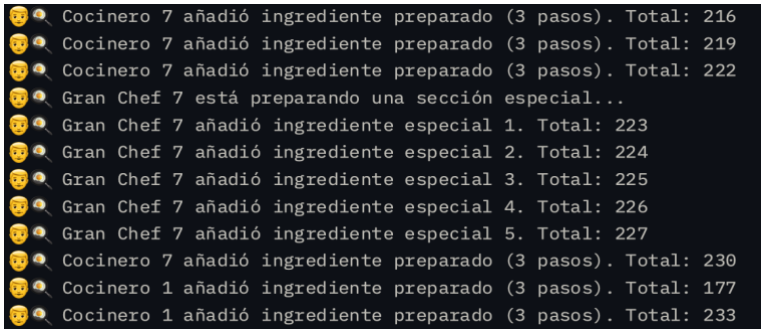
Ejercicio 4:

En este ejercicio aplicaremos las directivas de sincronización a un código que simula la colaboración de múltiples cocineros (en paralelo) para preparar un plato en 100 pasos. Cada iteración representa un paso en la preparación, en el que la variable compartida ***plato*** se va actualizando con la cantidad de ingredientes añadidos. En el paso 50, se implementa un comportamiento especial: la ejecución de una sección crítica por parte del "Gran Chef", quien añade 5 ingredientes de forma secuencial. En los demás pasos, cada cocinero actualiza la variable ***plato*** de forma concurrente mediante la directiva ***atomic***, asegurando así la integridad de la variable frente a condiciones de carrera.

La directiva ***critical*** se utiliza para delimitar una sección de código en la que únicamente un hilo puede ejecutar el bloque en cualquier instante. En este caso, se aplica en el paso 50 para garantizar que solo un hilo **(el Gran Chef)** realice la adición secuencial de 5 ingredientes especiales.
Al llegar al paso 50, aunque varios hilos puedan identificar la condición i == 50, la sección crítica obliga a que se ejecute de forma exclusiva. Esto evita que varios hilos compitan por ejecutar el mismo bloque y que se produzcan interferencias. Dentro del bloque critical, el bucle interno que añade los 5 ingredientes se ejecuta de manera secuencial. Esto simula el rol centralizado y especial del Gran Chef, asegurando que la operación se realice de forma ordenada y sin interrupciones.

La directiva ***atomic*** se utiliza para garantizar que la actualización de la variable compartida *plato* se realice de forma indivisible. Esto significa que cada suma (en este caso, la adición del valor de la variable ***preparado***) se ejecuta sin ser interrumpida por otro hilo, evitando condiciones de carrera.
A diferencia de la sección crítica, atomic tiene una sobrecarga menor porque está optimizada para operaciones simples (como una suma), lo que mejora la eficiencia en entornos de alta concurrencia.
Al minimizar el bloqueo de hilos, la directiva *atomic* permite que múltiples hilos actualicen la variable de forma casi simultánea, lo que resulta en un mejor rendimiento global del programa.

Para optimizar el programa podríamos En lugar de utilizar múltiples operaciones ***atomic***, se podría emplear la cláusula **reduction(+:plato)** en la directiva ***omp for***. Esto permitiría que cada hilo acumule su propio subtotal y, al finalizar, se realice una suma global de forma eficiente.
Cada hilo podría mantener una variable local para acumular los ingredientes añadidos en cada iteración y, posteriormente, actualizar la variable compartida *plato* en una única operación atómica o mediante reducción. Esto reduciría el número de operaciones atómicas y mejoraría la escalabilidad.



2. Sistema de Reconocimiento de Voz y Geolocalización Mundial

El objetivo era desarrollar en c (he usado c++ para poder usar std) un código que aplicara toda la teoría anterior siguiendo las directrices las cuales piden desarrollar un sistema paralelo eficiente que simule un reconocimiento de voz. El objetivo del sistema era gestionar las sugerencias de cambios de idioma de los usuarios. La lógica de la aplicación que he programado es la siguiente:

Inicialización y Preparación:
- Se define el número total de usuarios a procesar y se reserva un espacio para almacenar los datos de cada uno (identificador, idioma detectado, coordenadas GPS, región y bandera de adaptación de idioma).
- Se inicia la medición del tiempo total de procesamiento para evaluar el rendimiento.

Procesamiento Paralelo de Cada Usuario:
- Cada usuario recibe un ID único para su identificación.
Se simula la detección del idioma a partir de una muestra de audio. Esto se logra eligiendo aleatoriamente un idioma de un conjunto predefinido, representando la "huella de voz" del usuario.
- Se generan valores aleatorios para la latitud y longitud, simulando la ubicación geográfica de cada usuario a nivel mundial.
- Con base en las coordenadas, se asigna al usuario una región. La lógica puede ser, por ejemplo, clasificar la ubicación según si la latitud y la longitud son positivas o negativas.
- Se obtiene el idioma predominante de la región asignada y se compara con el idioma detectado del usuario. Si ambos difieren, se marca que se sugiere una adaptación del idioma para ese usuario.

Análisis Estadístico Paralelo:
- Se recorren los datos de todos los usuarios para contabilizar cuántos tienen cada uno de los idiomas detectados. Se utilizan mecanismos de sincronización para evitar condiciones de carrera durante la actualización de estos contadores.
- Similarmente, se cuenta cuántos usuarios en cada región requieren adaptación de idioma.
- Se suma el número total de usuarios que necesitan adaptación utilizando una reducción paralela, consolidando los resultados de cada hilo de procesamiento.

Impresión y Presentación de Resultados:
- Se muestra el tiempo total invertido en el procesamiento paralelo.
- Se despliegan las estadísticas de usuarios por idioma y por región, destacando aquellos que requieren adaptación.
- Finalmente, se presenta una muestra (por ejemplo, de los primeros 5 usuarios) con sus detalles de geolocalización, idioma detectado y la sugerencia de adaptación, simulando la generación de un "mapa global".

El procesamiento de los datos de cada usuario se realiza en paralelo. Cada hilo procesa un subconjunto de usuarios de forma independiente, lo que permite reducir el tiempo de ejecución esto supone que tengamos que emplear varias directivas para manejar las variables compartidas y la sincronización entre hilos. Se emplean directivas como ***parallel for*** para el procesamiento de usuarios, ***reduction*** para contar el total de adaptaciones y directivas ***atomic/critical*** para sincronizar el acceso a variables compartidas en el análisis estadístico. Además, se utiliza ***barrier*** para sincronizar los hilos en puntos críticos del procesamiento. Se cuida el acceso a estructuras de datos globales (como mapas de estadísticas) mediante secciones críticas y directivas atómicas donde corresponde, evitando condiciones de carrera.

Para gestionar el balanceo de carga se utiliza ***schedule(dynamic)*** en los bucles paralelos para balancear la carga, considerando que algunos análisis (por ejemplo, la simulación de procesamiento de audio) pueden ser más costosos que otros.

Código:

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
#include <omp.h>
#include <map>
#include <iomanip>
// Estructura para almacenar los datos de cada usuario
struct UserData {
    int id;
    std::string detectedLanguage;
    double latitude;
    double longitude;
    std::string region;
    bool languageAdaptation;
};
// Función que simula la detección de idioma a partir de una muestra de audio.
// Se selecciona aleatoriamente un idioma de una lista.
std::string simulateLanguageDetection(unsigned int* seed) {
    int idx = rand_r(seed) % 4;
    static const std::string languages[4] = {"English", "Spanish", "French", "Portuguese"};
    return languages[idx];
}
// Función que determina la región del usuario en función de sus coordenadas GPS.
std::string getRegion(double lat, double lon) {
    if (lat >= 0) {
        if (lon >= 0)
            return "Region A";  // Predominante: English
        else
            return "Region B";  // Predominante: French
    } else {
        if (lon >= 0)
            return "Region C";  // Predominante: Spanish
        else
            return "Region D";  // Predominante: Portuguese
    }
}
```

```
// Función que devuelve el idioma predominante según la región.
std::string getPredominantLanguage(const std::string& region) {
    if (region == "Region A") return "English";
    else if (region == "Region B") return "French";
    else if (region == "Region C") return "Spanish";
    else return "Portuguese";
}

int main() {
    // Número de usuarios a simular
    const int numUsers = 1000000;
    std::vector<UserData> users(numUsers);

    // Medición del tiempo de procesamiento
    double startTime = omp_get_wtime();

    // Procesamiento paralelo de cada usuario utilizando OpenMP
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < numUsers; i++) {
        // Cada hilo utiliza su propia semilla para la generación aleatoria
        unsigned int seed = i + omp_get_thread_num();  // Se puede agregar omp_get_thread_num() para mayor variabilidad

        // Simulación del reconocimiento de la huella de voz: asignación de un ID único
        users[i].id = i;

        // Simulación de la detección del idioma a partir de la muestra de audio
        users[i].detectedLanguage = simulateLanguageDetection(&seed);

        // Simulación de la obtención de coordenadas GPS (latitud y longitud aleatorias)
        double randLat = -90.0 + (rand_r(&seed) / (double)RAND_MAX) * 180.0;
        double randLon = -180.0 + (rand_r(&seed) / (double)RAND_MAX) * 360.0;
        users[i].latitude = randLat;
        users[i].longitude = randLon;

        // Determinar la región basada en las coordenadas GPS
        users[i].region = getRegion(randLat, randLon);

        // Obtener el idioma predominante para la región y comparar con el idioma detectado
        std::string predominantLanguage = getPredominantLanguage(users[i].region);
        users[i].languageAdaptation = (users[i].detectedLanguage != predominantLanguage);
    }

    double processingTime = omp_get_wtime() - startTime;

    // Análisis estadístico: conteo de usuarios por idioma
    std::map<std::string, int> languageCounts;
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < numUsers; i++) {
        // Se utiliza critical para evitar condiciones de carrera en el acceso al mapa
        #pragma omp critical
        {
            languageCounts[users[i].detectedLanguage]++;
        }
    }
    // Análisis estadístico: conteo de usuarios que requieren adaptación por región
    std::map<std::string, int> adaptationCounts;
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < numUsers; i++) {
        if (users[i].languageAdaptation) {
            #pragma omp critical
            {
                adaptationCounts[users[i].region]++;
            }
        }
    }
```

```
    // Conteo total de usuarios que requieren adaptación utilizando la directiva reduction
    int totalAdaptations = 0;
    #pragma omp parallel for reduction(+:totalAdaptations) schedule(dynamic)
    for (int i = 0; i < numUsers; i++) {
        if (users[i].languageAdaptation)
            totalAdaptations++;
    }

    // Impresión de resultados y estadísticas
    std::cout << "Tiempo de procesamiento: " << processingTime << " segundos.\n";
    std::cout << "\nDistribución de usuarios por idioma:\n";
    for (auto &entry : languageCounts) {
        std::cout << "  " << entry.first << ": " << entry.second << "\n";
    }

    std::cout << "\nUsuarios que requieren adaptación de idioma por región:\n";
    for (auto &entry : adaptationCounts) {
        std::cout << "  " << entry.first << ": " << entry.second << "\n";
    }
    std::cout << "\nTotal de usuarios que requieren adaptación: " << totalAdaptations << "\n";

    // Simulación de la generación de un mapa global: se muestran los datos de geolocalización de los primeros 5 usuarios
    std::cout << "\nMuestra de datos de geolocalización (ultimos 15 usuarios):\n";
    for (int i = numUsers - 1; i > numUsers - 16; i--) {
        std::cout << "Usuario " << users[i].id
                  << " - Latitud: " << std::fixed << std::setprecision(2) << users[i].latitude
                  << ", Longitud: " << users[i].longitude
                  << ", Región: " << users[i].region
                  << ", Idioma detectado: " << users[i].detectedLanguage
                  << ", Adaptación sugerida: " << (users[i].languageAdaptation ? "Sí" : "No")
                  << "\n";
    }

    return 0;
}
```

Salida:

```
Tiempo de procesamiento: 0.05226 segundos.

Distribución de usuarios por idioma:
  English: 250372
  French: 249520
  Portuguese: 250117
  Spanish: 249991

Usuarios que requieren adaptación de idioma por región:
  Region A: 186990
  Region B: 187774
  Region C: 188316
  Region D: 187965

Total de usuarios que requieren adaptación: 751045

Muestra de datos de geolocalización (ultimos 5 usuarios):
Usuario 999999 - Latitud: 64.70, Longitud: 69.19, Región: Region A, Idioma detectado: Portuguese, Adaptación sugerida: Sí
Usuario 999998 - Latitud: 67.04, Longitud: -73.21, Región: Region B, Idioma detectado: French, Adaptación sugerida: No
Usuario 999997 - Latitud: -73.16, Longitud: 67.46, Región: Region C, Idioma detectado: Portuguese, Adaptación sugerida: Sí
Usuario 999996 - Latitud: -21.83, Longitud: -144.59, Región: Region D, Idioma detectado: French, Adaptación sugerida: Sí
Usuario 999995 - Latitud: 20.22, Longitud: -146.32, Región: Region B, Idioma detectado: Spanish, Adaptación sugerida: Sí
```