Universitat d'Alacant
Universidad de Alicante

Práctica Semana 5: Optimización y Sincronización en OpenMP

Parte 1: Introducción General

En esta práctica de la **Semana 5**, los estudiantes se adentrarán en técnicas avanzadas de optimización y sincronización utilizando **OpenMP**. El objetivo principal es comprender cómo escribir código paralelo eficiente y seguro, explorando diversas estrategias de optimización, control de acceso a datos y balanceo de carga entre hilos.

Se abordarán aspectos clave como:

- Prevención de condiciones de carrera.
- Optimización del uso de memoria en entornos paralelos.
- Estrategias para distribuir la carga de trabajo eficientemente.
- Técnicas para medir el rendimiento y comparar la eficiencia de diferentes enfoques.

Al finalizar esta práctica, los estudiantes habrán implementado múltiples soluciones paralelas utilizando OpenMP, analizado su rendimiento, y adoptado buenas prácticas de programación paralela.

Parte 1: Introducción Guiada a Conceptos Clave

Objetivo: Asegurar que los estudiantes comprendan los conceptos esenciales antes de realizar la práctica.

1.1 Condiciones de carrera

Una **condición de carrera** ocurre cuando múltiples hilos acceden y modifican simultáneamente una variable compartida **sin sincronización adecuada**. Esto provoca resultados impredecibles o incorrectos debido a la sobreescritura de datos.

Ejemplo práctico: Incremento de un contador compartido sin protección.

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

int main() {
    int contador = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10000; i++) {
        contador += 1;  // Condición de carrera aquí
    }

    printf("Valor esperado: 10000\n");
    printf("Valor real (con condición de carrera): %d\n", contador);

    return 0;
```

```
}
```

Explicación:

- El **valor esperado** es **10000**.
- Sin embargo, debido a la condición de carrera, el resultado será menor porque múltiples hilos intentan leer, incrementar y escribir en la variable **`contador`** al mismo tiempo.

Solución: Uso de `atomic` para evitar la condición de carrera

La siguiente modificación usa `#pragma omp atomic` para evitar la condición de carrera.

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

int main() {
   int contador = 0;

   #pragma omp parallel for
   for (int i = 0; i < 10000; i++) {
      #pragma omp atomic
      contador += 1;  // Operación protegida por atomic
   }

   printf("Valor esperado: 10000\n");
   printf("Valor real (con atomic): %d\n", contador);

   return 0;
}
```

¿Qué cambió?

- **`#pragma omp atomic`** garantiza que la operación **`contador += 1`** se realice de forma atómica (no interrumpida) por cada hilo.
- Ahora el resultado será siempre **10000**.

2. Ejercicios Iniciales Guiados

🔹 *Ejercicio 1: Condición de carrera y `atomic`*

1. Implementa un bucle que incremente una variable global desde múltiples hilos **sin** protección.
2. Ejecuta el programa y observa si el resultado es el esperado.
3. Modifica el código para usar **`#pragma omp atomic`** y corrige la condición de carrera.
4. Compara el tiempo de ejecución antes y después del cambio.

🔹 *Ejercicio 2: Acumulación segura con `reduction`*

1. Crea un programa que sume los números del 1 al 1,000,000 utilizando múltiples hilos.
2. Implementa la solución **sin** usar `reduction` y observa los problemas que aparecen.
3. Corrige la solución usando **`#pragma omp reduction(+:suma)`**.
4. Mide los tiempos y calcula el **speed-up** respecto al código secuencial.

✍ *Ejercicio 3: Uso de `barrier` para sincronización*

1. Implementa un programa donde varios hilos actualicen un array de forma paralela.
2. Usa `#pragma omp barrier` para asegurarte de que todos los hilos completen la escritura antes de continuar a la siguiente fase del programa.
3. Verifica que los datos escritos sean correctos y que no haya sobrescrituras.

Parte 2: Desarrollo Práctico Guiado

Objetivo: Aplicar los conceptos aprendidos en ejercicios prácticos incrementales, abordando aspectos fundamentales como condiciones de carrera, sincronización entre hilos, balanceo de carga, y optimización de memoria.

? Ejercicio 1: Sumas Paralelas y Uso de `reduction`

Objetivo: Comprender y aplicar la cláusula **`reduction`** para realizar acumulaciones seguras en entornos paralelos.

Instrucciones:

1. Implementa un programa que sume los números del **1 al 1,000,000** usando **OpenMP**.
2. Crea tres versiones:
   - **Versión Secuencial:** Sin paralelización.
   - **Versión Paralela sin `reduction`:** Para observar condiciones de carrera.
   - **Versión Paralela con `reduction`:** Para corregir la condición de carrera.

Código Base:

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

#define N 1000000

int main() {
   long long suma = 0;

   // Versión secuencial
   double start = omp_get_wtime();
   for (int i = 1; i <= N; i++) {
      suma += i;
   }
   double end = omp_get_wtime();
   printf("Secuencial - Suma: %lld Tiempo: %f segundos\n", suma, end - start);

   // Versión paralela sin reduction (tendrá condición de carrera)
   suma = 0;
   start = omp_get_wtime();
   #pragma omp parallel for
   for (int i = 1; i <= N; i++) {
      suma += i;  // Condición de carrera aquí
   }
   end = omp_get_wtime();
   printf("Paralela sin reduction - Suma: %lld Tiempo: %f segundos\n", suma, end - start);

   // Versión paralela con reduction
   suma = 0;
   start = omp_get_wtime();
   #pragma omp parallel for reduction(+:suma)
   for (int i = 1; i <= N; i++) {
```

```
        suma += i;
    }
    end = omp_get_wtime();
    printf("Paralela con reduction - Suma: %lld Tiempo: %f segundos\n", suma, end - start);

    return 0;
}
```

Análisis:

- **¿Qué problemas observaste en la versión paralela sin `reduction`?**
- **¿Cómo mejora el rendimiento con `reduction`?**
- Calcula el **speed-up** comparando la versión secuencial y la versión paralela con `reduction`:

$$Speed - up = \frac{Tsecuencial}{Tparalelo}$$

Ejercicio 2: Balanceo de Carga con `schedule`

Objetivo: Analizar cómo las diferentes estrategias de balanceo de carga afectan el rendimiento de un programa paralelo.

Instrucciones:

1. Implementa un programa que calcule la suma de los cuadrados de los números del **1 al 1,000,000**.
2. Ejecuta el programa con tres estrategias de **`schedule`**:
    - **`static`**
    - **`dynamic`**
    - **`guided`**

Código Base:

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

#define N 1000000

int main() {
    long long suma = 0;

    omp_sched_t schedules[] = {omp_sched_static, omp_sched_dynamic, omp_sched_guided};
    const char* schedule_names[] = {"static", "dynamic", "guided"};

    for (int s = 0; s < 3; s++) {
        suma = 0;
        omp_set_schedule(schedules[s], 1000);  // Tamaño de chunk

        double start = omp_get_wtime();
        #pragma omp parallel for schedule(runtime) reduction(+:suma)
        for (int i = 1; i <= N; i++) {
            suma += i * i;  // Suma de cuadrados
```

```
        }
        double end = omp_get_wtime();

        printf("Schedule: %s - Suma: %lld Tiempo: %f segundos\n", schedule_names[s], suma, end - start);
    }

    return 0;
}
```

Análisis:

- **¿Cuál estrategia fue más eficiente?**
- **¿Por qué `dynamic` o `guided` podrían ser mejores en ciertos casos?**
- **¿Cómo afecta el tamaño de los chunks al rendimiento?**

Ejercicio 3: Sincronización y Condiciones de Carrera

Objetivo: Explorar cómo las directivas **`atomic`**, **`critical`** y **`barrier`** afectan la sincronización y el rendimiento.

Instrucciones:

1. Implementa un programa donde varios hilos actualicen una variable compartida.
2. Realiza tres versiones:
    - Sin protección.
    - Usando **`#pragma omp atomic`**.
    - Usando **`#pragma omp critical`**.

Código Base:

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

#define N 1000000

int main() {
    long long suma = 0;

    // Sin protección
    double start = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        suma += 1;  // Condición de carrera
    }
    double end = omp_get_wtime();
    printf("Sin protección - Suma: %lld Tiempo: %f\n", suma, end - start);

    // Con atomic
    suma = 0;
    start = omp_get_wtime();
```

```
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        #pragma omp atomic
        suma += 1;
    }
    end = omp_get_wtime();
    printf("Con atomic - Suma: %lld Tiempo: %f\n", suma, end - start);

    // Con critical
    suma = 0;
    start = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        #pragma omp critical
        {
            suma += 1;
        }
    }
    end = omp_get_wtime();
    printf("Con critical - Suma: %lld Tiempo: %f\n", suma, end - start);

    return 0;
}
```

Análisis:

- **¿Cuál enfoque fue más eficiente?**
- **¿En qué casos es mejor usar `atomic` en lugar de `critical`?**
- **¿Qué impacto tiene `critical` en el rendimiento comparado con `atomic`?**

□ Ejercicio 4: Ejemplo Complejo – Cocineros y el Gran Chef

Objetivo: Aplicar sincronización avanzada (**`atomic`**, **`critical`**, **`barrier`**) en un ejemplo colaborativo.

Enunciado:
Un equipo de cocineros trabaja en paralelo para preparar un gran plato. Cada cocinero añade ingredientes al plato, pero el **Gran Chef** debe realizar una tarea crítica añadiendo varios ingredientes especiales en orden. Todos los cocineros deben esperar a que el Gran Chef termine antes de continuar.

Código Base:

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

#define PASOS 100

int main() {
    int plato = 0;

    printf("□□ Cocineros colaborando para preparar un plato en %d pasos...\n", PASOS);
```

```
    #pragma omp parallel
    {
        int id = omp_get_thread_num();

        #pragma omp for
        for (int i = 0; i < PASOS; i++) {
            if (i == 50) {
                #pragma omp critical
                {
                    printf("□□□ Gran Chef %d está preparando una sección especial...\n", id);
                    for (int j = 1; j <= 5; j++) {
                        plato += 1;
                        printf("□□□ Gran Chef %d añadió ingrediente especial %d. Total: %d\n", id, j, plato);
                    }
                }
            } else {
                int preparado = 0;
                for (int k = 0; k < 3; k++) {
                    preparado += 1;
                }

                #pragma omp atomic
                plato += preparado;

                printf("□□ Cocinero %d añadió ingrediente preparado (%d pasos). Total: %d\n", id,
preparado, plato);
            }
        }
    }

    printf("□ Plato terminado con %d ingredientes.\n", plato);
    return 0;
}
```

Análisis:

- **¿Cómo afecta `critical` al flujo de trabajo?**
- **¿Qué ventajas aporta `atomic` en este escenario?**
- **¿Cómo se podría optimizar aún más este programa?**

□ Parte 3: Ejercicio Complejo – Sistema de Reconocimiento de Voz y Geolocalización Mundial

□ Objetivo

Desarrollar un sistema paralelo eficiente que simule un **reconocimiento de voz a nivel mundial** para usuarios de dispositivos Android. Este sistema debe ser capaz de identificar a cada usuario por su huella de voz, determinar el idioma que utiliza habitualmente, geolocalizarlo mediante GPS, y generar un mapa global con estos datos para fines estadísticos y comerciales.

El sistema debe procesar grandes volúmenes de datos rápidamente, aplicando técnicas de paralelización y sincronización aprendidas en esta unidad.

Enunciado del Proyecto

Contexto:
Una compañía global quiere lanzar una nueva aplicación de **reconocimiento de voz y geolocalización** para usuarios de teléfonos móviles Android. El sistema debe analizar a millones de usuarios simultáneamente, identificarlos por su huella de voz registrada previamente, y crear un mapa de distribución global con información sobre su idioma y localización.

✅Requisitos Funcionales

1. **Identificación de Usuario por Huella de Voz:**
   - Procesar una **base de datos masiva** de huellas de voz.
   - Cada hilo del programa se encargará de procesar un conjunto de huellas.
   - **Usar `reduction`** para acumular estadísticas globales como número de usuarios identificados por idioma.
2. **Detección de Idioma:**
   - Analizar muestras de audio para determinar el **idioma principal** del usuario.
   - Utilizar **`schedule(dynamic)`** para balancear la carga de trabajo, ya que algunos análisis de audio pueden ser más complejos que otros.
3. **Geolocalización Global:**
   - Acceder al GPS del dispositivo para obtener las coordenadas.
   - Asegurar la correcta actualización de datos utilizando **`atomic`** o **`critical`** para evitar condiciones de carrera.
4. **Adaptación de Idioma:**
   - Si el idioma habitual del usuario es diferente al predominante en su ubicación geográfica, el sistema sugerirá adaptar el idioma al entorno local.
   - Esta lógica debe implementarse usando **`critical`** para asegurar la correcta gestión de las sugerencias.
5. **Generación del Mapa Mundial:**
   - Generar un **mapa global** que muestre la distribución de los usuarios por idioma y ubicación.
   - Sincronizar los hilos mediante **`barrier`** antes de la generación final y usar **`reduction`** para consolidar los datos.
6. **Análisis Estadístico y Comercial:**
   - Calcular estadísticas globales como:
     - Número total de usuarios por idioma.
     - Áreas con mayor concentración de usuarios multilingües.
     - Zonas donde se recomienda promover servicios de traducción.
   - Optimizar el análisis usando técnicas de paralelización.

⚙ Indicaciones Técnicas y Reglas del Proyecto

- **Paralelización:**
  - Uso obligatorio de **OpenMP**.
  - Aplicar directivas como **`parallel for`**, **`reduction`**, **`atomic`**, **`critical`**, y **`barrier`**.
- **Sincronización:**

  - Proteger variables compartidas adecuadamente.
  - Escoger entre **`atomic`** y **`critical`** según la complejidad de la operación.
  - Usar **`barrier`** para sincronizar fases importantes.
- **Optimización:**
  - Experimentar con diferentes estrategias de **`schedule`** (**static**, **dynamic**, **guided**) para balancear la carga.
  - Medir el rendimiento usando **`omp_get_wtime()`** y calcular el **speed-up**.
- **Documentación:**
  - Entregar un informe detallado que incluya:
    - Descripción de la solución.
    - Código fuente comentado.
    - Gráficos o tablas de rendimiento.
    - Análisis de resultados y discusión de técnicas utilizadas.
    - Reflexión sobre mejoras y desafíos encontrados.

Anexo: Rúbrica de Evaluación

Criterio	Peso	Descripción
Implementación de paralelización	30%	Uso correcto de directivas OpenMP (`reduction`, `atomic`, `critical`, `schedule`, `barrier`).
Optimización y balanceo de carga	20%	Aplicación efectiva de estrategias de balanceo y optimización del rendimiento.
Sincronización adecuada	15%	Protección de variables compartidas y uso eficiente de `atomic` y `critical`.
Análisis de rendimiento	15%	Medición de tiempos, speed-up y eficiencia usando `omp_get_wtime()`.
Claridad y documentación del código	10%	Código limpio, bien comentado, y diagramas explicativos.
Originalidad y profundidad del análisis	10%	Propuestas innovadoras, reflexión crítica y discusión de posibles mejoras.

Anexo: Justificación Académica

Esta práctica ha sido diseñada para consolidar los conocimientos teóricos y prácticos adquiridos durante la **Semana 5**. A través del desarrollo de un ejercicio complejo, los estudiantes aplicarán técnicas avanzadas de paralelización, sincronización y optimización, enfrentándose a un escenario que simula un sistema de gran escala en el mundo real.

Competencias desarrolladas:

- **Programación paralela eficiente** aplicando OpenMP.
- **Optimización de recursos** mediante balanceo de carga y técnicas de sincronización.
- **Análisis y medición de rendimiento** para evaluar la eficiencia de las soluciones propuestas.
- **Documentación técnica** clara y precisa para comunicar los resultados obtenidos.
- **Resolución de problemas complejos** y desarrollo de pensamiento crítico.

Resultados de aprendizaje esperados:

1. Implementar aplicaciones paralelas eficientes utilizando OpenMP.
2. Aplicar técnicas de sincronización y optimización en entornos multiprocesador.
3. Analizar y comparar el rendimiento de soluciones paralelas frente a versiones secuenciales.
4. Documentar adecuadamente el proceso de desarrollo y análisis de resultados.
5. Proponer mejoras y alternativas basadas en los desafíos encontrados durante la práctica.