UA

Universidad
de Alicante

ESCUELA POLITÉCNICA SUPERIOR

Optimización y Sincronización en OpenMP

Curso académico 2024 / 2025
Semestre 2
Grado en Ingeniería en Inteligencia Artificial
Grupo 1

Profesora: Ricardo Moreno Rodríguez
Alumno: Santiago Álvarez Geanta

Universitat d'Alacant
Universidad de Alicante

Febrero 2025

ÍNDICE

Parte 1. Condición 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 con #pragma omp atomic**

Para evitar la condición de carrera en la modificación de una variable compartida, se puede utilizar ***#pragma omp atomic***. A continuación, se presenta un código en C utilizando OpenMP donde se incrementa un contador en un bucle paralelo:

```c
#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;
}
```

#pragma omp atomic garantiza que la operación contador += 1 se realice de forma atómica, evitando interrupciones entre la lectura y escritura de la variable por los distintos hilos.

Ejercicio 1: Condición de carrera y atomic

En esta primera versión del programa, implementamos un bucle donde varios hilos incrementan una variable global sin ningún tipo de protección. Debido a que múltiples hilos acceden a la variable **simultáneamente**, se produce una condición de carrera, lo que genera un resultado incorrecto ya que cada hilo lee el valor actual de la variable compartida contador, lo incrementa y lo escribe nuevamente. Como consecuencia **el valor final será menor al valor esperado** porque algunas actualizaciones se pierden.

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

int main() {
    int contador = 0;
    int N = 10000;
    double end;

    double start = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        contador += 1;  // Condición de carrera aquí
    }
    end = omp_get_wtime();
    printf("Sin atomic - Valor de contador: %d, Tiempo: %f segundos\n", contador, end - start);


    contador = 0;
    start = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        #pragma omp atomic
        contador += 1;
    }
    end = omp_get_wtime();
    printf("Con atomic - Valor de contador: %d, Tiempo: %f segundos\n", contador, end - start);

    return 0;
}
```

Para corregir el problema, se nos pide usar la directiva **#pragma omp atomic**, que garantiza que la operación de incremento se realice de manera atómica. De esta forma se bloquea la operación **contador += 1** para que ningún otro hilo pueda interrumpirla hasta que se complete. Como resultado, el valor final de contador será exactamente 10,000 en todas las ejecuciones.

Al compilar y ejecutar el archivo podemos ver que, además de obtener el verdadero valor de la variable contador, obtenemos una mayor eficiencia respecto al tiempo.

```
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA-2/2 CUATRI/CAR/P5$ ./E1

Sin atomic - Valor de contador: 2088, Tiempo: 0.013180 segundos
Con atomic - Valor de contador: 10000, Tiempo: 0.000230 segundos
```

Ejercicio 2 Acumulación segunda con *reduction*

En esta primera parte, varios hilos suman los números del 1 al 1,000,000 sin protección. Como resultado, algunos valores pueden perderse debido a la interferencia entre hilos, causando una suma incorrecta. Para corregir la condición de carrera, usamos #pragma omp reduction(+:suma), lo que permite que cada hilo tenga su propia variable local y luego combine los resultados al final de la operación.

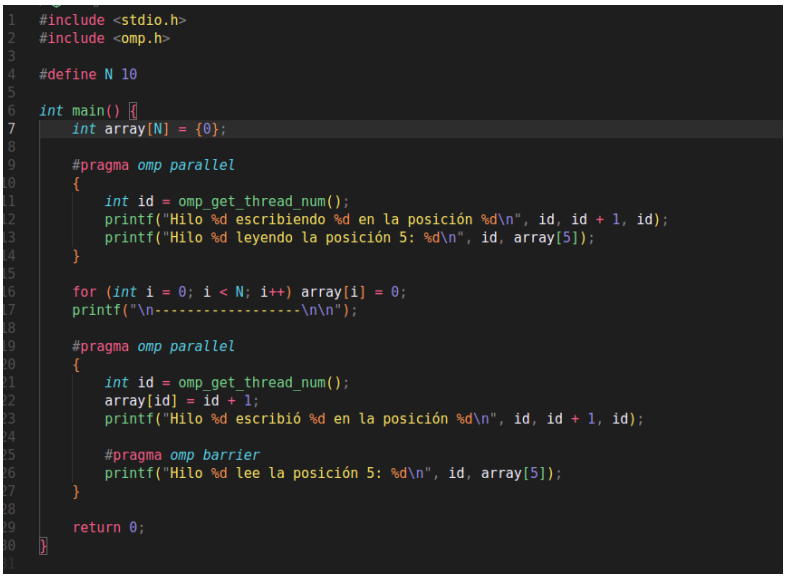
```c
#include <stdio.h>
#include <omp.h>

#define N 1000000

int main() {
    long long suma = 0;
    double end;

    double 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("Sin reduction - Suma: %lld, Tiempo: %f segundos\n", suma, end - start);

    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("Con reduction - Suma: %lld, Tiempo: %f segundos\n", suma, end - start);

    return 0;
}
```

Sin reduction la suma puede ser incorrecta debido a la condición de carrera. **Con reduction**, nos aseguramos una acumulación correcta y eficiente. Al compilar observamos el siguiente resultado.

```
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA-2/2 CUATRI/CAR/P5$ ./E2
Sin reduction - Suma: 51946402917, Tiempo: 0.015573 segundos
Con reduction - Suma: 500000500000, Tiempo: 0.000097 segundos
```

Ejercicio 3 Uso de *barrier* para sincronización

En la primera parte, varios hilos actualizan posiciones de un array en paralelo sin ninguna sincronización. Como resultado, algunos valores pueden sobrescribirse debido a la interferencia entre hilos, causando resultados incorrectos. Para corregir el problema, usamos **#pragma omp barrier**, lo que obliga a que todos los hilos completen la fase de escritura antes de pasar a la siguiente etapa.



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

#define N 10

int main() {
    int array[N] = {0};

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hilo %d escribiendo %d en la posición %d\n", id, id + 1, id);
        printf("Hilo %d leyendo la posición 5: %d\n", id, array[5]);
    }

    for (int i = 0; i < N; i++) array[i] = 0;
    printf("\n-----------------\n\n");

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        array[id] = id + 1;
        printf("Hilo %d escribió %d en la posición %d\n", id, id + 1, id);

        #pragma omp barrier
        printf("Hilo %d lee la posición 5: %d\n", id, array[5]);
    }

    return 0;
}
```

Sin barrier, los hilos pueden leer valores incorrectos del array porque algunos hilos pueden no haber terminado su escritura antes de la lectura. Con barrier, se garantiza que todos los hilos hayan escrito antes de cualquier lectura, asegurando consistencia en los datos. En conclusión, barrier es esencial cuando hay dependencias entre operaciones en paralelo.

Parte 2. Desarrollo Práctico Guiado

Ejercicio 1

El objetivo de este ejercicio es implementar un programa que sume los números del 1 al 1,000,000 utilizando OpenMP. Compararemos tres versiones:

1. **Versión secuencial:** Sin paralelización.
2. **Versión paralela sin reduction** (con condición de carrera).
3. **Versión paralela con reduction** (solución correcta).

En cada versión mediremos el tiempo de ejecución para evaluar la versión paralela optimizada.

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

#define N 1000000

int main() {
    long long suma = 0;
    double start, end;


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


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


    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;
}
```

Como aclaramos anteriormente, sin reduction, hay condición de carrera y el resultado es incorrecto. Con reduction, la suma es correcta y se logra una mejora en el rendimiento.

```
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA-2/2 CUATRI/CAR/P5/2PART
E$ ./E1
Secuencial - Suma: 500000500000, Tiempo: 0.001036 segundos
Paralela sin reduction - Suma: 25545169273, Tiempo: 0.015089 segundos
Paralela con reduction - Suma: 500000500000, Tiempo: 0.000088 segundos
```

Ejercicio 2

En este ejercicio se nos pide implementar un programa que calcule la suma de los cuadrados de los números del 1 al 1,000,000 utilizando tres estrategias de **schedule**.

schedule(static, chunk_size)

- Divide las iteraciones en bloques fijos de tamaño *chunk_size* y los asigna a los hilos de manera equitativa antes de iniciar la ejecución.
- Bueno para cargas de trabajo homogéneas donde todas las iteraciones toman un tiempo similar.
- Puede ser ineficiente si algunas iteraciones toman más tiempo que otras, ya que algunos hilos pueden quedar inactivos esperando a otros.

schedule(dynamic, chunk_size)

- Asigna iteraciones en bloques pequeños y las distribuye dinámicamente entre los hilos conforme terminan su trabajo.
- Bueno para cargas de trabajo desiguales, ya que los hilos más rápidos pueden recibir más iteraciones.
- Puede generar más overhead debido a la constante reasignación de iteraciones.

schedule(guided, chunk_size)

- Similar a dynamic, pero al inicio asigna bloques grandes y reduce el tamaño de los bloques conforme avanza la ejecución.
- Optimiza la eficiencia distribuyendo mejor el trabajo en iteraciones largas.
- Generalmente es un buen punto intermedio entre static y dynamic.

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

#define N 1000000

int main() {
    long long suma = 0;
    double start, end;

    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 del chunk

        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
        }
        end = omp_get_wtime();

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

    return 0;
}
```

Si las iteraciones toman tiempo similar, static suele ser la mejor opción. En cambio, si hay desbalance en la carga de trabajo, dynamic o guided funcionan mejor. Aunque Guided es útil cuando algunas iteraciones son más costosas que otras, pero no en exceso.

```
E$ ./E2
Schedule: static - Suma: 16866193336416, Tiempo: 0.000868 segundos
Schedule: dynamic - Suma: 16866193336416, Tiempo: 0.000168 segundos
Schedule: guided - Suma: 16866193336416, Tiempo: 0.000169 segundos
```

Ejercicio 3

En este ejercicio se nos pide implementar un programa donde varios hilos actualicen una variable compartida realizando tres versiones:

Sin protección (Condición de carrera)

- Varios hilos intentan modificar suma al mismo tiempo sin sincronización.
- Errores en la suma debido a accesos concurrentes no controlados.
- Más rápido pero incorrecto.

#pragma omp atomic (Solución eficiente)

- Asegura que la operación suma += 1; se realice sin interrupciones.
- Más eficiente que *critical*, ya que solo protege una operación puntual.

#pragma omp critical (Solución más costosa)

- Permite que solo un hilo a la vez modifique suma.
- Correcto, pero más lento debido a la sobrecarga de sincronización global.

```c
int main() {
    long long suma = 0;
    double start, end;


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


    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 segundos\n", suma, end - start);


    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 segundos\n", suma, end - start);

    return 0;
}
```

En resumen, *atomic* suele ser más rápido que *critical*, ya que solo protege la instrucción específica. Esto se debe a que *critical* introduce más sobrecarga, pues bloquea el acceso globalmente. Es importante recalcar que la versión más rápida es la que no tiene protección (pero esta es incorrecta).

```
Sin protección - Suma: 97175, Tiempo: 0.015840 segundos
Con atomic - Suma: 1000000, Tiempo: 0.022733 segundos
Con critical - Suma: 1000000, Tiempo: 0.226015 segundos
```

Ejercicio 4

En este ejercicio se nos pide aplicar sincronización avanzada para un problema. Utilizaremos atomic, critical y barrier dependiendo de la situación.

#pragma omp atomic → Para acumulaciones seguras

- Cada cocinero añade ingredientes al plato sin riesgo de condiciones de carrera.
- #pragma omp atomic evita sobrescrituras cuando múltiples hilos modifican plato simultáneamente.

#pragma omp critical → Para la fase especial del Chef

- Solo un hilo (el Gran Chef) puede acceder a la sección crítica donde se añaden ingredientes especiales.
- Protege operaciones más complejas que no pueden dividirse en operaciones atómicas.

#pragma omp barrier → Para sincronización global

- Evita que los cocineros continúen antes de que el Gran Chef termine su tarea especial.
- Asegura que los ingredientes se añadan en el orden correcto.

```c
#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) {  // Momento crítico del Gran Chef
                #pragma omp critical
                {
                    printf("Gran Chef %d preparando una sección especial...\n", id);
                    for (int j = 1; j <= 5; j++) {
                        plato += 1;
                        printf("Gran Chef %d añadió el ingrediente especial %d. Total: %d\n", id, j, plato);
                    }
                }
            } else {  // Trabajo normal de los cocineros
                int preparado = 0;
                for (int k = 0; k < 3; k++) {
                    preparado += 1;
                }

                #pragma omp atomic
                plato += preparado;

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

        }
        // Barrera para asegurarnos de que todos esperan antes de continuar
        #pragma omp barrier
    }

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

Podemos concluir que: sin atomic, se perderían ingredientes debido a la condición de carrera; sin critical, varios hilos podrían interferir con la fase del Gran Chef; y sin barrier,l os cocineros seguirían añadiendo ingredientes antes de que la fase especial termine, causando inconsistencias.

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

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.

Resolución

Identificación y Detección:
Cada iteración del bucle simula el reconocimiento de voz asignando aleatoriamente un idioma a un usuario y registrando coordenadas GPS. Se usa *schedule(dynamic)* para distribuir la carga de forma flexible y *reduction* para acumular conteos globales de usuarios por idioma.

Adaptación de Idioma:
Se compara el idioma detectado con un idioma "predominante" (definido de forma simple según la latitud) y, si no coinciden, se marca que se debe sugerir una adaptación. Esta operación se protege con una sección *critical* para garantizar que la lógica compleja se ejecute de forma exclusiva.

Sincronización y Generación del Mapa Global:
Una vez procesados todos los usuarios, se utiliza *barrier* para sincronizar a los hilos y luego se imprimen las estadísticas globales, que representan el "mapa" con la distribución de idiomas y el número de sugerencias de adaptación.

Santiago Álvarez Geanta Grupo 1

UA Universidad de Alicante

Universidad de Alicante