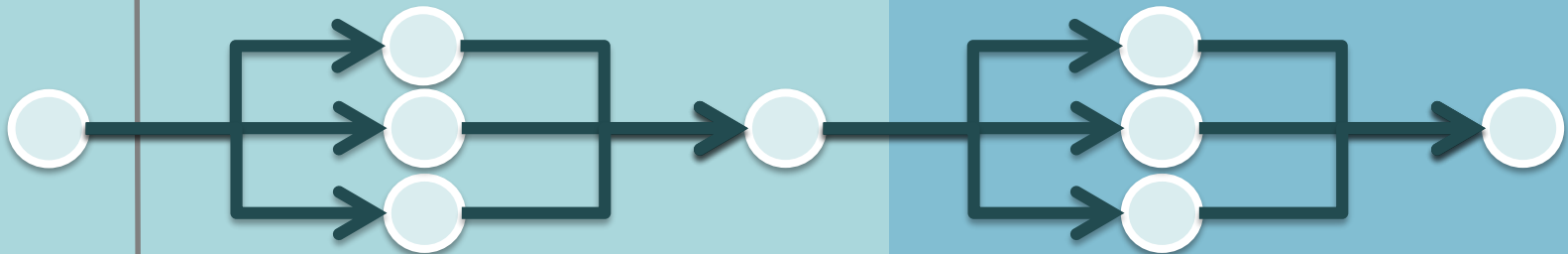


INTRODUCCIÓN A LA PROGRAMACIÓN PARALELA CON OPENMP (PARTE 2)



Computación de alto rendimiento

Índice

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Escalado

Presentación de
prácticas

1. Introducción a la Optimización en Programación Paralela
2. Sincronización y Gestión de Hilos en OpenMP
3. Balanceo de Carga y Distribución de Trabajo
4. Optimización de Accesos a Memoria y Reducción de Datos
5. Medición y Análisis de Rendimiento
6. Estrategias Avanzadas y Mejores Prácticas
7. Estrategias de escalado híbrido (OpenMP + MPI)

Introducción a la Optimización en Programación Paralela

Índice

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

1. Introducción a la Optimización en Programación Paralela

1.1 ¿Por qué optimizar código paralelo?

1.2 Principios fundamentales de optimización

1. Introducción a la Optimización en Programación Paralela

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

1.1 ¿Por qué optimizar código paralelo?

- **Sobrecarga por sincronización:** Tiempo extra que se pierde esperando en barreras o bloqueos.
- **Contención de recursos:** Accesos concurrentes a memoria compartida pueden ralentizar la ejecución.
- **Desbalanceo de carga:** Si algunos hilos tienen más trabajo que otros, se generan cuellos de botella.
- **Overhead de creación de hilos:** En algunos casos, crear demasiados hilos puede ser contraproducente.

1. Introducción a la Optimización en Programación Paralela

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

1.2 Principios fundamentales de optimización:

- Minimizar secciones críticas y sincronizaciones.
- Reducir el número de accesos a memoria compartida.
- Balancear la carga entre hilos.
- Evitar trabajo redundante.

Sincronización y Gestión de Hilos en OpenMP

Índice

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2. Sincronización y Gestión de Hilos en OpenMP

2.1 Condiciones de carrera y sincronización

2.2 Directivas de sincronización: critical, atomic y barrier

2.3 Comparación entre atomic y critical

2.4 Uso de reduction para sincronización segura

2.5 Manejo de excepciones en entornos paralelos

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

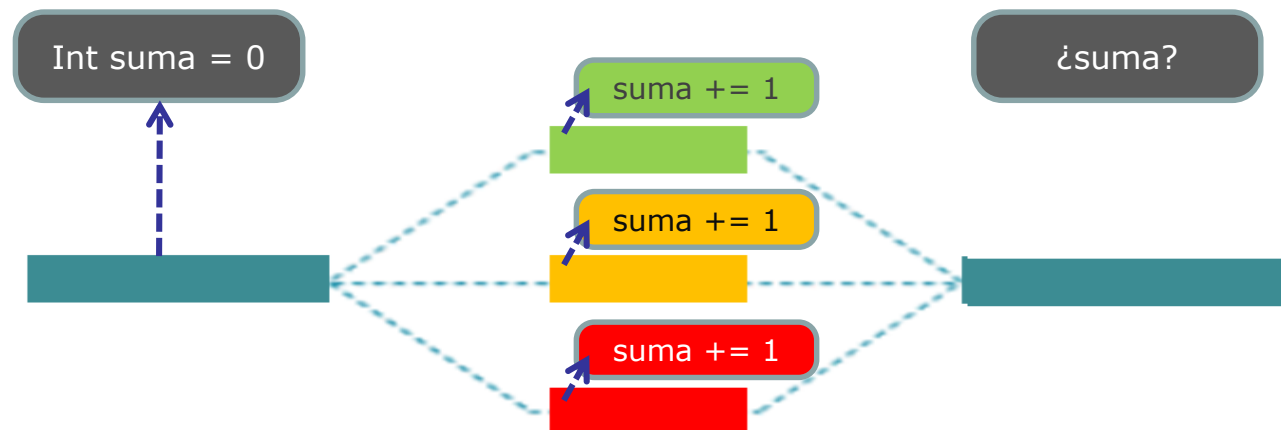
Rendimiento

Estrategias

Presentación de prácticas

2.1 Condiciones de carrera y sincronización

- Las condiciones de carrera ocurren cuando múltiples hilos acceden y modifican una variable compartida sin la debida sincronización
- Generan resultados impredecibles.



2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.2 Directivas de sincronización: critical, atomic y barrier

- **critical**: Protege secciones críticas complejas.
- **atomic**: Más eficiente para operaciones simples.
- **barrier**: Sincroniza todos los hilos en un punto específico.

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

Critical:

- Protege secciones críticas complejas.
- Garantiza que solo un hilo a la vez ejecute el bloque de código protegido.
- Ventaja: Seguro para operaciones complejas.
- Desventaja: Introduce sobrecarga significativa al hacer que los hilos esperen su turno.

```
#pragma omp critical  
{  
    contador += 1;  
}
```

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

atomic:

- Protege operaciones simples sobre variables compartidas (sumas, incrementos, etc.).
- Ventaja: Más eficiente que critical para operaciones simples.
- Desventaja: No admite operaciones complejas o múltiples líneas.

```
#pragma omp atomic  
contador += 1;
```

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

barrier:

- Fuerza a todos los hilos a detenerse en un punto hasta que todos hayan alcanzado esa barrera.
- Uso: Sincronizar distintas fases de ejecución.

```
#pragma omp barrier
```

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.3 Comparación entre atomic y critical

- Veamos dos escenarios en los que utilizaremos una de las dos opciones (atomic y crítico)
- En ambos escenarios varios hilos están incrementando un contador global.
- Queremos evitar condiciones de carrera, pero también optimizar el rendimiento.

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de prácticas

2.3 Comparación entre atomic y critical

- Opción 1 – Uso de critical:

```
#include <stdio.h>
#include <omp.h>
int main() {
    int contador = 0;
    #pragma omp parallel for
    for (int i = 0; i < 1000000; i++) {
        #pragma omp critical
        {
            contador += 1;
        }
    }
    printf("Contador final: %d\n", contador);
    return 0;
}
```

Resultado correcto, pero **ineficiente** debido a que solo un hilo puede acceder a la sección crítica a la vez.

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.3 Comparación entre atomic y critical

- Opción 2: Uso de atomic

```
#include <stdio.h>
#include <omp.h>
int main() {
    int contador = 0;
    #pragma omp parallel for
    for (int i = 0; i < 1000000; i++) {
        #pragma omp atomic
        contador += 1; // Operación protegida de forma más eficiente
    }
    printf("Contador final: %d\n", contador);
    return 0;
}
```

Resultado correcto y **más eficiente que critical**.

¿Por qué?

- atomic protege operaciones simples como incrementos y es menos costoso en términos de sincronización.
- Evita la sobrecarga del bloqueo completo que introduce critical.

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.3 Comparación entre atomic y critical

- ¿Cuándo usar critical sobre atomic?
- Recomendamos usar **atomic** para **operaciones simples** sobre variables compartidas (suma, resta, multiplicación).
- Recomendamos usar **critical** cuando la operación involucra **múltiples líneas o es más compleja (por ejemplo, actualizar múltiples variables o escribir en un archivo)**.

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

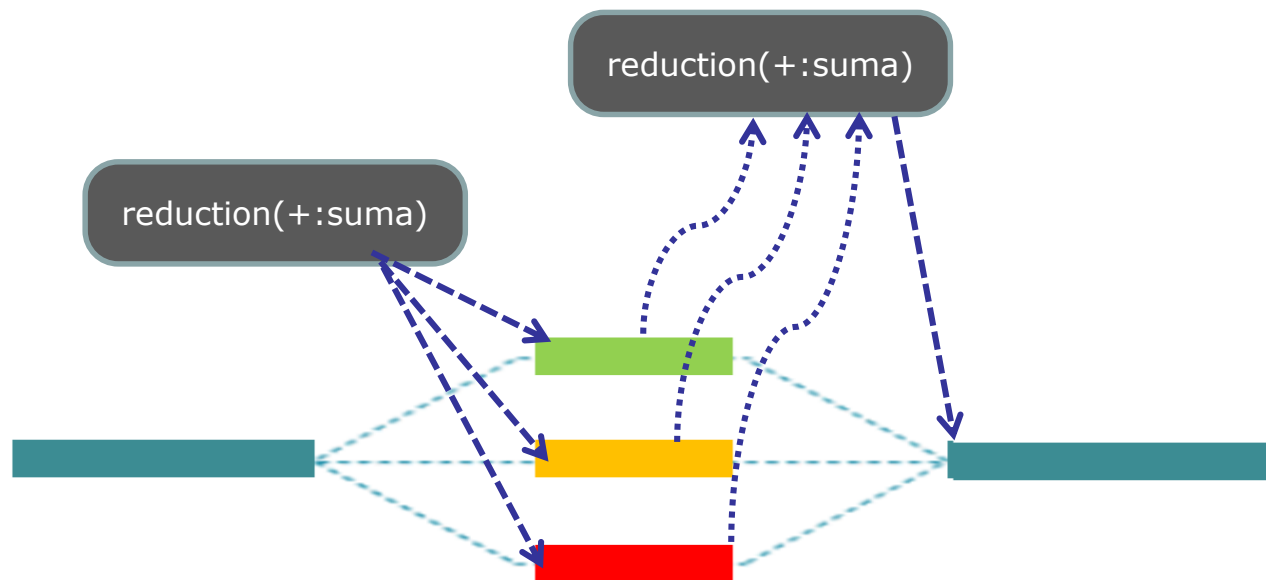
Rendimiento

Estrategias

Presentación de prácticas

2.4 Uso de reduction para sincronización segura

- Cada hilo obtiene su propia copia privada de la variable.



2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.4 Uso de reduction para sincronización segura

- La cláusula reduction permite acumular resultados de manera segura evitando condiciones de carrera
- Es más eficiente que critical o atomic para operaciones acumulativas.
- Al finalizar la región paralela, los resultados se combinan utilizando el operador especificado.

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.4 Uso de reduction para sincronización segura

- Ejemplo de suma paralela con reduction:

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

int main() {
    int N = 1000000;
    long long suma = 0;

    #pragma omp parallel for reduction(+:suma)
    for (int i = 1; i <= N; i++) {
        suma += i;
    }

    printf("Suma total: %lld\n", suma);
    return 0;
}
```

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.4 Uso de reduction para sincronización segura

- Como podemos ver en el código.
 - Cada hilo tiene su propia copia de suma para evitar interferencias.
 - Al finalizar el bucle, OpenMP combina los resultados.
 - Este enfoque elimina condiciones de carrera y es más eficiente que usar atomic o critical (los veremos a continuación).

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

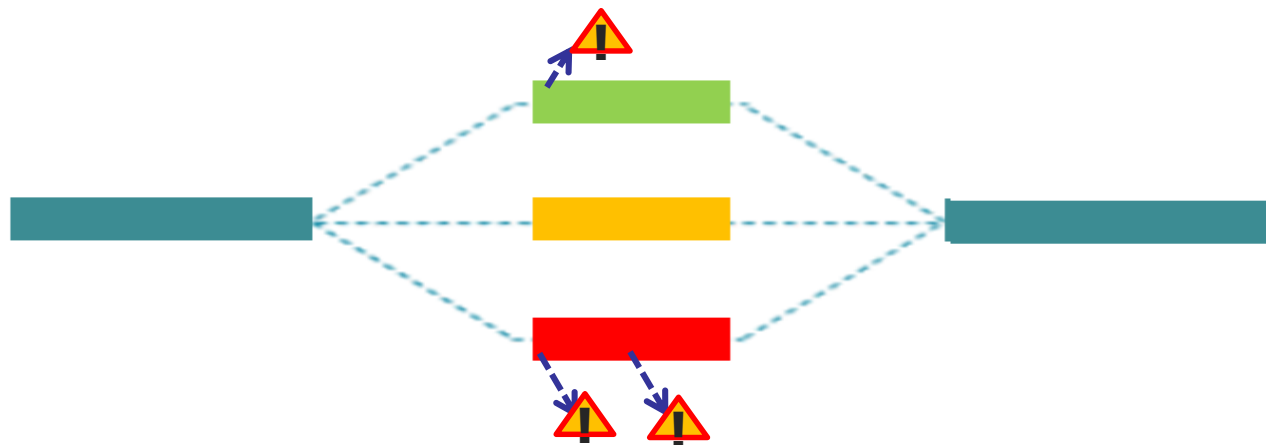
Rendimiento

Estrategias

Presentación de
prácticas

2.5 Manejo de excepciones en entornos paralelos

- En entornos paralelos con OpenMP, el manejo de excepciones puede ser complicado
- Múltiples hilos podrían generar errores al mismo tiempo.
- OpenMP no proporciona una estructura formal de manejo de excepciones
- Debemos seguir el uso de buenas prácticas para asegurar una ejecución controlada.



2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.5 Manejo de excepciones en entornos paralelos

- Buenas prácticas para manejo de errores:
 - Utilizar **variables compartidas** y banderas (**Flags**) de error para notificar fallos entre hilos.
 - **Proteger secciones críticas** que podrían generar excepciones usando directivas como **critical**.
 - Evitar lanzar excepciones desde dentro de regiones paralelas en C/C++, ya que esto puede generar comportamientos indefinidos.
- Ejemplo:

2. Sincronización y Gestión de Hilos en OpenMP

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

2.5 Manejo de excepciones en entornos paralelos

- Ejemplo de manejo de errores usando una bandera:

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

int main() {
    int error_flag = 0;

    #pragma omp parallel for shared(error_flag)
    for (int i = 0; i < 10; i++) {
        if (i == 5) { // Simulación de un error
            #pragma omp critical
            {
                printf("Error en el hilo %d en la iteración %d\n", omp_get_thread_num(), i);
                error_flag = 1;
            }
        }
    }

    if (error_flag) {
        printf("Ejecución completada con errores.\n");
    } else {
        printf("Ejecución completada sin errores.\n");
    }

    return 0;
}
```


Balanceo de Carga y Distribución de Trabajo

Índice

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

3. Balanceo de Carga y Distribución de Trabajo

3.1 Distribución eficiente de carga con `schedule()`

3.2 ¿Por qué usar `schedule(dynamic)`?

3.3 Paralelización Anidada (Nested Parallelism)

3. Balanceo de Carga y Distribución de Trabajo

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

3.1 Distribución eficiente de carga con schedule()

- Problema del desbalanceo de carga:
 - Si ciertos hilos tienen significativamente **más trabajo que otros**, los hilos más rápidos quedarán ociosos mientras esperan a los más lentos
- Solución:
 - Utilizar **schedule** para que los hilos tomen nuevas tareas cuando terminan.
 - **Reducir** el tamaño del "**chunk**" para mejorar el equilibrio (a costa de mayor overhead de sincronización).

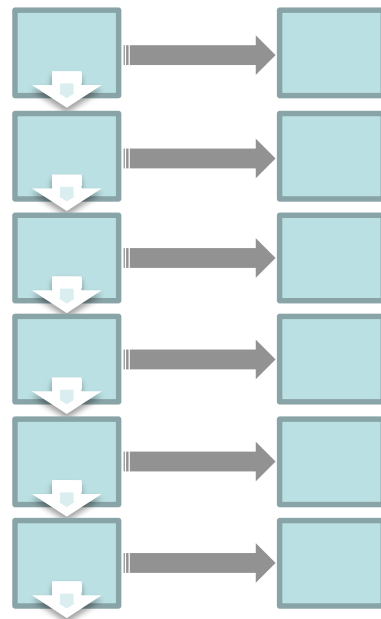
3. Balanceo de Carga y Distribución de Trabajo

3.1 Distribución eficiente de carga con schedule()

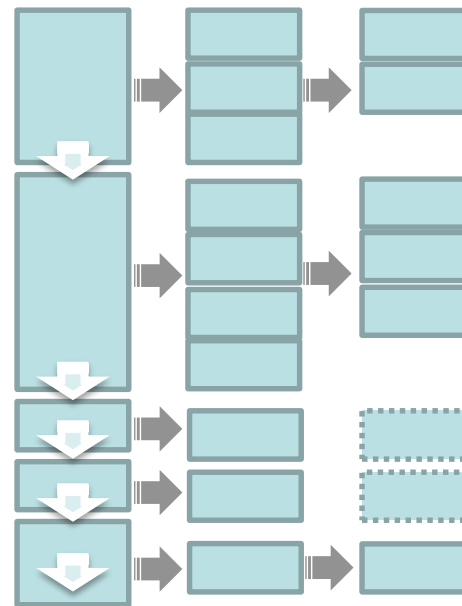
static: Cargas uniformes y predecibles.

dynamic: Cargas desbalanceadas o variables.

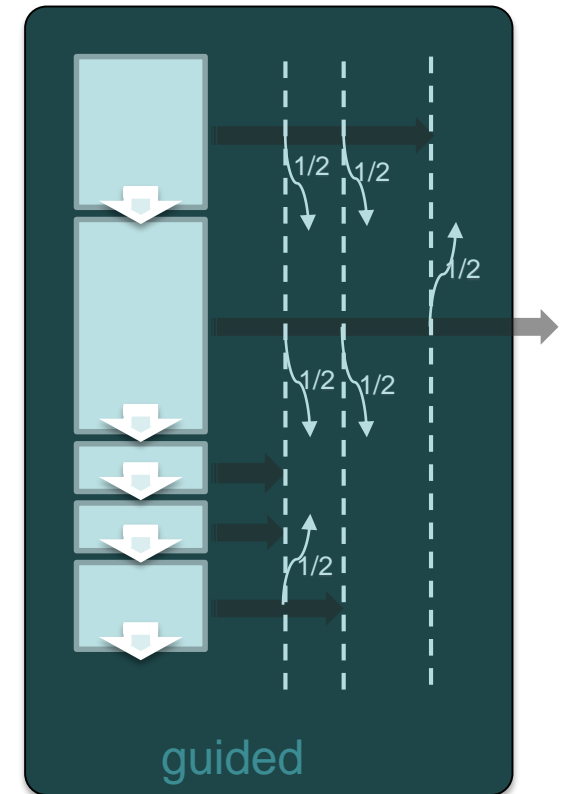
guided: Cargas pesadas e impredecibles.



static



dynamic



guided

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

3. Balanceo de Carga y Distribución de Trabajo

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

3.2 ¿Por qué usar `schedule(dynamic)`?

- `schedule(static)` puede provocar un desbalance de carga.
- `schedule(dynamic)` mejora el equilibrio
- Ejemplo con **`schedule(static)`**:

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

int main() {
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < 8; i++) {
        printf("Hilo %d procesando iteración %d\n", omp_get_thread_num(), i);
        sleep(i % 3); // Simula tareas con duraciones variables
    }
    return 0;
}
```

- **Problema:**

- Las tareas más largas ralentizan el hilo al que se asignaron inicialmente, mientras que otros hilos terminan antes y **quedan ociosos**.

3. Balanceo de Carga y Distribución de Trabajo

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

3.2 ¿Por qué usar `schedule(dynamic)`?

- Solución: Uso de **`schedule(dynamic)`**, Ejemplo:

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

int main() {
    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < 8; i++) {
        printf("Hilo %d procesando iteración %d\n", omp_get_thread_num(), i);
        sleep(i % 3); // Simula tareas con duraciones variables
    }
    return 0;
}
```

- Ventajas de `schedule(dynamic)`:
 - Los hilos toman nuevas iteraciones dinámicamente al finalizar su trabajo actual.
 - Mejora el balance de carga en escenarios con tareas heterogéneas.
 - Reduce el tiempo total de ejecución.

3. Balanceo de Carga y Distribución de Trabajo

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

3.3 Paralelización Anidada (Nested Parallelism)

- OpenMP permite la creación de regiones **paralelas** dentro de otras regiones paralelas: **paralelización anidada**
- Aunque está desactivada por defecto, se puede **habilitar** usando la función **omp_set_nested(1)**
- **Ventajas:**
 - Permite explotar **más niveles de paralelismo**, especialmente en aplicaciones complejas.
 - Útil en problemas jerárquicos, como simulaciones multi-escala o algoritmos de matrices bloqueadas.
- **Desventajas:**
 - Puede incrementar la sobrecarga por gestión de hilos.
 - Requiere un buen balanceo de carga para evitar ineficiencias.

3. Balanceo de Carga y Distribución de Trabajo

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

3.3 Paralelización Anidada (Nested Parallelism)

- Ejemplo de paralelización anidada:

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

int main() {
    omp_set_nested(1); // Habilitar paralelización anidada

    #pragma omp parallel num_threads(2)
    {
        printf("Nivel 1: Hilo %d\n", omp_get_thread_num());

        #pragma omp parallel num_threads(2)
        {
            printf("  Nivel 2: Hilo %d\n", omp_get_thread_num());
        }
    }
    return 0;
}
```

```
Nivel 1: Hilo 0
  Nivel 2: Hilo 0
  Nivel 2: Hilo 1
Nivel 1: Hilo 1
  Nivel 2: Hilo 0
  Nivel 2: Hilo 1
```


Optimización de Accesos a Memoria y Reducción de Datos

Índice

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

4. Optimización de Accesos a Memoria y Reducción de Datos

4.1 Minimizar accesos a memoria compartida

4.2 Alineación de datos y uso eficiente de cachés

4.3 Variables privadas: private, firstprivate, lastprivate

4.4 Inicialización con copyin

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

4.1 Minimizar accesos a memoria compartida

- El acceso frecuente a memoria compartida por múltiples hilos introduce latencia
- Puede generar **contention**, afectando la eficiencia general
- Ejemplo:

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de prácticas

4.1 Minimizar accesos a memoria compartida

- Ejemplo con Accesos Frecuentes a Memoria Compartida

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

int main() {
    int data[10000];
    int suma = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10000; i++) {
        #pragma omp atomic
        suma += data[i]; // Cada iteración accede a memoria compartida
    }

    printf("Suma total: %d\n", suma);
    return 0;
}
```

- Problema:
 - Cada hilo lee y actualiza la memoria compartida (suma) en cada iteración, generando cuello de botella.

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de prácticas

4.1 Minimizar accesos a memoria compartida

- Solución: Uso de **Variables Locales** por Hilo:

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

int main() {
    int data[10000];
    int suma = 0;

    // Utiliza reduction para evitar accesos frecuentes a suma
    #pragma omp parallel for reduction(+:suma)
    for (int i = 0; i < 10000; i++) {
        suma += data[i]; // Acumulación local por hilo
    }

    printf("Suma total: %d\n", suma);
    return 0;
}
```

- Beneficios:
 - Cada hilo acumula la suma en su propia variable local.
 - Los resultados se combinan al final, minimizando accesos a memoria compartida.
- **Resultado:** Mejora significativa del rendimiento.

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

4.1 Minimizar accesos a memoria compartida

Consejos Generales para Reducir Accesos a Memoria:

1. Usar **variables locales** dentro de regiones paralelas siempre que sea posible.
2. Agrupar lecturas/escrituras para minimizar los accesos dispersos.
3. Evitar variables compartidas innecesarias.
4. Usar técnicas de **reducción** para consolidar resultados de manera eficiente.

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

4.2 Alineación de datos y uso eficiente de cachés

- La forma en la que accedemos a memoria también afecta al rendimiento de aplicaciones paralelas
 - Mejor: usar accesos a memorias locales (caché)
 - Alinear estructuras de datos a los límites de caché para evitar penalizaciones de acceso
 - Evitar falsas comparticiones: varios hilos acceden a diferentes variables almacenadas en la misma línea de caché, generando conflictos innecesarios.

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de prácticas

4.2 Alineación de datos y uso eficiente de cachés

- Ejemplo de falsa compartición:

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

#define N 1000000

typedef struct {
    double x;
    double y; // Sin separación, puede causar falsa compartición
} Point;

int main() {
    Point points[4];

    #pragma omp parallel for
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < N; j++) {
            points[i].x += j * 0.1;
        }
    }

    return 0;
}
```


4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de prácticas

4.2 Alineación de datos y uso eficiente de cachés

- **Solución:** Añadir relleno para evitar que dos variables usadas por distintos hilos compartan la misma línea de caché:

```
typedef struct {  
    double x;  
    char padding[64]; // Relleno para evitar falsa compartición  
    double y;  
} Point;
```

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

4.3 Variables privadas: `private`, `firstprivate`, `lastprivate`

- **private:** Cada hilo tiene su propia copia no inicializada.
- **firstprivate:** Inicializa las copias privadas con el valor original.
- **lastprivate:** Copia el valor de la última iteración al ámbito global.
- Ejemplo:

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de prácticas

4.3 Variables privadas: private, firstprivate, lastprivate

- Ejemplo:

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

int main() {
    int x = 10;

    #pragma omp parallel for private(x) firstprivate(x) lastprivate(x)
    for (int i = 0; i < 5; i++) {
        x += i;
        printf("Hilo %d - Valor de x: %d\n", omp_get_thread_num(), x);
    }

    printf("Valor final de x después del bucle: %d\n", x);
    return 0;
}
```

- Resultado esperado:
 - Cada hilo trabaja con su propia copia de x.
 - La última iteración actualiza el valor global de x.

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

4.4 Inicialización con la clausula copyin

- **inicializa** las variables declaradas como **threadprivate** en cada hilo con el valor de la variable global antes de entrar en una región paralela
- ¿Cuándo usar copyin?
 - Cuando se tiene una **variable global** declarada como **threadprivate**.
 - Para asegurar que cada hilo comience con el mismo valor inicial antes de ejecutar la región paralela.
- **Ejemplo básico:**

4. Optimización de Accesos a Memoria y Reducción de Datos

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de prácticas

4.4 Inicialización con la clausula copyin

- **Ejemplo básico:**

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

// Variable global
int global_config = 42;

// Declarar como threadprivate
#pragma omp threadprivate(global_config)

int main() {
    // Modificar antes de la región paralela
    global_config = 100;

    // Región paralela usando copyin
    #pragma omp parallel copyin(global_config)
    {
        printf("Hilo %d - Config inicial: %d\n", omp_get_thread_num(), global_config);
    }

    return 0;
}
```

Hilo 0 - Config inicial: 100
Hilo 1 - Config inicial: 100
Hilo 2 - Config inicial: 100
Hilo 3 - Config inicial: 100

Medición y Análisis de Rendimiento

Índice

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5. Medición y Análisis de Rendimiento

5.1 Uso de `omp_get_wtime()`

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

5.3 Ley de Amdahl y su aplicación

5.5 Depuración y perfiles de rendimiento

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.1 Uso de omp_get_wtime()

- Se utiliza para medir el tiempo transcurrido entre dos puntos del código (recordatorio).

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

int main() {
    double start = omp_get_wtime();

    #pragma omp parallel for
    for (int i = 0; i < 1000000; i++) {
        // Simulación de trabajo pesado
    }

    double end = omp_get_wtime();
    printf("Tiempo de ejecución: %f segundos\n", end - start);
    return 0;
}
```


5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

- Las métricas de rendimiento permiten evaluar y cuantificar la eficiencia de un programa paralelo en comparación con su versión secuencial
 - **Speed-up:** Mide la mejora de rendimiento en paralelo.
 - **Eficiencia:** Evalúa el aprovechamiento de los hilos.
 - **Escalabilidad:** Analiza el rendimiento al variar el número de hilos o el tamaño del problema.

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

- **Speed-up (S)**: Mide la mejora obtenida al ejecutar un programa en paralelo en lugar de secuencialmente.

$$S = \frac{T_{\text{secuencial}}}{T_{\text{paralelo}}}$$

- Un Speed-up **ideal** sería igual al número de hilos utilizados (Speed-up lineal), es decir, **S = N**.
- En la práctica, alcanzar un Speed-up perfecto es difícil debido a sincronización

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

Ejemplo práctico – Cálculo de Speed-up:

Supongamos que tenemos un programa que ejecuta una operación de suma sobre un arreglo grande.

- **Versión secuencial:**
 - Tiempo de ejecución = **20 segundos**
- **Versión paralela con 4 hilos:**
 - Tiempo de ejecución = **6 segundos**

Cálculo del Speed-up:

$$S = \frac{20}{6} \approx 3.33$$

Interpretación:

- El programa paralelo es **3.33 veces más rápido** que la versión secuencial al usar 4 hilos.
- Aunque idealmente el Speed-up sería **4**, un resultado de **3.33** es bastante eficiente.

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

- **Eficiencia:** Evalúa qué tan bien se utilizan los hilos disponibles.

$$E = \frac{S}{N}$$

- **S** = Speed-up obtenido
- **N** = Número de hilos
- Eficiencia ideal: $E = 1$ (100%)
- En la práctica, valores entre 0.7 y 0.9 se consideran buenos.

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

Ejemplo práctico – Cálculo de Eficiencia:

- **Speed-up obtenido (S):** 3.33
- **Número de hilos (N):** 4

$$E = \frac{3.33}{4} = 0.833$$

- **Interpretación:**

La eficiencia es **83.3%**, lo cual indica un excelente aprovechamiento de los recursos paralelos.

¿Qué afecta la eficiencia?

1. **Sincronización excesiva:** Barreras y secciones críticas disminuyen la eficiencia.
2. **Partes no paralelizables:** Según la **Ley de Amdahl**, la fracción de código que no se puede paralelizar limita el Speed-up.
3. **Sobrecarga de hilos:** Demasiados hilos para pocas tareas puede disminuir la eficiencia debido al overhead.

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

- **Escalabilidad:** Analiza **cómo se comporta** el rendimiento de un programa paralelo **cuando se modifican ciertos parámetros**, como el número de hilos o el tamaño del problema.
- Su objetivo es entender si agregar más recursos computacionales realmente mejora el rendimiento y hasta qué punto.
- Se trata de poner a prueba el sistema y verificar si tiene una escalabilidad **fuerte** o **débil**

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

- **Escalabilidad Fuerte** → "¿Qué pasa si aumento los hilos para resolver el mismo problema más rápido?"
 - El tamaño del problema es constante.
 - Evalúa cómo mejora el tiempo de ejecución al agregar más hilos.
 - Útil cuando el objetivo es acortar tiempos de ejecución.
- **Escalabilidad Débil** → "¿Qué pasa si aumento los hilos y al mismo tiempo aumento la carga de trabajo para que cada hilo tenga la misma cantidad de trabajo?"
 - El tamaño del problema crece junto con los hilos.
 - Evalúa si el sistema puede manejar más trabajo manteniendo el mismo tiempo de ejecución.
 - Útil cuando el objetivo es procesar más datos sin que aumente el tiempo.

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.2 Métricas clave de rendimiento: Speed-up, Eficiencia, Escalabilidad

- Cómo los calculo y qué analizan:

| Métrica | Escalabilidad Fuerte | Escalabilidad Débil |
|-------------------|--|--|
| Objetivo | Resolver el mismo problema más rápido | Resolver problemas más grandes en el mismo tiempo |
| Cálculo principal | Speed-up: $S = \frac{T_1}{T_N}$ | Eficiencia Débil: $E_w = \frac{T_1}{T_N}$ |
| Ideal | Speed-up lineal ($S = N$) | Tiempo de ejecución constante |
| Qué analiza | Mejora de rendimiento con más hilos | Capacidad de manejar cargas crecientes |

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.3 Ley de Amdahl y su aplicación (recordatorio)

- La aceleración máxima que se puede obtener mediante paralelización está limitada por la fracción del código que no se puede paralelizar.

$$Speed - Up = \frac{1}{(1 - p) + \frac{p}{N}}$$

- Donde:
 - p = fracción paralelizable
 - N = número de hilos
- Recordar que:
 - Altos Speed-up solo se logran cuando la fracción paralelizable es muy alta (**$p \approx 0.9$ o más**).
 - Agregar más hilos no siempre mejora el rendimiento significativamente.
 - Es esencial optimizar el código secuencial antes de escalar.

5. Medición y Análisis de Rendimiento

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

5.4 Depuración y perfiles de rendimiento

- Optimizar el rendimiento no solo requiere escribir buen código paralelo, sino también herramientas para medir y depurar posibles problemas.
- Herramientas de perfilado recomendadas:
 - **Intel VTune Profiler:** Permite identificar cuellos de botella, problemas de sincronización y patrones de acceso a memoria.
 - **gprof:** Herramienta simple para generar perfiles de rendimiento de programas.
 - **Valgrind (con Callgrind):** Útil para obtener información sobre la utilización de la caché y el consumo de memoria.

Estrategias Avanzadas y Mejores Prácticas

Índice

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

6. Estrategias Avanzadas y Mejores Prácticas

6.1 Resumen de estrategias para optimización

6.2 Mejores prácticas para código paralelo eficiente

6. Estrategias Avanzadas y Mejores Prácticas

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

6.1 Resumen de estrategias para optimización

- **Minimizar sincronización innecesaria:** Reducir el uso de secciones críticas y barreras que puedan causar bloqueos y ralentización.
- **Utilizar directivas eficientes:** Emplear `atomic` para operaciones simples y `reduction` para acumulaciones seguras en lugar de `critical`.
- **Balancear la carga de trabajo efectivamente:** Implementar `schedule(dynamic)` o `schedule(guided)` para distribuir tareas desiguales entre hilos.
- **Optimizar accesos a memoria compartida:** Usar variables locales y técnicas de reducción para disminuir la contención.

6. Estrategias Avanzadas y Mejores Prácticas

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

6.1 Resumen de estrategias para optimización

- **Ajustar el tamaño de los bloques de trabajo:** Experimentar con diferentes configuraciones de chunk size en `schedule()` para encontrar el equilibrio óptimo entre sobrecarga y balanceo.
- **Aplicar técnicas de paralelización granular:** Dividir las tareas en unidades más pequeñas para aprovechar mejor los recursos.
- **Medir y analizar el rendimiento continuamente:** Utilizar herramientas como `omp_get_wtime()` y métricas clave (Speed-up, Eficiencia) para identificar cuellos de botella y áreas de mejora.

6. Estrategias Avanzadas y Mejores Prácticas

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

6.2 Mejores prácticas para código paralelo eficiente

- **Diseño modular y claro:** Estructurar el código en módulos bien definidos facilita la paralelización y el mantenimiento.
- **Evitar secciones críticas extensas:** Limitar el tamaño de las secciones críticas reduce la contención y mejora el rendimiento.
- **Uso de variables locales:** Siempre que sea posible, emplear variables locales para minimizar accesos a memoria compartida.
- **Análisis y optimización continua:** Realizar pruebas frecuentes y utilizar herramientas de análisis de rendimiento para detectar cuellos de botella.

6. Estrategias Avanzadas y Mejores Prácticas

Índice

Introducción

Sincronización

Balanceo

Acceso Memoria

Rendimiento

Estrategias

Presentación de
prácticas

6.2 Mejores prácticas para código paralelo eficiente

- **Escalabilidad como prioridad:** Diseñar el código pensando en la escalabilidad para aprovechar futuros aumentos de recursos.
- **Documentación clara y completa:** Documentar todas las decisiones de paralelización y optimización facilita futuras modificaciones y el trabajo en equipo.
- **Manejo adecuado de excepciones y errores:** Implementar mecanismos robustos para manejar fallos durante la ejecución paralela.
- **Revisión de código y buenas prácticas:** Realizar revisiones periódicas del código para asegurar el cumplimiento de las mejores prácticas y detectar posibles mejoras.

Preguntas...