1. ¿Qué es OpenMP? Modelo fork–join y memoria compartida

OpenMP (Open Multi-Processing) es una API de alto nivel para programación paralela en memoria compartida. Se basa en el modelo **fork–join**:

1. **Fork**: el hilo maestro (“master”) encuentra una directiva `#pragma omp parallel` y crea un equipo (“team”) de hilos.
2. **Trabajo concurrente**: dentro de la región paralela, todos los hilos pueden ejecutar secciones de código simultáneamente, compartiendo el mismo espacio de direcciones.
3. **Join**: al finalizar la región paralela, todos los hilos menos el maestro terminan y éste continúa la ejecución en serie.

Bajo este modelo:

- **Variables compartidas** (`shared`) residen en un único lugar de memoria accesible por todos los hilos.
- **Variables privadas** (`private`) existen por separado en cada hilo, evitando condiciones de carrera.

2. Principales directivas de OpenMP

OpenMP está basado en **directivas** (pragmas en C/C++ y comentarios especiales en Fortran). Las más usadas son:

- `#pragma omp parallel`
  Define una región paralela “fork” en la que se ejecuta el bloque asociado con múltiples hilos.

- `#pragma omp for` (o `do` en Fortran)
  Distribuye las iteraciones de un bucle entre los hilos del equipo.

- `#pragma omp parallel for`
  Combina ambas: crea el equipo y reparte el bucle en una sola directiva.

- `#pragma omp sections`
  Divide el trabajo en secciones independientes, cada una ejecutada por un hilo distinto.

- `#pragma omp single`
  Solo un único hilo ejecuta el bloque asociado, los demás esperan en un punto de **barrier** implícito.

- `#pragma omp master`
  Solo el hilo maestro ejecuta el bloque; los demás hilos no esperan (no hay barrier).

- Sincronización y consistencia:

  - `#pragma omp barrier`
    Todos los hilos sincronizan y esperan en este punto.
  - `#pragma omp atomic`
    Actualización atómica de una variable compartida.
  - `#pragma omp critical`
    Región exclusiva para un único hilo a la vez.
  - `#pragma omp flush`
    Forzar la consistencia de las vistas de memoria entre hilos. □cite□turn0file4□

> **Nota:** No todas las directivas admiten cláusulas (por ejemplo, `master` no acepta cláusulas de ámbito de variable).

---

2. Cláusulas de control de hilos: `if` y `num_threads`

`if`

- **Qué hace**: decide en tiempo de ejecución si la región paralela se "forkea" (se crea el equipo de hilos) o se ejecuta secuencialmente por el hilo maestro.
- **Sintaxis**:

```
#pragma omp parallel if(condición)
{ /* … */ }
```

- **Semántica**:
  - Si `condición` es verdadera, se crea un equipo de hilos y se ejecuta el bloque en paralelo.
  - Si es falsa, sólo el hilo maestro ejecuta el bloque (sin fork) □cite□turn1file0□.
- **Dónde se puede usar**: en directivas que generan una región paralela (`parallel`, `parallel for`, `parallel sections`, `parallel workshare`) □cite□turn1file1□.

> **Ejemplo**
>
> ```
> if (N > 20)
>   #pragma omp parallel if(N > 20)
>   {
>     // Se crea el team sólo si N>20
>   }
> else
>   // Se ejecuta secuencialmente
>   { … }
> ```
>
> En este caso, con N≤20 no hay fork y todo corre en un único hilo □cite□turn1file1□.

`num_threads`

- **Qué hace**: fuerza el número de hilos que se usarán en la región paralela (si los recursos lo permiten).
- **Sintaxis**:

```
#pragma omp parallel num_threads(n_hebras)
{ /* … */ }
```

- **Semántica**:
  - Intenta arrancar exactamente `n_hebras` hilos.

    - Si no hay suficientes recursos, el runtime puede crear menos hilos.
- **Dónde se puede usar**: en las mismas directivas que admiten `if` (por ejemplo, `parallel`, `parallel for`, etc.) □cite□turn1file1□.

> **Ejemplo combinado**
>
> ```
> #pragma omp parallel if(N>20) num_threads(8)
> for (i = 0; i < N; ++i) {
>   // … trabajo paralelo con hasta 8 hilos si N>20
> }
> ```
>
> Aquí, si `N>20` el bucle se reparte entre (hasta) 8 hilos; si no, se ejecuta en serie □cite□turn1file1□.

---

3. Cláusulas de compartición de datos

En OpenMP, el comportamiento de las variables dentro de una región paralela se ajusta mediante cláusulas de compartición de datos. Estas cláusulas se usan con directivas como `parallel`, `for`, `sections`, etc. □cite□turn1file0□.

Tipo de cláusula	Ejemplos	Función principal
Ámbito de variables	`shared`, `private`, `firstprivate`, `lastprivate`, `default(none)`	Controlan si cada hilo ve una única copia o una copia privada y cómo se inicializa y finaliza.

`shared(list)`

- **Qué hace**: todos los hilos comparten la **misma** instancia de las variables en `list`.
- **Precaución**: si varios hilos escriben concurrentemente sin sincronización, se producen **condiciones de carrera**.
- **Ejemplo**:

  ```
  int a[7];
  // … inicializar a …
  #pragma omp parallel for shared(a)
  for (i = 0; i < 7; ++i)
    a[i] += i;
  // Todas las hebras acceden y modifican el mismo array a
  ```

  □cite□turn1file13□

`private(list)`

- **Qué hace**: cada hilo obtiene una copia **no inicializada** de cada variable de `list`. El valor al salir de la región es indefinido, incluso si la variable existía antes.
- **Ámbito predeterminado**: los índices de bucle en `for` son `private` por defecto.
- **Ejemplo**:

```
#pragma omp parallel private(suma)
{
  suma = 0;
  #pragma omp for
  for (i = 0; i < n; ++i)
    suma += a[i];
  // suma es local a cada hebra
}
```

□cite□turn1file14□

`firstprivate(list)`

- **Qué hace**: como `private`, pero **inicializa** cada copia con el valor que tenía fuera de la región paralela.
- **Útil cuando** quieres que cada hilo empiece con el mismo valor base.
- **Ejemplo**:

```
int suma = 0;
#pragma omp parallel for firstprivate(suma)
for (i = 0; i < n; ++i) {
  suma += a[i];
  printf("hebra %d suma=%d\n", omp_get_thread_num(), suma);
}
// Fuera, 'suma' no cambia
```

□cite□turn1file2□

`lastprivate(list)`

- **Qué hace**: como `private`, pero al **salir** de la región paralela copia el valor de la **última** iteración (o sección) a la variable original.
- **Útil si** necesitas conservar, por ejemplo, el índice o resultado de la última iteración.
- **Ejemplo**:

```
int v;
#pragma omp parallel for lastprivate(v)
for (i = 0; i < 7; ++i) {
  v = a[i];
```

```
}
// Al final, 'v' vale a[6]
```

□cite□turn1file9□

`default(none)`

- **Qué hace**: obliga a **declarar** explícitamente el ámbito de **todas** las variables usadas en la construcción (`shared`, `private`, etc.).
- **Ventaja**: previene olvidos que puedan causar condiciones de carrera.
- **Restricción**: sólo puede aparecer una vez.
- **Ejemplo**:

```
int n, a[N];
#pragma omp parallel for default(none) shared(a,n) private(i)
for (i = 0; i < n; ++i)
  a[i] += i;
```

□cite□turn1file4□

Excepciones comunes

- **Índices de bucle** (`for`): son `private` por defecto □cite□turn1file14□.
- **Variables** `static` (globales o estáticas): permanecen `shared` a menos que se use `threadprivate`.

---

4. Cláusulas de reducción y de copia de valores

`reduction(op: lista)`

- **Qué hace**: cada hebra tiene su propia copia privada de la variable, inicializada al valor neutro de la operación (`0` para `+`, `1` para `*`, etc.). Al salir de la región paralela, todas esas copias se combinan **atómicamente** (como una operación colectiva "todos a uno") en la variable original.
- **Sintaxis**:

```
#pragma omp parallel for reduction(+: suma)
for (i = 0; i < n; ++i)
  suma += a[i];
```

- **Operadores soportados** (`v3.0`): `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `min`, `max`, ... □cite□turn2file12□turn2file6□
- **Ventaja**: evita condiciones de carrera y te olvida de `atomic` o `critical`.

`copyin(lista)`

- **Qué hace**: inicializa las variables marcadas con `threadprivate` en cada hebra con el valor que tenían en el hilo maestro **antes** de entrar en la región paralela.
- **Para qué sirve**: cuando has declarado variables `threadprivate` a nivel global (por ejemplo, para mantener estado en subrutinas), `copyin` propaga ese estado inicial a todas las hebras.
- **Sintaxis**:

```
#pragma omp parallel copyin(mi_estado_threadprivate)
{
  // aquí cada hebra ve la copia inicializada de mi_estado_threadprivate
}
```

- **Nota**: sólo aplica a variables globales o estáticas marcadas previamente con `#pragma omp threadprivate` □cite□turn2file0□.

`copyprivate(lista)`

- **Qué hace**: en una región `single`, tras ejecutarla, la **unica** hebra que la ejecutó "difunde" el valor de sus variables privadas (lista) al resto de hebras, copiándoselo a sus propias copias privadas.
- **Cuándo usarlo**: para leer un dato (por ejemplo, mediante `scanf` o calculado en `single`) y que luego todas las hebras lo compartan **como privado**.
- **Sintaxis**:

```
#pragma omp parallel
{
  int x;
  #pragma omp single copyprivate(x)
  {
    // sólo un hilo pide:
    scanf("%d", &x);
  }
  // aquí, tras la single, todas las hebras tienen su copia privada x
  printf("hebra %d: x=%d\n", omp_get_thread_num(), x);
}
```

- **Restricción**: sólo válida con `single`. □cite□turn2file7□

Con estas tres cláusulas ya puedes:

- `reduction`: combinar resultados numéricos de forma segura.
- `copyin`: propagar estado inicial de variables `threadprivate`.
- `copyprivate`: distribuir desde un único hilo el valor inicial de una variable a todos.

5. Cláusulas de sincronización y control de iteraciones

`schedule`

- **Qué hace**: controla cómo se **distribuyen las iteraciones de un bucle** entre los hilos.
- **Opciones**:
  - `static`: las iteraciones se distribuyen en bloques fijos de tamaño igual.
  - `dynamic`: las iteraciones se asignan de manera dinámica a los hilos, lo que puede mejorar el rendimiento cuando las iteraciones tienen tiempos de ejecución muy desiguales.
  - `guided`: similar a `dynamic`, pero las iteraciones más grandes se asignan primero.
- **Sintaxis**:

```
#pragma omp parallel for schedule(static, 4)
for (int i = 0; i < N; ++i) {
  // código paralelo
}
```

  Aquí, se asignan bloques de 4 iteraciones por cada hilo.
- **Ejemplo**:

```
int a[100];
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < 100; i++) {
  // código de trabajo en paralelo
}
```

  En este caso, las iteraciones se reparten dinámicamente entre los hilos según disponibilidad.

`ordered`

- **Qué hace**: asegura que las iteraciones de un bucle **se ejecuten en el orden original**.
- **Uso principal**: cuando necesitas que el orden de ejecución no se pierda dentro de un bucle paralelo.
- **Sintaxis**:

```
#pragma omp parallel for ordered
for (int i = 0; i < N; ++i) {
  #pragma omp ordered
  {
    // Código que debe ejecutarse en orden
  }
}
```

- **Ejemplo**:

```
#pragma omp parallel for ordered
for (int i = 0; i < N; ++i) {
  #pragma omp ordered
```

```
  {
    // hacer algo en orden secuencial
  }
}
```

---

nowait

- **Qué hace**: se usa al final de directivas `for`, `sections` o `single` para **eliminar la sincronización** esperada.
- **Uso principal**: cuando no quieres que se haga una barrera (sincronización implícita) después de una región paralela.
- **Sintaxis**:

```
#pragma omp parallel for nowait
for (int i = 0; i < N; ++i) {
  // código paralelo
}
```

- **Ejemplo**: si no deseas esperar a que todos los hilos terminen antes de proceder:

```
#pragma omp parallel for nowait
for (int i = 0; i < N; ++i) {
  // Código paralelo
}
```

---

atomic

- **Qué hace**: permite que una operación sea **atómica**, es decir, que no sea interrumpida por otro hilo durante su ejecución. Esto es útil para evitar condiciones de carrera en operaciones simples, como un incremento o suma.
- **Sintaxis**:

```
#pragma omp atomic
suma += a[i];
```

Esto asegura que la operación de suma no se interrumpa por otros hilos.

---

critical

- **Qué hace**: se usa para **proteger** una sección de código crítica, permitiendo que **solo un hilo** ejecute esa parte a la vez.

- **Sintaxis**:

```
#pragma omp critical
{
  // código crítico
}
```

- **Ejemplo**:

```
#pragma omp parallel
{
  #pragma omp critical
  {
    // Solo un hilo puede ejecutar este bloque a la vez
  }
}
```

`barrier`

- **Qué hace**: se utiliza para **sincronizar todos los hilos** en una región paralela. Todos los hilos deben llegar al punto de la barrera antes de continuar.
- **Sintaxis**:

```
#pragma omp barrier
```

- **Uso principal**: al final de una sección paralela para esperar a que todos los hilos terminen antes de continuar.

Resumen de cláusulas sincronización y control

- `schedule`: cómo se distribuyen las iteraciones.
- `ordered`: mantiene el orden de ejecución de las iteraciones.
- `nowait`: elimina la sincronización implícita al final de un bucle.
- `atomic`: realiza operaciones atómicas para evitar condiciones de carrera.
- `critical`: asegura que solo un hilo ejecute una sección crítica a la vez.
- `barrier`: sincroniza todos los hilos en un punto.

6. Uso de `threadprivate` en OpenMP

La cláusula `threadprivate` se utiliza para **variables globales estáticas** que deben ser mantenidas para cada hilo de ejecución en una región paralela, **sin que se comparta entre los hilos**. Esto es útil cuando quieres que los hilos mantengan un estado independiente, pero dentro de una variable estática global.

¿Cómo funciona?

1. **Declaración global**: Se debe usar `#pragma omp threadprivate` para las variables estáticas o globales que necesitas tratar como privadas a nivel de hilo.
2. **Propósito**: Cada hilo tiene una copia privada de esa variable, pero el valor se mantiene entre las ejecuciones de distintas regiones paralelas.

Ejemplo:

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

int var = 0;  // Variable global

#pragma omp threadprivate(var)

int main() {
    #pragma omp parallel
    {
        var = omp_get_thread_num();  // Cada hilo obtiene su propio número
        printf("Hilo %d: var = %d\n", omp_get_thread_num(), var);
    }

    // Al salir de la región paralela, cada hilo mantiene su valor de 'var'
    printf("Fuera de la región paralela: var = %d\n", var);
    return 0;
}
```

En este ejemplo, la variable `var` se declara como `threadprivate`. Cada hilo obtiene un valor único para `var`, el cual no es compartido entre hilos. Al final, cada hilo mantiene su propia copia del valor de `var`.

Resumen:

- `threadprivate`: Es para variables **globales estáticas** que deben ser privadas para cada hilo, pero que retienen su valor entre ejecuciones paralelas.
- Utilízalo cuando necesites que **cada hilo** mantenga su propio valor para una variable global a lo largo de varias ejecuciones de región paralela, sin que la variable se comparta entre hilos.

Hemos cubierto una buena parte de las cláusulas en OpenMP, pero aún nos falta hablar sobre los aspectos más avanzados que están relacionados con la planificación del código y la sincronización entre hilos. Continuemos con los últimos puntos de este tema:

7. **Sincronización avanzada y comunicación entre hilos**

Además de las cláusulas de sincronización básicas como `atomic`, `critical`, y `barrier`, OpenMP proporciona otras herramientas para optimizar y controlar el comportamiento de las regiones paralelas.

`flush`

- **Qué hace**: **fuerza** la sincronización entre las vistas de memoria de los hilos, asegurando que los valores en las variables compartidas sean consistentes entre los hilos.
- **Uso**: se usa para garantizar que los valores leídos y escritos por los hilos estén correctamente sincronizados en memoria.
- **Sintaxis**:

```
#pragma omp flush(variable)
```

- **Cuándo usarlo**: cuando deseas asegurar que las modificaciones hechas por un hilo se reflejan inmediatamente en los demás hilos sin esperar a que una barrera ocurra.

Ejemplo de `flush`:

```
int x = 0;
#pragma omp parallel
{
    x = 42;  // Cada hilo establece x en 42
    #pragma omp flush(x)  // Asegura que otros hilos vean el valor actualizado de
x
}
```

En este caso, `flush(x)` asegura que el valor de `x` se actualice en la memoria compartida y sea visible para todos los hilos antes de que continúe la ejecución.

---

8. **Combinación de `parallel` y otras directivas**

Cuando usas directivas como `parallel` junto con otras (por ejemplo, `for`, `sections`, etc.), puedes combinar cláusulas de ambas directivas para controlar mejor el comportamiento paralelo.

`parallel for`

Es una de las combinaciones más comunes, ya que permite ejecutar un bucle en paralelo mientras que el equipo de hilos se crea al mismo tiempo.

Sintaxis:

```
#pragma omp parallel for schedule(static, 4)
for (int i = 0; i < N; ++i) {
    // Código ejecutado en paralelo
}
```

`parallel sections`

Es útil cuando tienes varias secciones de código independientes que pueden ejecutarse simultáneamente.

Sintaxis:

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // Código de la primera sección
    }
    #pragma omp section
    {
        // Código de la segunda sección
    }
}
```

---

9. **Estrategias avanzadas de sincronización y control de ejecución**

En OpenMP también puedes controlar la **ejecución** de las hebras y la sincronización de manera más avanzada mediante directivas adicionales que permiten un control más fino sobre el comportamiento concurrente.

`task`

Permite crear tareas que se pueden ejecutar de forma paralela. Cada tarea puede ser ejecutada por cualquier hilo en el equipo.

Sintaxis:

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
            // Tarea 1
        }
        #pragma omp task
        {
            // Tarea 2
        }
    }
}
```

Las tareas pueden ejecutarse de forma independiente, y OpenMP decide cuándo y cómo asignarlas a los hilos.

`taskwait`

Sincroniza las tareas dentro de una región paralela. Los hilos esperan a que todas las tareas se completen antes de continuar.

Sintaxis:

```
#pragma omp taskwait
```

---

10. **Mejores prácticas de OpenMP**

Cuando programas con OpenMP, es importante seguir ciertas prácticas para evitar errores comunes y maximizar el rendimiento:

- **Minimizar el uso de `critical`**: usar `critical` en exceso puede reducir el rendimiento ya que limita el paralelismo. Solo debería usarse cuando es estrictamente necesario.
- **Uso correcto de `private` y `shared`**: asegúrate de que las variables que no necesitan ser compartidas entre hilos estén marcadas como `private`, y las que deben ser compartidas, como `shared`.
- **Evitar la dependencia de los hilos**: siempre que sea posible, cada hilo debe realizar su trabajo de forma independiente. Las dependencias entre hilos pueden generar cuellos de botella y reducir la eficiencia.
- **No abusar de las barreras**: las barreras sincronizan a todos los hilos, lo que puede causar un gran tiempo de espera innecesario. Usa barreras solo cuando sea necesario.

Resumen Final

- Hemos cubierto las **principales cláusulas** de OpenMP: `shared`, `private`, `reduction`, `copyin`, `atomic`, `critical`, etc.
- También hemos hablado de la **sincronización avanzada**, las **directivas combinadas**, y las **tareas paralelas** con `task`.
- Es importante entender cómo controlar la **distribución de tareas**, cómo evitar **condiciones de carrera**, y cómo **sincronizar los hilos** de forma eficiente.

Perfecto, vamos a repasar algunas **preguntas tipo test** basadas en lo que hemos aprendido. Para esto, tomaremos ejemplos de preguntas de los archivos que has proporcionado. Aquí te dejo algunas que te ayudarán a afianzar los conceptos:

Pregunta 1:

¿Cuál de las siguientes directivas de OpenMP permite que las iteraciones de un bucle se asignen dinámicamente a los hilos disponibles?

a) `schedule(static)`
b) `schedule(guided)`
c) `schedule(dynamic)`
d) `schedule(reduction)`

Respuesta correcta:
c) `schedule(dynamic)`

- **Explicación**: La directiva `schedule(dynamic)` distribuye las iteraciones de un bucle de manera dinámica entre los hilos disponibles.

Pregunta 2:

¿Qué ocurre cuando se utiliza la cláusula `firstprivate(x)` en OpenMP?

a) Cada hilo obtiene una copia privada de `x`, inicializada con el valor de `x` fuera de la región paralela.
b) Cada hilo obtiene una copia privada de `x`, inicializada a 0.
c) Cada hilo tiene acceso a la misma copia de `x` sin alterarlo.
d) Cada hilo obtiene una copia privada de `x`, pero no tiene acceso a la variable `x`.

Respuesta correcta:
a) Cada hilo obtiene una copia privada de `x`, inicializada con el valor de `x` fuera de la región paralela.

- **Explicación**: `firstprivate(x)` crea copias privadas de la variable `x` para cada hilo, y esas copias se inicializan con el valor de `x` fuera de la región paralela.

Pregunta 3:

¿Cuál de las siguientes directivas de OpenMP NO admite cláusulas?

a) `parallel`
b) `single`
c) `master`
d) `critical`

Respuesta correcta:
c) `master`

- **Explicación**: La directiva `master` no admite ninguna cláusula. Es una directiva que solo ejecuta el bloque de código en el hilo maestro.

Pregunta 4:

¿Qué hace la directiva `#pragma omp barrier` en OpenMP?

a) Ejecuta un bloque de código en paralelo.
b) Sincroniza todos los hilos, asegurando que todos lleguen a este punto antes de continuar.
c) Termina la ejecución del bloque paralelo.
d) Asigna dinámicamente las iteraciones del bucle a los hilos.

Respuesta correcta:
b) Sincroniza todos los hilos, asegurando que todos lleguen a este punto antes de continuar.

- **Explicación**: La directiva `barrier` sincroniza a todos los hilos en un punto específico. Los hilos no pueden continuar hasta que todos hayan alcanzado esta barrera.

Pregunta 5:

¿Cuál es la función de la cláusula `reduction` en OpenMP?

a) Permite que los hilos trabajen en paralelo sin compartir datos.
b) Sincroniza las variables entre los hilos.
c) Crea copias locales de una variable para cada hilo, que luego se combinan de acuerdo con el operador indicado.
d) Distribuye las iteraciones del bucle entre los hilos.

Respuesta correcta:
c) Crea copias locales de una variable para cada hilo, que luego se combinan de acuerdo con el operador indicado.

- **Explicación**: La cláusula `reduction` crea copias locales de las variables especificadas en cada hilo, y luego realiza una operación (como suma, multiplicación, etc.) al final de la región paralela para combinar esas copias.

Pregunta 6:

¿Qué ocurre cuando se usa la cláusula `atomic` en OpenMP?

a) Permite que varias operaciones se ejecuten simultáneamente sin sincronización.
b) Realiza una operación en una variable compartida de manera atómica, asegurando que no haya interferencia entre hilos.
c) Ejecuta un bloque de código en secuencia, sin paralelismo.
d) Sincroniza todos los hilos para garantizar la coherencia de las variables.

Respuesta correcta:
b) Realiza una operación en una variable compartida de manera atómica, asegurando que no haya interferencia entre hilos.

- **Explicación**: La directiva `atomic` asegura que una operación sobre una variable compartida se realice de forma atómica, evitando condiciones de carrera sin bloquear toda la región.

Vamos a repasar las preguntas tipo test proporcionadas en los archivos, explicando las respuestas como en las preguntas anteriores. Aquí tienes el formato con las explicaciones:

Pregunta 8:

Analiza el siguiente código e indica qué nos dirá el compilador referido a las siguientes líneas de código:

```
int i, n = 1;

#pragma omp parallel for default(none) private(i)
for (i = 0; i < 5; ++i)
  n += i;
```

a) La variable i no puede ser privada
b) La variable n no está especificada
c) La variable n no está especificada como privada
d) Ninguna opción de las anteriores

Respuesta correcta:
b) La variable n no está especificada

- **Explicación**: En este código, el uso de `default(none)` obliga a que todas las variables compartidas sean especificadas explícitamente. Dado que `n` no está especificada explícitamente, el compilador generará un error indicando que la variable `n` no está especificada.

Pregunta 9:

Indica qué valor tendrá la variable `ret` después de ejecutar la siguiente reducción cuando se ejecuta con 4 hebras:

```
int i, n = 6, ret = 1;

#pragma omp parallel reduction(+:ret)
for (i = omp_get_thread_num(); i < omp_get_max_threads(); i +=
omp_get_num_threads())
  ret += i;
```

a) 5
b) 7
c) El valor de `ret` será indeterminado porque existe una condición de carrera
d) 3

Respuesta correcta:
b) 7

- **Explicación**: En este código, la reducción se realiza sobre la variable `ret`. Los hilos están sumando sus valores a `ret` de forma segura debido a la cláusula `reduction`. Debido a la forma en que se distribuyen las iteraciones entre los hilos, al final, `ret` será igual a 7, ya que los valores de `i` que se suman son 0, 1, 2, 3 (para 4 hebras), y el valor inicial de `ret` era 1.

Pregunta 10:

Asumiendo que `v2` de dimensión `N` y que todos sus elementos están inicializados a cero, ¿cuál de los siguientes códigos calcula de forma correcta el producto de la matriz `m` (dimensión NxN) por el vector `v1` (dimensión N), paralelizando el bucle que recorre las columnas?

a)

```
#pragma omp parallel private(j) for(i = 0; i < N; i++) {
  #pragma omp for reduction(+:v2[i])
  for(j = 0; j < N; j++) {
    v2[i] += m[i][j] * v1[j];
  }
}
```

b)

```
#pragma omp parallel for private(j)
for(i = 0; i < N; i++) {
  for(j = 0; j < N; j++) {
    v2[i] += m[i][j] * v1[j];
  }
}
```

c)

```
#pragma omp parallel private(i, j) for(i = 0; i < N; i++) {
  for(j = 0; j < N; j++) {
    v2[i] += m[i][j] * v1[j];
  }
}
```

d)

```
#pragma omp parallel private(i) for(i = 0; i < N; i++) {
  #pragma omp for reduction(+:v2[i])
  for(j = 0; j < N; j++) {
    v2[i] += m[i][j] * v1[j];
  }
}
```

Respuesta correcta:
b)

- **Explicación**: La opción b) paraleliza correctamente el bucle exterior sobre las filas de la matriz, distribuyendo las iteraciones entre los hilos. Además, cada hilo acumula el resultado de su propia fila en

`v2[i]` sin necesidad de una reducción, ya que el acceso a las posiciones de `v2` es independiente.

Pregunta 11:

¿Cuál de las siguientes directivas no admite ninguna cláusula?

a) `single`
b) `critical`
c) `parallel`
d) Todas las directivas admiten alguna cláusula

Respuesta correcta:
a) `single`

- **Explicación**: La directiva `single` no admite ninguna cláusula adicional. Su único propósito es asegurar que un solo hilo ejecute el bloque de código contenido.

Pregunta 12:

Supongamos una máquina en la que el número de hebras de las que se puede disponer para ejecutar zonas paralelas de código es limitado. En esas condiciones, ¿qué valdrá la variable `n` al terminar de ejecutar el siguiente código?

```
int n = 1;

#pragma omp parallel for firstprivate(n) lastprivate(n)
for (int i = 0; i < 5; ++i)
  n += 1;
```

a) n al salir del bucle está indefinido porque es privada
b) Dependerá del número de hebras que lo ejecuten en cada momento
c) 1
d) 5

Respuesta correcta:
b) Dependerá del número de hebras que lo ejecuten en cada momento

- **Explicación**: La variable `n` se declara como `firstprivate`, lo que significa que cada hilo tiene una copia inicializada con el valor de `n` al inicio. `lastprivate` garantiza que el valor final de `n` de la última iteración del bucle se copie de vuelta a la variable original, pero el valor dependerá de cuántos hilos estén involucrados y cómo se distribuyen las iteraciones.

Pregunta 13:

¿Cuál será el valor de `n` al final de la ejecución del siguiente código?

```
int i, n = 2;

#pragma omp parallel shared(n) private(i)
for(i = 0; i < 4; i++) {
  #pragma omp single
  {
    n += i;
  }
}
```

a) 16
b) 8
c) 2
d) Indeterminado

Respuesta correcta:
b) 8

- **Explicación**: La directiva `single` asegura que solo un hilo ejecute el bloque dentro de ella. Como `n` es una variable compartida entre los hilos, solo un hilo sumará los valores de `i` (0, 1, 2, 3) a `n`. El valor final de `n` será 8 (inicialmente 2 + 0 + 1 + 2 + 3).

Pregunta 14:

¿Cuál de las siguientes afirmaciones es correcta?

a) Las cláusulas ajustan el comportamiento de las directivas
b) Las cláusulas deben ser siempre especificadas junto con las directivas
c) Las directivas `parallel` y `for` no pueden usarse juntas
d) Ninguna de las anteriores

Respuesta correcta:
a) Las cláusulas ajustan el comportamiento de las directivas

- **Explicación**: Las cláusulas permiten modificar cómo se comportan las directivas, controlando aspectos como el alcance de las variables, la forma de distribución de las iteraciones, entre otros.

Parece que hemos cubierto la mayoría de las preguntas en los archivos anteriores. Sin embargo, en los documentos proporcionados existen algunas preguntas adicionales. A continuación, las presento con las explicaciones, siguiendo el formato que te ha gustado:

Pregunta 12:

¿Cuál de las siguientes directivas no admite ninguna cláusula?

a) `single`
b) `critical`

c) `parallel`
d) Todas las directivas admiten alguna cláusula

Respuesta correcta:
a) `single`

- **Explicación**: La directiva `single` no admite ninguna cláusula adicional. Su único propósito es ejecutar el bloque de código contenido en ella solo una vez en un hilo, sin modificar otras configuraciones.

Pregunta 13:

Supongamos una máquina en la que el número de hebras de las que se puede disponer para ejecutar zonas paralelas de código es limitado. En esas condiciones, ¿qué valdrá la variable `n` al terminar de ejecutar el siguiente código?

```
int n = 1;
#pragma omp parallel for firstprivate(n) lastprivate(n)
for (int i = 0; i < 5; ++i)
  n += i;
```

a) `n` al salir del bucle está indefinida porque es privada
b) Dependerá del número de hebras que lo ejecuten en cada momento
c) 1
d) 5

Respuesta correcta:
b) Dependerá del número de hebras que lo ejecuten en cada momento

- **Explicación**: La cláusula `firstprivate(n)` asegura que cada hilo obtendrá su propia copia inicializada con el valor de `n`. La cláusula `lastprivate(n)` garantiza que el valor final de `n` en la última iteración se copie de vuelta a la variable global `n`. El resultado final depende de cómo los hilos ejecuten las iteraciones del bucle y de cuántos hilos estén involucrados.

Pregunta 14:

¿Cuál será el valor de `n` tras ejecutar el siguiente código?

```
int i, n = 2;
#pragma omp parallel shared(n) private(i)
for(i = 0; i < 4; i++){
  #pragma omp single
  {
    n += i;
  }
}
```

a) 16
b) 8
c) 2
d) Indeterminado

Respuesta correcta:
b) 8

- **Explicación**: La directiva `single` garantiza que solo un hilo ejecutará el bloque de código dentro de ella. En este caso, el valor de `n` se incrementa por los valores de `i` (0, 1, 2, 3) en la única ejecución de ese bloque. El valor final de `n` será 8 (inicialmente 2 + 0 + 1 + 2 + 3).

Parece que hemos cubierto una parte importante de las preguntas, pero algunas más están disponibles. Aquí están las siguientes preguntas que faltaban:

Pregunta 15:

¿Cuál de los siguientes fragmentos de código paralelo calcula correctamente la sumatoria de los primeros números impares hasta N = 1000?

a)

```
int sum = 0;
#pragma omp parallel for
for (long i = 1; i < N; i += 2)
  sum += i;
```

b)

```
int sum = 0;
#pragma omp parallel
for (long i = 1; i < N; i += 2)
  sum += i;
```

c)

```
int sum = 0;
#pragma omp parallel sections {
  #pragma omp section
  for (long i = 1; i < N; i += 4)
    sum += i;
  #pragma omp section
  for (long i = 3; i < N; i += 4)
    sum += i;
}
```

d) Ninguna otra respuesta es correcta

Respuesta correcta:
c)

- **Explicación**: La opción c) utiliza `sections` para dividir el trabajo entre los hilos. Cada sección calcula una parte de la sumatoria de los números impares. Esta opción es la más adecuada ya que divide el rango de los números impares entre las secciones, lo que permite una ejecución paralela eficiente.

Pregunta 16:

Indica cuál será el valor de la variable `n` al final de la ejecución del siguiente código:

```
int n = 0;
#pragma omp parallel for reduction(*:n)
for (int i = n; i < size; ++i)
  n += i;
```

a) Ninguna respuesta es correcta
b) El valor de `n` será igual a `size - 1`
c) 0
d) Dependerá del valor de la variable `size`

Respuesta correcta:
a) Ninguna respuesta es correcta

- **Explicación**: La cláusula `reduction(*:n)` no es válida en este contexto. Las reducciones suelen usarse con operaciones como `+`, `-`, `*` o `&`, pero la operación `*` no es apropiada para la acumulación de valores de una variable sumando, ya que afectaría negativamente a la operación de acumulación. Además, el uso de `n` en la inicialización del bucle podría generar un comportamiento inesperado, ya que el valor de `n` está siendo modificado por cada hilo.

Pregunta 17:

¿Cuánto vale `ret` al final de este código?

```
int i, n = 6; ret = 1;
#pragma omp parallel reduction(+:ret) private(i)
for (i = omp_get_thread_num(); i < n; i += omp_get_num_threads())
  ret += i;
return ret;
```

a) 15
b) 1

c) 16
d) Ninguna de las otras respuestas es correcta

Respuesta correcta:
a) 15

- **Explicación**: La reducción en la variable `ret` hace que cada hilo acumule su valor de forma independiente, sumando los valores de `i` para cada hilo. Dado que `n = 6` y cada hilo realiza parte de la suma (sin compartir el resultado), al final el valor de `ret` será la suma total de todas las iteraciones realizadas por cada hilo, que es 15.

Pregunta 18:

¿Cuál de las siguientes afirmaciones es correcta?

a) Las cláusulas ajustan el comportamiento de las directivas
b) Las cláusulas deben ser siempre especificadas junto con las directivas
c) Las directivas `parallel` y `for` no pueden usarse juntas
d) Ninguna de las anteriores

Respuesta correcta:
a) Las cláusulas ajustan el comportamiento de las directivas

- **Explicación**: Las cláusulas permiten especificar detalles adicionales sobre cómo se comportan las directivas. Por ejemplo, pueden usarse para definir el ámbito de las variables, la distribución de iteraciones o la sincronización entre hilos. No es obligatorio usarlas, pero ajustan el comportamiento de la paralelización.
