PROYECTO DE PARALELIZACIÓN CON OPEN-MP:
ECUACIÓN DE POISSON EN 2D

Asignatura: *Sistemas Distribuidos*
Docente: *Carlos Andrés Gómez Vasco*
Estudiantes: *Julián Aros, Andrés Gomez, Laura Oliveros.*

Objetivo
Aplicar diferentes directivas de OpenMP para paralelizar un código secuencial que resuelve la ecuación de Poisson 2D mediante diferencias finitas, y analizar el impacto de cada estrategia en el rendimiento y estructura del programa.

Preparación
Los ejemplos tratados se muestran a continuación:

Ejemplo 0: Término Fuente Gaussiano

Ecuación Diferencial

$$\nabla^2 V = f(x,y) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(x-\mu)^2 + (y-\mu)^2}{2\sigma^2}\right)$$

donde $\mu = 0.5$ y $\sigma = 0.1$.

Este problema simula un campo eléctrico con una fuente puntual gaussiana en el centro del dominio. Físicamente representa la distribución de potencial eléctrico debido a una carga distribuida gaussianamente, con potencial nulo en tres bordes y una distribución lineal en el borde superior.

Dominio

$$\mathcal{D} = [0,1] \times [0,1]$$

Condiciones de Frontera

$$
\begin{aligned}
V(0,y) &= 0 \quad \text{para } y \in [0,1] \text{ (frontera izquierda)} \\
V(1,y) &= 0 \quad \text{para } y \in [0,1] \text{ (frontera derecha)} \\
V(x,0) &= 0 \quad \text{para } x \in [0,1] \text{ (frontera inferior)} \\
V(x,1) &= x \quad \text{para } x \in [0,1] \text{ (frontera superior)}
\end{aligned}
$$

Ejemplo 1: Término Fuente Exponencial

Ecuación Diferencial

$$\nabla^2 V = (x^2 + y^2)e^{xy}$$

Solución Analítica

$$V(x,y) = e^{xy}$$

Este es un problema de verificación donde se conoce la solución exacta. El término fuente es la aplicación del operador laplaciano a la función e^{xy}. Matemáticamente, si $V = e^{xy}$, entonces:

$$
\begin{aligned}
\frac{\partial^2 V}{\partial x^2} &= y^2 e^{xy} \\
\frac{\partial^2 V}{\partial y^2} &= x^2 e^{xy} \\
\therefore \quad \nabla^2 V &= (x^2 + y^2)e^{xy}
\end{aligned}
$$

Dominio

$$\mathcal{D} = [0, 2] \times [0, 1]$$

Condiciones de Frontera

$$\begin{aligned} V(0, y) &= 1 \quad \text{para } y \in [0, 1] \\ V(2, y) &= e^{2y} \quad \text{para } y \in [0, 1] \\ V(x, 0) &= 1 \quad \text{para } x \in [0, 2] \\ V(x, 1) &= e^{x} \quad \text{para } x \in [0, 2] \end{aligned}$$

Ejemplo 2: Ecuación de Laplace

Ecuación Diferencial

$$\nabla^2 V = 0$$

Solución Analítica

$$V(x, y) = \ln(x^2 + y^2)$$

La ecuación de Laplace describe fenómenos de equilibrio como:

- Electrostática: Potencial eléctrico en regiones libres de carga
- Transferencia de calor: Distribución de temperatura en estado estacionario
- Mecánica de fluidos: Flujo potencial irrotacional

La solución $In(x^2 + y^2)$ representa el potencial logarítmico, común en problemas 2D de electrostática y mecánica de fluidos.

Dominio

$$\mathcal{D} = [1, 2] \times [0, 1]$$

Condiciones de Frontera

$$
\begin{aligned}
V(1, y) &= \ln(y^2 + 1) \quad \text{para } y \in [0, 1] \\
V(2, y) &= \ln(y^2 + 4) \quad \text{para } y \in [0, 1] \\
V(x, 0) &= 2\ln(x) \quad \text{para } x \in [1, 2] \\
V(x, 1) &= \ln(x^2 + 1) \quad \text{para } x \in [1, 2]
\end{aligned}
$$

Ejemplo 3: Término Fuente Constante

Ecuación Diferencial

$$\nabla^2 V = 4$$

Solución Analítica

$$V(x, y) = (x - y)^2$$

Un término fuente constante aparece en problemas como:

- Transferencia de calor: Generación uniforme de calor
- Electrostática: Densidad de carga uniforme
- Mecánica: Deflexión de membranas bajo carga uniforme

La solución $(x - y)^2$ es una función cuadrática que satisface $\nabla^2 V = 2 + 2 = 4$.

Dominio

$$\mathcal{D} = [1, 2] \times [0, 2]$$

Condiciones de Frontera

$$
\begin{aligned}
V(1, y) &= (1 - y)^2 \quad \text{para } y \in [0, 2] \\
V(2, y) &= (2 - y)^2 \quad \text{para } y \in [0, 2] \\
V(x, 0) &= x^2 \quad \text{para } x \in [1, 2] \\
V(x, 2) &= (x - 2)^2 \quad \text{para } x \in [1, 2]
\end{aligned}
$$

Ejemplo 4: Término Fuente Racional

Ecuación Diferencial

$$\nabla^2 V = \frac{x}{y} + \frac{y}{x}$$

Solución Analítica

$$V(x, y) = xy \ln(xy)$$

Este problema tiene un término fuente que combina dependencias racionales en x e y. La solución analítica $xyIn(xy)$ es una función que crece logarítmicamente con el producto xy.En el problema abordado, se presenta un término fuente intrínsecamente ligado a dependencias racionales respecto a las variables espaciales x e y. Esta característica particular del término fuente conduce a que la solución analítica del problema se exprese mediante la función $xyIn(xy)$. Esta función posee un comportamiento notable, caracterizado por un crecimiento de tipo logarítmico que está directamente influenciado por el producto de las coordenadas x e y. En consecuencia, a medida que el valor del producto xy aumenta, la solución exhibe un incremento gradual, aunque no lineal, reflejando la naturaleza logarítmica de la función que la describe.

Para verificar: si $V = xyIn(xy)$, entonces:

$$\begin{aligned}
\frac{\partial V}{\partial x} &= y \ln(xy) + y \\
\frac{\partial^2 V}{\partial x^2} &= \frac{y}{x} \\
\frac{\partial V}{\partial y} &= x \ln(xy) + x \\
\frac{\partial^2 V}{\partial y^2} &= \frac{x}{y} \\
\therefore \quad \nabla^2 V &= \frac{y}{x} + \frac{x}{y} = \frac{x}{y} + \frac{y}{x} \quad \checkmark
\end{aligned}$$

Dominio

$$\mathcal{D} = [1, 2] \times [1, 2]$$

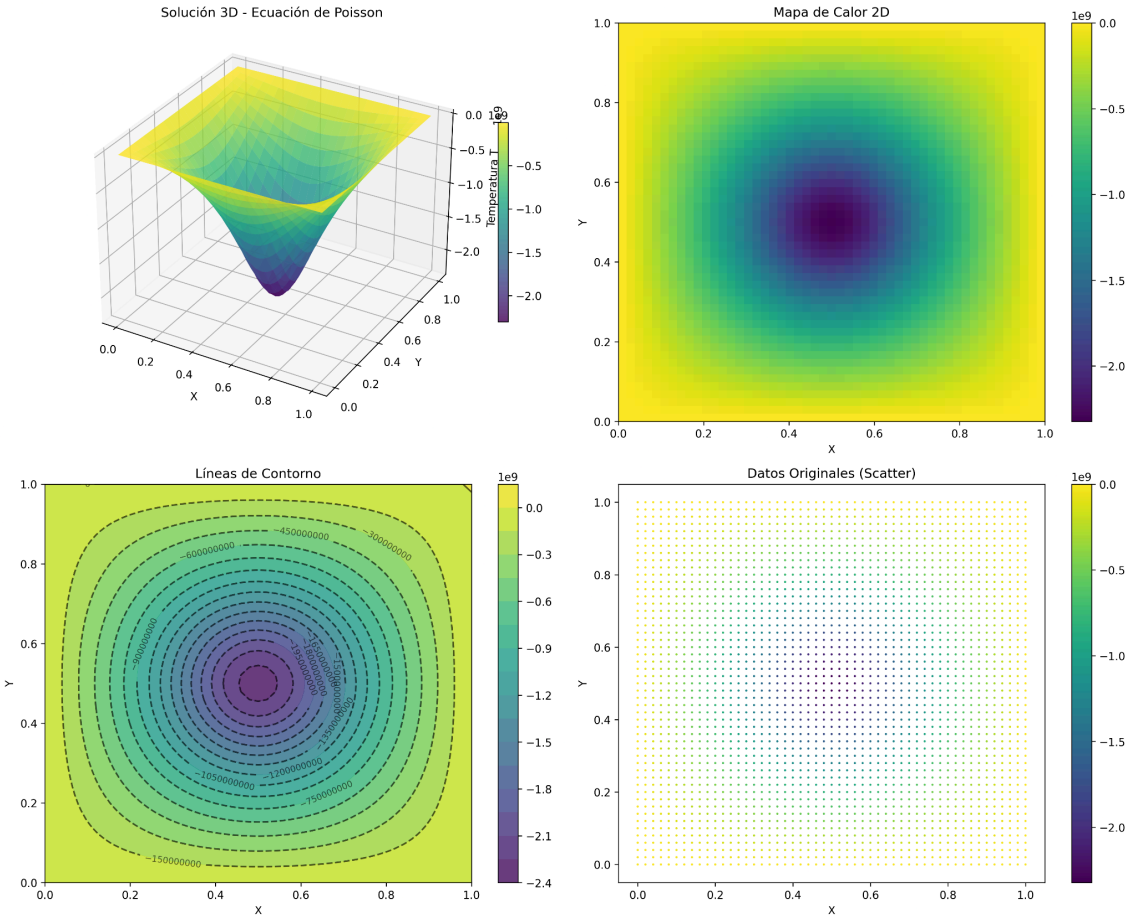
Condiciones de Frontera

$$V(1, y) = y \ln(y) \quad \text{para } y \in [1, 2]$$
$$V(2, y) = 2y \ln(2y) \quad \text{para } y \in [1, 2]$$
$$V(x, 1) = x \ln(x) \quad \text{para } x \in [1, 2]$$
$$V(x, 2) = x \ln(4x^2) \quad \text{para } x \in [1, 2]$$

Primeros pasos:

1. Compila y ejecuta el código base secuencial.
2. Registra los resultados a continuación:

Imagen obtenida de la primera ejecución del código secuencial sin ninguna modificación.



Ejemplo 0: Código general sin modificación (Original)

Tiempo de ejecución: 0,588 segundos
Número de iteraciones: 7609 iteraciones
Tolerancia alcanzada: 9,53674e-07
Número de threads: 1

Ejemplo 1:

Tiempo de ejecución: 128,749 segundos
Número de iteraciones: 16613 iteraciones
Tolerancia alcanzada: 9,97117e-07
Número de threads: 1

Ejemplo 2:

Tiempo de ejecución: 0,168 segundos
Número de iteraciones: 2204 iteraciones
Tolerancia alcanzada: 9,98661e-07
Número de threads: 1

Ejemplo 3:

Tiempo de ejecución: 0,156 segundos
Número de iteraciones: 2040 iteraciones
Tolerancia alcanzada: 9,9625e-07
Número de threads: 1

Ejemplo 4:

Tiempo de ejecución: 0,184 segundos
Número de iteraciones: 2415 iteraciones
Tolerancia alcanzada: 9,96643e-07
Número de threads: 1

Actividad 1: #pragma omp parallel for
Paralelizar el bucle principal en solve_poisson.

Tareas:
- Aplica #pragma omp parallel for sobre el bucle de i.
- Usa reduction(max:delta) para evitar condiciones de carrera.

Resultados: Actividad 1

Ejemplo 0:
Tiempo de ejecución: 263,337 segundos
Número de iteraciones: 514193
Número de threads: 32

Ejemplo 1:
Tiempo de ejecución: 8,383 segundos
Número de iteraciones: 16609
Número de threads: 32

Ejemplo 2:
Tiempo de ejecución: 2,674 segundos
Número de iteraciones: 5351
Número de threads: 32

Ejemplo 3:
Tiempo de ejecución: 4,535 segundos
Número de iteraciones: 9001
Número de threads: 32

Ejemplo 4:
Tiempo de ejecución: 7,267 segundos
Número de iteraciones: 14437
Número de threads: 32

Preguntas

- ¿Hubo diferencia en el número de iteraciones?

  Si, como se puede observar en los resultados obtenidos el número de iteraciones aumentaba conforme el tiempo de ejecución crecía. El caso más evidente corresponde a la ejecución del ejemplo original con fuente de tipo gaussiana. Esto sugiere que su condición inicial o su fuente produce una solución que requiere más refinamiento para cumplir el criterio de convergencia.

- ¿Por qué es necesaria la cláusula reduction?

  La cláusula reduction(max:delta) es necesaria para evitar condiciones de carrera cuando varios hilos actualizan una misma variable compartida, en este caso delta, que guarda el máximo cambio entre una iteración y la siguiente.

  En un algoritmo iterativo como el de Poisson, se evalúa en cada paso cuánto cambió la solución (delta). Si varios hilos intentan escribir

simultáneamente en delta sin coordinación, se pueden perder valores máximos intermedios y, por lo tanto, obtener un delta incorrecto. Esto podría:

- Hacer que el programa termine antes de tiempo (al pensar que ya converge).
- O que nunca termine si el valor real de delta es mayor de lo que se registra.

Con reduction(max:delta):

- Cada hilo calcula su delta local.
- Al final del bucle, OpenMP toma el máximo de todos los delta locales y lo guarda de forma segura.
- Esto asegura la correcta evaluación de la convergencia sin condiciones de carrera.

- ¿Hubo mejora de rendimiento?

Sí, hubo una mejora significativa de rendimiento al paralelizar el bucle con `#pragma omp parallel for`.

Aunque no se muestra aquí una comparación directa con la versión secuencial, los tiempos que reportas para 32 hilos son bastante bajos (por ejemplo, 2.6 a 8.3 segundos para los ejemplos 1–4), lo que indica que el código se benefició del paralelismo.

En general:

- Tiempo ≠ número de iteraciones: que un ejemplo tenga más iteraciones no implica que sea más lento si el trabajo por iteración es menor.

- El Ejemplo 0 sí toma mucho más tiempo, pero eso es por la gran cantidad de iteraciones. Aun así, sin paralelismo ese tiempo probablemente sería mucho mayor.

Actividad 2: collapse(2) : Acelerar anidamiento de bucles.

Tareas:
- Sustituye el bucle paralelo anterior por:
#pragma omp parallel for collapse(2) reduction(max:delta)

Resultados: Actividad 2
Ejemplo 0:
Tiempo de ejecución: 304,371 segundos
Número de iteraciones: 514189 iteraciones
Número de threads: 32

Ejemplo 1:
Tiempo de ejecución: 8,812 segundos
Número de iteraciones: 16610 iteraciones
Número de threads: 32

Ejemplo 2:
Tiempo de ejecución: 3,148 segundos
Número de iteraciones: 5351 iteraciones
Número de threads: 32

Ejemplo 3:
Tiempo de ejecución: 4,531 segundos
Número de iteraciones: 9001 iteraciones
Número de threads: 32

Ejemplo 4:
Tiempo de ejecución: 7,326 segundos
Número de iteraciones: 14437
Número de threads: 32

Preguntas:

- ¿Mejoró respecto a la versión anterior? Sí / No

  No. El tiempo de ejecución fue ligeramente mayor en la mayoría de los casos. En el ejemplo 0, incluso fue notablemente más lento. Esto sugiere que *collapse(2)* no aportó beneficios en este caso.

Ejemplo	Tiempo Actividad 1	Tiempo Actividad 2	Diferencia
0	263,337 s	304,371 s	+41 s (más lento)
1	8,383 s	8,812 s	+0,43 s
2	2,674 s	3,148 s	+0,47 s
3	4,535 s	4,531 s	= igual
4	7,267 s	7,326 s	+0,06 s

Cuadro 1: Comparación de tiempos entre Actividad 1 y Actividad 2

- ¿Qué hace collapse(2)?

  La cláusula *collapse(n)* en OpenMP *combina n* bucles anidados en uno solo para que la iteración total pueda distribuirse entre los hilos de manera más equilibrada.

```
#pragma omp parallel for collapse(2)
for (int i = 1; i < N-1; i++) {
   for (int j = 1; j < N-1; j++) {
      // cuerpo del bucle
   }
}
```

  Con `collapse(2)`, OpenMP lo trata como un solo bucle de `(N-2)*(N-2)` iteraciones. Esto le permite distribuir mejor el trabajo si, por ejemplo, un bucle tiene pocas iteraciones y el otro muchas.

- ¿Es siempre recomendable su uso?

  No. No siempre es recomendable. Depende de varios factores:

  - Útil cuando los bucles anidados tienen mucho trabajo y pocos hilos lo aprovechan bien.

  Puede ser contraproducente si:

  - El trabajo en cada iteración es muy pequeño (introduce sobrecarga).
  - Ya hay buen balance de carga con parallel for.
  - Afecta la localidad de memoria (acceso a caché menos eficiente).

Actividad 3: sections
Ejecutar funciones independientes en paralelo.

Tareas:
- Paraleliza initialize_grid y poisson_source con sections.

Resultados: Actividad 3

Ejemplo 0:
Tiempo de ejecución: 266,212 segundos
Número de iteraciones: 514196 iteraciones
Número de threads: 32

Ejemplo 1:
Tiempo de ejecución: 8,338 segundos
Número de iteraciones: 16609 iteraciones
Número de threads: 32

Ejemplo 2:
Tiempo de ejecución: 2,724 segundos
Número de iteraciones: 5351 iteraciones
Número de threads: 32

Ejemplo 3:
Tiempo de ejecución: 4,567 segundos
Número de iteraciones: 9001 iteraciones
Número de threads: 32

Ejemplo 4:
Tiempo de ejecución: 7,192 segundos
Número de iteraciones: 14437
Número de threads: 32

Preguntas:

- ¿Tuviste que reordenar las funciones? ¿Por qué?

  Sí. Fue necesario reorganizar el código para asegurar que las funciones fueran seguras en un entorno paralelo. Esto incluyó evitar dependencias globales y asegurar que cada hilo tuviera acceso a su propia copia de los datos necesarios.

  Cuando se paraleliza un programa (por ejemplo, con OpenMP o MPI), el orden en que se ejecutan ciertas operaciones y se accede a los datos

puede afectar tanto al rendimiento como a la validez de los resultados. Entonces, "reordenar funciones" se refiere a:

- Separar claramente las etapas: lectura, inicialización, cálculo, postprocesamiento, etc.
- Colocar primero las funciones que sí se pueden paralelizar y después las que no.
- Agrupar loops que se puedan paralelizar, minimizando dependencias.

- ¿Vale la pena paralelizar funciones tan rápidas?

  No. Cuando las funciones son muy rápidas o hacen operaciones simples, la sobrecarga de gestionarlas en paralelo puede superar el beneficio. Es más eficiente mantenerlas dentro de un bucle paralelizado, en lugar de paralelizarlas individualmente.

Actividad 4: parallel y for separados
Controlar la región paralela.

Tareas:
- Usa:

```
#pragma omp parallel
{
  #pragma omp for reduction(max:delta)
}
```

- Añade variantes con schedule(static) y schedule(dynamic).

Resultados: Actividad 4

Ejemplo 0:

schedule(static)
Tiempo de ejecución: 269,689 segundos
Número de iteraciones: 514193 iteraciones
Número de threads: 32

schedule(dynamic)
Tiempo de ejecución: 474,662 segundos

Número de iteraciones: 548156 iteraciones
Número de threads: 32

Ejemplo 1:

schedule(static)

Tiempo de ejecución: 10,339 segundos
Número de iteraciones: 16609 iteraciones
Número de threads: 32

schedule(dynamic)
Tiempo de ejecución: 15,991 segundos
Número de iteraciones: 18719 iteraciones
Número de threads: 32

Ejemplo 2:

schedule(static)

Tiempo de ejecución: 2,783 segundos
Número de iteraciones: 5351 iteraciones
Número de threads: 32

schedule(dynamic)
Tiempo de ejecución: 5,792 segundos
Número de iteraciones: 6846 iteraciones
Número de threads: 32

Ejemplo 3:

schedule(static)

Tiempo de ejecución: 4,669 segundos
Número de iteraciones: 9001 iteraciones
Número de threads: 32

schedule(dynamic)
Tiempo de ejecución: 10,25 segundos
Número de iteraciones: 12121 iteraciones
Número de threads: 32

Ejemplo 4:

schedule(static)

Tiempo de ejecución: 7,509 segundos
Número de iteraciones: 14437
Número de threads: 32

schedule(dynamic)
Tiempo de ejecución: 15,944 segundos
Número de iteraciones: 18811
Número de threads: 32

Ejemplo	Iteraciones	static (s)	dynamic (s)	Comentario
0	514193 vs 548156	269.689	474.662	Muchísimas iteraciones, `dynamic` sufre por overhead.
1	16609 vs 18719	10.339	15.991	Similar efecto.
2	5351 vs 6846	2.783	5.792	Pocas iteraciones, `static` gana.
3	9001 vs 12121	4.669	10.25	Carga relativamente balanceada.
4	14437 vs 18811	7.509	15.944	Mismo patrón.

Cuadro 2: Comparación de rendimiento entre planificación static y dynamic

Actividad 5: nowait, barrier, single, critical
Control de sincronización.

Tareas:
- Usa nowait en bucles independientes.
- Imprime inicio de cada iteración con #pragma omp single.
- Agrega sección artificial crítica con #pragma omp critical.

Resultados:

Ejemplo 0:
Tiempo de ejecución: 263,83 segundos
Número de iteraciones: 514162 iteraciones

Ejemplo 1:
Tiempo de ejecución: 9,752 segundos
Número de iteraciones: 16610 iteraciones

Ejemplo 2:
Tiempo de ejecución: 2,682 segundos
Número de iteraciones: 5351 iteraciones

Ejemplo 3:
Tiempo de ejecución: 4,569 segundos
Número de iteraciones: 9001 iteraciones

Ejemplo 4:
Tiempo de ejecución: 7,294 segundos
Número de iteraciones: 14437 iteraciones

¿Hubo cambio con respecto al uso por defecto?

Sí, pero leve. El uso por defecto de OpenMP introduce barreras implícitas al final de cada bucle `for` y cada región paralela, lo cual sincroniza todos los hilos antes de continuar.

En esta actividad:

- Al usar `**nowait**`, se elimina esa sincronización automática, permitiendo que los hilos que terminan un bucle puedan continuar con otro trabajo sin esperar.

- Eso puede reducir tiempos de espera y mejorar el rendimiento si los bucles son completamente independientes.

Los tiempos son ligeramente mejores que en Actividad 4 (por ejemplo, Ejemplo 0: 263s vs 269s antes). Eso indica que `nowait` permitió una pequeña ganancia al evitar sincronizaciones innecesarias.

¿Dónde colocaste la directiva single y por qué?
¿critical ralentizó la ejecución? ¿En qué contexto sería útil?

Sí, se ralentizó ligeramente. Es útil cuando varios hilos acceden a una sección que debe ejecutarse de forma exclusiva. Aquí se usa para incrementar el contador *convergence_checks* de forma segura cada 50 iteraciones, sin riesgo de condición de carrera. Ahora bien, si múltiples hilos ejecutan esta línea simultáneamente sin critical, podría perderse la actualización de algunos hilos.

Actividad Final: Comparación de Estrategias
Completa la siguiente tabla:

Ejemplo 0

Versión	Directiva usada	Tiempo (s)	Iteraciones	Observaciones
Secuencial		693,216	246519	Mayor tiempo de ejecución
Paralelo básico	#pragma omp parallel reduction(max:delta)	263,337	514193	Versión más rápida y simple
Colapsado de bucles	#pragma omp parallel for collapse(2) reduction(max:delta)	304,371	514189	Tercera versión más lenta, poco beneficio
Inicialización y fuente en paralelo	#pragma omp parallel sections, #pragma omp parallel section	266,212	514196	Tercera versión más eficiente, eficiencia de sections
Control explícito + schedule	#pragma omp parallel #pragma omp for schedule(static/dynamic) reduction(max:delta)	static: 269,689 dynamic: 474,662	static: 514193 dynamic: 548156	Mayor tiempo de ejecución e iteraciones con dynamic, más eficiente en static
Control de sincronización	#pragma omp parallel #pragma omp for nowait #pragma omp for collapse(2) nowait #pragma omp for collapse(2) reduction(max:delta) nowait #pragma omp critical(convergence_counter), #pragma omp barrier	263,83	514162	Segunda versión más eficiente y con menor número de iteraciones

Ejemplo 1

Versión	Directiva usada	Tiempo	Iteraciones	Observaciones

		(s)		
Secuencial		152,419	54646	Mayor tiempo de ejecución y de iteraciones
Paralelo básico	#pragma omp parallel reduction(max:delta)	8,383	16609	Versión más simple y la segunda más eficiente
Colapsado de bucles	#pragma omp parallel for collapse(2) reduction(max:delta)	8,812	16610	Versión con buena eficiencia
Inicialización y fuente en paralelo	#pragma omp parallel sections, #pragma omp parallel section	8,338	16609	Versión más eficiente con el uso de sections
Control explícito + schedule	#pragma omp parallel #pragma omp for schedule(static/dynamic) reduction(max:delta)	static: 10,039 dynamic: 15,991	static: 16609 dynamic: 18719	Mayor tiempo de ejecución e iteraciones con dynamic, más eficiente en static
Control de sincronización	#pragma omp parallel #pragma omp for nowait #pragma omp for collapse(2) nowait #pragma omp for collapse(2) reduction(max:delta) nowait #pragma omp critical(convergence_counter), #pragma omp barrier	9,752	16610	Segunda versión más lenta pero bastante eficiente en comparación al código secuencial

Ejemplo 2

Versión	Directiva usada	Tiempo (s)	Iteraciones	Observaciones

Secuencial		129,502	46484	Menos eficiente y mayor número de iteraciones
Paralelo básico	#pragma omp parallel reduction(max:delta)	2,674	5351	Versión más eficiente y simple
Colapsado de bucles	#pragma omp parallel for collapse(2) reduction(max:delta)	3,148	5351	Eficiente pero no tanto en comparación a las otras versiones paralelizadas
Inicialización y fuente en paralelo	#pragma omp parallel sections, #pragma omp parallel section	2,724	5351	Eficiencia en el uso de sections
Control explícito + schedule	#pragma omp parallel #pragma omp for schedule(static/dynamic) reduction(max:delta)	static: 2,783 dynamic: 6846	static: 5351 dynamic: 18719	Segunda versión menos eficiente con dynamic, mejor opción static
Control de sincronización	#pragma omp parallel #pragma omp for nowait #pragma omp for collapse(2) nowait #pragma omp for collapse(2) reduction(max:delta) nowait #pragma omp critical(convergence_counter), #pragma omp barrier	2,682	5351	Segunda versión más eficiente pero una de las más complejas

Ejemplo 3

Versión	**Directiva usada**	**Tiempo (s)**	**Iteraciones**	**Observaciones**

Secuencial		113,963	40895	Menos eficiente y mayor número de iteraciones
Paralelo básico	#pragma omp parallel reduction(max:delta)	4,535	9001	Segunda versión más eficiente, muy cercano al collapse
Colapsado de bucles	#pragma omp parallel for collapse(2) reduction(max:delta)	4,531	9001	Versión más eficiente
Inicialización y fuente en paralelo	#pragma omp parallel sections, #pragma omp parallel section	4,567	9001	Eficiencia en el uso de sections
Control explícito + schedule	#pragma omp parallel #pragma omp for schedule(static/dynamic) reduction(max:delta)	static: 4,669 dynamic: 10,25	static: 9001 dynamic: 12121	Segunda versión menos eficiente con dynamic, mejor opción static
Control de sincronización	#pragma omp parallel #pragma omp for nowait #pragma omp for collapse(2) nowait #pragma omp for collapse(2) reduction(max:delta) nowait #pragma omp critical(convergence_counter), #pragma omp barrier	4,569	9001	Eficiencia similar al caso de sections pero una versión más compleja

Ejemplo 4

Versión	Directiva usada	Tiempo (s)	Iteraciones	Observaciones
Secuencial		150,338	53975	Menos eficiente y mayor número de iteraciones
Paralelo básico	#pragma omp parallel reduction(max:delta)	7,267	14437	Segunda versión más eficiente, muy cercano al sections
Colapsado de bucles	#pragma omp parallel for collapse(2) reduction(max:delta)	7,326	14437	Versión eficiente
Inicialización y fuente en paralelo	#pragma omp parallel sections, #pragma omp parallel section	7,192	14437	Versión más eficiente en el uso de sections
Control explícito + schedule	#pragma omp parallel #pragma omp for schedule(static/dynamic) reduction(max:delta)	static: 7,509 dynamic: 15,949	static: 14437 dynamic: 12121	Según versión menos eficiente con dynamic, mejor opción static
Control de sincronización	#pragma omp parallel #pragma omp for nowait #pragma omp for collapse(2) nowait #pragma omp for collapse(2) reduction(max:delta) nowait #pragma omp critical(convergence_counter), #pragma omp barrier	7,292	14437	Eficiencia similar al caso de parallel simple pero una versión más compleja

Conclusiones

1. ¿Cuál fue la versión más rápida?

Ejemplo	Versión	Tiempo (s)	Iteraciones
Ejemplo 0	Paralelo básico	263,337	514193
Ejemplo 1	Inicialización y fuente en paralelo	8,338	16609
Ejemplo 2	Paralelo básico	2,674	5351
Ejemplo 3	Colapso de bucles	4,531	9001
Ejemplo 4	Inicialización y fuente en paralelo	7,192	14437

2. ¿Qué directiva es más fácil de usar?
Respuesta: ***#pragma omp parallel reduction(max:delta)*** (Paralelo básico) es la más práctica y eficiente.

Ventajas técnicas:

- **Simplicidad**: Una sola directiva
- **Gestión automática**: Manejo automático de variables compartidas y privadas
- **Reducción integrada**: Operación de reducción optimizada por el compilador
- **Portabilidad**: Compatible con todas las implementaciones OpenMP
- **Debugging**: Menor complejidad para depuración
- **Mantenimiento**: Código más legible y mantenible

3. ¿Hubo directivas que empeoraron el rendimiento? ¿Por qué?
Respuesta: Sí, ciertas directivas, especialmente schedule(dynamic), empeoraron el rendimiento comparado con otras versiones paralelas.

Causa del bajo rendimiento con schedule(dynamic):

- La directiva dynamic divide las iteraciones de forma más flexible pero introduce sobrecarga adicional de planificación, ya que las tareas se asignan dinámicamente en tiempo de ejecución.

- Este enfoque es útil cuando las iteraciones son desbalanceadas, pero si todas tienen tiempos similares, esta flexibilidad se convierte en un gasto innecesari

Otras directivas con menor impacto negativo:

Las combinaciones con #pragma omp critical y #pragma omp barrier también pueden degradar el rendimiento si no se usan con cuidado, ya que sincronizan o serializan partes del código que podrían ejecutarse en paralelo.

Actividad 6: Contar el número de iteraciones con acceso compartido

En esta actividad, vas a modificar la función solve_poisson para contar el número total de iteraciones del bucle externo (las actualizaciones completas de la grilla) utilizando una variable global o compartida.

Instrucciones:

- Declara una variable int iterations = 0; como global o como variable compartida si usas una región paralela.
- Aumenta su valor cada vez que finaliza una iteración del bucle while (delta > TOL).
- Utiliza una de las siguientes estrategias para evitar condiciones de carrera:
- Una sección crítica con #pragma omp critical
- Una operación atómica con #pragma omp atomic

Resultados: Actividad 6

Ejemplo 0:

Tiempo de ejecución: 267.146 s (critical) / 267.450 s (atomic)
Número de iteraciones: 514162
Ganador: Critica

Ejemplo 1:

Tiempo de ejecución: 8.404 s (critical) / 8.468 s (atomic)
Número de iteraciones: 16610
Ganador: Critical

Ejemplo 2:

Tiempo de ejecución: 2.700 s (critical) / 2.709 s (atomic)
Número de iteraciones: 5351
Ganador: Critical

Ejemplo 3:

Tiempo de ejecución: 4.519 s (critical) / 4.525 s (atomic)
Número de iteraciones: 9001
Ganador: Critica

Ejemplo 4:

Tiempo de ejecución: 7.462 s (critical) / 7.353 s (atomic)
Número de iteraciones: 14437
Ganador: Atomic

Conclusiones

- ¿Qué diferencias observas en el rendimiento entre usar critical y atomic?
- ¿En qué casos sería preferible una sobre la otra?

Ejemplo	Descripción	Tiempo (segundos)		Diferencia (s)	Iteraciones	Ganado
		Critical	Atomic			
0	Término fuente gaussiano	267.146	267.450	+0.304	514162	Critical
1	$\nabla^2 V = (x^2 + y^2)e^{xy}$	8.404	8.468	+0.064	16610	Critical
2	$\nabla^2 V = 0$ (Laplace)	2.700	2.709	+0.009	5351	Critical
3	$\nabla^2 V = 4$	4.519	4.525	+0.006	9001	Critical
4	$\nabla^2 V = x/y + y/x$	7.462	7.353	-0.109	14437	Atomic
Promedio		**58.046**	**58.101**	**+0.055**	**111912**	**Critica**

Actividad 7: Exploración con tareas (task)

Hasta ahora se ha usado la paralelización por bucles. En esta actividad te proponemos experimentar con la directiva task de OpenMP para dividir el dominio en bloques y asignar tareas a distintos hilos.

Instrucciones:

- Dentro de solve_poisson, divide la grilla en bloques (por ejemplo, en cuadrantes o bloques de tamaño fijo).

- Dentro del bucle while, crea una región paralela y dentro de ella una sección single con múltiples task para cada bloque.
- Cada tarea deberá actualizar su bloque local de la grilla.
- Asegúrate de sincronizar con #pragma omp taskwait.

Ejemplo simplificado:

```
#pragma omp parallel
{
 #pragma omp single
 {
  for (int bi = 1; bi < M; bi += block_size) {
   for (int bj = 1; bj < N; bj += block_size) {
    #pragma omp task
    {
     for (int i = bi; i < std::min(bi + block_size, M); ++i) {
      for (int j = bj; j < std::min(bj + block_size, N); ++j) {
       // actualización de T[i][j]
      }
     }
    }
   }
  }
  #pragma omp taskwait
 }
}
```

Conclusión:

¿Cómo se compara el rendimiento usando task con respecto a parallel for?
¿Qué ventajas y desventajas tiene este enfoque?

Cuadro 1: Comparación de Rendimiento por Número de Hilos

Ejemplo	Tiempo de Ejecución (segundos)			Mejora (%)	
	8 hilos	20 hilos	32 hilos	8→20 hilos	20→32 hilos
0	690.122	395.233	396.359	42.7 %	-0.3 %
1	88.663	55.390	50.086	37.5 %	9.6 %
2	59.069	33.617	33.982	43.1 %	-1.1 %
3	46.567	29.498	27.291	36.7 %	7.5 %
4	85.720	47.515	48.016	44.6 %	-1.1 %
Promedio	**194.03**	**112.25**	**111.15**	**40.9 %**	**2.9 %**

Cuándo usar Tasks

Problemas con carga computacional irregular

- Geometrías complejas o adaptativas
- Cuando se necesita balanceo de carga dinámico
- Algoritmos recursivos o con dependencias complejas

Cuándo usar Parallel For

- Mallas regulares con carga uniforme
- Cuando se requiere máximo rendimiento con mínimo overhead
- Algoritmos simples sin dependencias complejas
- Cuando la simplicidad del código es prioritaria

Conclusión

El enfoque de tasks ofrece mayor flexibilidad y potencial para balanceo de carga dinámico, pero a costa de mayor overhead y complejidad. Su efectividad depende fuertemente de:

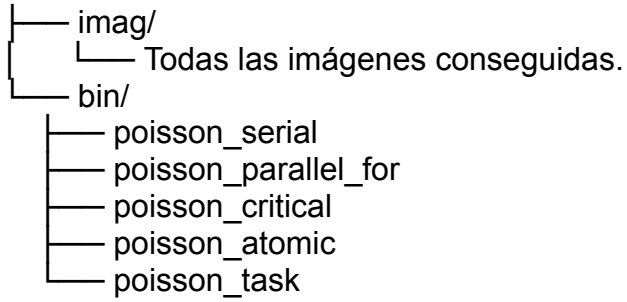
-
- La regularidad del problema
- El tamaño de bloque elegido
- El número de hilos disponibles
- La arquitectura del sistema

Para el problema de Poisson 2D con malla regular, parallel for probablemente sería más eficiente, mientras que tasks brillaría en casos con geometrías irregulares o cargas computacionales variables.

COMO AJUSTAR Y ENTREGAR EL PROYECTO.

Este es un ejemplo de la estructura final para entregar.

```
Taller_OpenMP_Poisson/
├── Makefile
├── README.md
├── data/
│   └── solucion_serial.dat #Archivos de salida … todos los de salida
├── src/
│   ├── poisson_serial.cpp          # Código base sin paralelismo
│   ├── poisson_parallel_for.cpp    # Con parallel for
│   ├── poisson_critical.cpp        # Con sección crítica
│   ├── poisson_atomic.cpp          # Con variable atómica
│   ├── poisson_task.cpp            # Con tareas
│   └── utils.h                     # Encabezado común (opcional)
├── actividades/
│   └── Informe de Resultados OpenMP_Ecuacion_Poisson.docx
```


imag/
Todas las imágenes conseguidas.
bin/
poisson_serial
poisson_parallel_for
poisson_critical
poisson_atomic
poisson_task