CAGE – Grupo de Investigación

@sc3uis

SC3 – Super Cómputo y Cálculo Científico – UIS

Programación paralela con OpenMP

Shared memory: hilos, directivas y rendimiento

Módulo 4 – Semillero CAGE

Luis Alejandro Torres Niño | sc3.operaciones@uis.edu.co

Objetivos del módulo

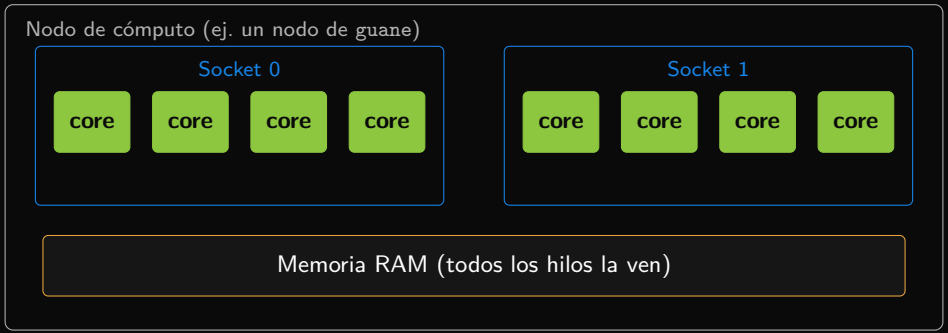
- ➔ Entender el modelo de **shared memory** y el esquema *fork–join* de OpenMP.
- ➔ Paralelizar bucles con `#pragma omp parallel for` y controlar qué datos son compartidos o privados.
- ➔ Reconocer y corregir **race conditions** (`atomic`, `critical`, `reduction`).
- ➔ Balancear la carga con `schedule` y expresar paralelismo irregular con **tasks**.
- ➔ Medir **speedup** y **eficiencia**; entender *false sharing*, NUMA y afinidad.
- ➔ Ejecutar programas OpenMP en el cluster con SLURM (`--cpus-per-task`).

Agenda

1. Paralelismo en shared memory y modelo OpenMP
2. Paralelizar bucles: `parallel for`, datos compartidos y privados
3. Race conditions y sincronización
4. Scheduling de iteraciones y load balancing
5. Tasks y sections
6. Rendimiento: speedup, *false sharing*, NUMA y afinidad
7. OpenMP en el cluster con SLURM
8. Retos individuales y reto grupal

1
Paralelismo en shared memory

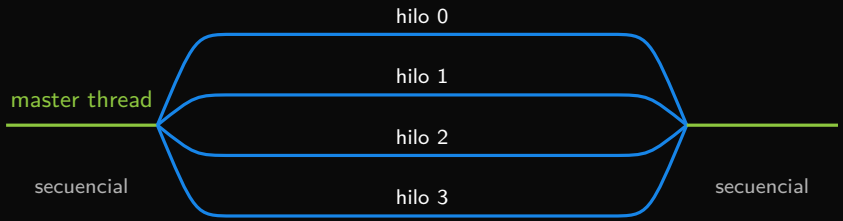
Un nodo es una máquina de shared memory



Procesos (MPI, módulo 5): memoria *separada*, se comunican con mensajes.

Hilos (OpenMP): *comparten* el espacio de direcciones del proceso; comunicarse es leer y escribir variables.

El modelo fork–join



El programa corre con un hilo; al entrar a una parallel region se **crea un equipo de hilos** y al terminar todos se sincronizan.

¿Qué es OpenMP?

Una API estándar para C, C++ y Fortran (openmp.org) con tres componentes:

- **Directivas** del compilador: `#pragma omp …`
- **Biblioteca de ejecución**: `omp_get_thread_num()`, `omp_get_wtime()`…
- **Variables de entorno**: `OMP_NUM_THREADS`, `OMP_SCHEDULE`…

Es **incremental**: un programa secuencial correcto sigue siéndolo si el compilador ignora los `pragma` (siempre que no use llamadas `omp_*`).

★ Compilar y ejecutar

```
module load gnu14
g++ -std=c++23 -O2 -fopenmp \
    hola.cpp -o hola
export OMP_NUM_THREADS=4
./hola
```

`-fopenmp` activa las directivas y enlaza `libgomp`.

Hola OpenMP

```cpp
#include <omp.h>
#include <cstdio>

int main() {
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int n  = omp_get_num_threads();
        std::printf("Hola desde el hilo %d de %d\n", id, n);
    }
}
```

```
$ OMP_NUM_THREADS=2 ./hola
Hola desde el hilo 0 de 2
Hola desde el hilo 1 de 2
```

El orden de las líneas **no está garantizado**.

Número de hilos: `OMP_NUM_THREADS`, `omp_set_num_threads(n)` o la cláusula `num_threads(n)`.

Reto individual 1: conoce tu equipo de hilos

■ Reto 1 – Threads y entorno

1. Compila `hola.cpp` con `-fopenmp` y ejecútalo con 1, 2, 4 y 8 hilos.
2. Imprime también el nodo (`gethostname`) y `omp_get_num_procs()`. ¿Coinciden con los núcleos que pediste a SLURM?
3. Ejecútalo **sin** `OMP_NUM_THREADS` dentro de un `srun -p legacy -n 1`. ¿Cuántos hilos crea por defecto? ¿Por qué?
4. Compílalo **sin** `-fopenmp`: ¿qué error aparece y por qué? ¿Cómo lo evitarías con `#ifdef _OPENMP`?

2
Paralelizar bucles

parallel for: el caballo de batalla

```cpp
#include <omp.h>
#include <cstdio>

int main() {
    const long N = 200000000;
    const double h = 1.0 / N;
    double suma = 0.0;
    double t0 = omp_get_wtime();

    #pragma omp parallel for reduction(+:suma)
    for (long i = 0; i < N; ++i) {
        double x = (i + 0.5) * h;
        suma += 4.0 / (1.0 + x * x);          // integral de 4/(1+x^2) = pi
    }
    std::printf("pi ~ %.10f  (%d hilos, %.3f s)\n",
                suma * h, omp_get_max_threads(), omp_get_wtime() - t0);
}
```

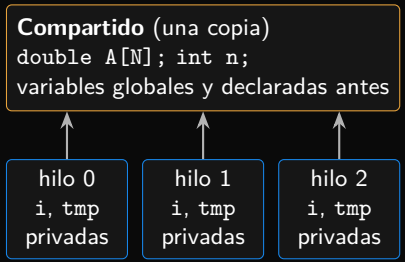
Cada hilo recibe un bloque de iteraciones. Las iteraciones deben ser **independientes** entre sí.

¿Cómo se reparten las iteraciones?



- Por defecto (`schedule(static)`) el rango se parte en **bloques contiguos** de tamaño similar.
- El índice del bucle (`i`) es **privado** de cada hilo automáticamente.
- Al final del bucle hay una **barrier implícita**; con `nowait` se elimina.

Datos compartidos y privados




```c
int n = 1000;
double escala = 2.0;
double tmp;

#pragma omp parallel for \
        private(tmp) firstprivate(escala)
for (int i = 0; i < n; ++i) {
    tmp  = A[i] * escala;   // tmp: privada
    B[i] = tmp + 1.0;       // B: compartido
}
```

Regla de oro: lo declarado *dentro* de la región es privado; lo declarado *fuera* es compartido, salvo que lo cambies con una cláusula.

Cláusulas de datos

Cláusula	Efecto
`shared(x)`	Todos los hilos usan la misma `x`.
`private(x)`	Cada hilo tiene su `x`, **sin inicializar**.
`firstprivate(x)`	Copia privada **inicializada** con el valor previo.
`lastprivate(x)`	Al terminar, `x` toma el valor de la **última iteración**.
`reduction(+:s)`	Copia privada por hilo; al final se **combinan** con el operador.
`default(none)`	Obliga a declarar el alcance de cada variable (**recomendado**).

★ Buen hábito

Escribe `default(none)`: el compilador te avisa de cada variable cuyo alcance no declaraste, y así se evitan *races* silenciosas.

Reto individual 2: calcula pi en paralelo

Reto 2 – Escalabilidad de pi

1. Compila el programa de pi con `-fopenmp` y usa `default(none)` en el `pragma`. Sin `-fopenmp` no enlaza: ¿por qué?
2. Ejecuta con 1, 2, 4, 8 y 16 hilos (con `srun -p legacy -c <n>`) y registra el tiempo.
3. Calcula **speedup** $S = T_1/T_p$ y **eficiencia** $E = S/p$; haz una tabla y una gráfica.
4. ¿Hasta cuántos hilos escala bien? ¿Qué ocurre cuando p supera los núcleos físicos del nodo?

3
Race conditions y sincronización

Un data race en acción

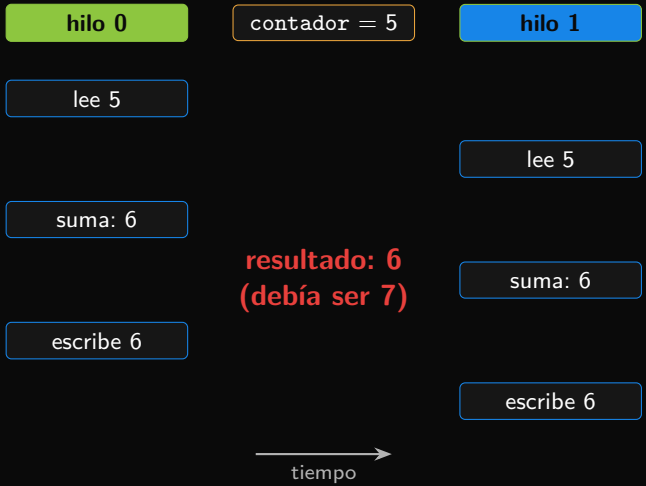
```
const long N = 10000000;
volatile long contador = 0;
#pragma omp parallel for
for (long i = 0; i < N; ++i)
    contador = contador + 1;          // varios hilos leen y escriben a la vez
std::printf("esperado %ld, obtenido %ld\n", N, contador);
```

```
$ OMP_NUM_THREADS=2 ./carrera
esperado 10000000, obtenido 5386913       # cambia en cada ejecucion
```

❄ Comportamiento indefinido

Si dos hilos acceden a la misma variable, al menos uno escribe y no hay sincronización, hay una ***data race***. A veces el resultado "sale bien" por casualidad u optimizaciones del compilador; el `volatile` solo evita que se oculte en este ejemplo.

¿Por qué se pierden actualizaciones?



Tres soluciones

```
#pragma omp parallel for reduction(+:suma)      // 1) la mejor
for (long i = 0; i < N; ++i) suma += v[i];

#pragma omp parallel for
for (long i = 0; i < N; ++i) {
    #pragma omp atomic                           // 2) actualizacion atomica
    a++;
    #pragma omp critical                         // 3) bloque en exclusion mutua
    { b += 2; }
}
```

Mecanismo	Cubre	Costo
`reduction`	acumulaciones (`+ * min max`)	bajo (sin contención)
`atomic`	una actualización de variable	medio
`critical`	cualquier bloque	alto (serializa)

Otras herramientas de sincronización

- ➔ `#pragma omp barrier`: todos esperan aquí.
- ➔ `single`: solo **un** hilo (cualquiera) ejecuta el bloque; los demás esperan al final.
- ➔ `master`: solo el hilo 0, sin espera.
- ➔ `nowait`: quita la barrier al final de un `for`.

```
#pragma omp parallel
{
    #pragma omp single
    std::printf("preparando\n");   //
        1 hilo

    #pragma omp for nowait
    for (int i = 0; i < n; ++i) A[i] =
         f(i);

    #pragma omp for
    for (int i = 0; i < n; ++i) B[i] =
         g(i); // no usa A
}
```

★ Menos es más

Cada sincronización hace esperar a los hilos: prefiere **no compartir escrituras** (*reductions*, datos privados) a proteger con `critical`.

Reto individual 3: caza de races

■ Reto 3 – Cuatro programas con errores

Dado un conjunto de programas con *races* (histograma, *prefix sum*, búsqueda del máximo, contador compartido):

1. Ejecuta cada uno 10 veces con 4 hilos. ¿El resultado cambia?
2. Identifica la variable que se comparte indebidamente.
3. Corrígelos con `reduction`, `atomic` o `critical`, **eligiendo el más barato** en cada caso.
4. Compara los tiempos de las tres versiones del contador (`atomic`, `critical`, `reduction`) y explica el orden.

4
Scheduling y load balancing

La cláusula schedule



`static[,k]`	Reparto fijo en *chunks* de `k`. Sin *overhead*; ideal si cada iteración cuesta lo mismo.
`dynamic[,k]`	Los hilos piden un *chunk* de `k` al terminar el anterior. Balancea carga desigual; más *overhead*.
`guided[,k]`	Como `dynamic` con *chunks* que **decrecen**. Compromiso entre ambos.
`runtime`	Se elige con `OMP_SCHEDULE` sin recompilar.

Ver el reparto en la práctica

```
#pragma omp parallel for \
        schedule(static, 2) num_threads
            (2)
for (int i = 0; i < 8; ++i)
    std::printf("i=%d hilo %d\n",
                i, omp_get_thread_num());
```

```
i=0 hilo 0
i=1 hilo 0
i=4 hilo 0
i=5 hilo 0
i=2 hilo 1
i=3 hilo 1
i=6 hilo 1
i=7 hilo 1
```

Con `static,2` los *chunks* de 2 iteraciones se asignan **en turnos**: hilo 0 recibe $\{0, 1\}$, hilo 1 $\{2, 3\}$, hilo 0 $\{4, 5\}$…

★ Cuándo usar cuál

Costo uniforme → `static`. Costo irregular (Mandelbrot, grafos, matrices triangulares) → `dynamic` o `guided`.

collapse: bucles anidados

```
// 2 bucles anidados -> n*m iteraciones repartibles entre los hilos
#pragma omp parallel for collapse(2) schedule(static)
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        C[i*m + j] = A[i*m + j] + B[i*m + j];
```

- ➔ Con pocas iteraciones externas (`n` menor que los hilos) `collapse(2)` da más trabajo para repartir.
- ➔ Requiere que los bucles sean **perfectamente anidados** y sin dependencias.
- ➔ **Dependencias** entre iteraciones (`a[i] = a[i-1] + 1`) impiden `parallel for`: hay que reescribir el algoritmo (por ejemplo, un *scan*).

Reto individual 4: Mandelbrot y load balancing

■ Reto 4 – Carga desigual

1. Escribe un programa que calcule el conjunto de Mandelbrot de 2000×2000 píxeles (máximo 1000 iteraciones) y guárdalo como imagen PGM.
2. Paraleliza el bucle de filas con `schedule(static)`, `dynamic`, `dynamic,16` y `guided`.
3. Mide el tiempo con 8 hilos y mide también cuánto tarda **cada hilo** (`omp_get_wtime` dentro de la región).
4. Dibuja o describe qué hilo recibe qué filas y explica por qué `static` desbalancea.

5
Tasks y sections

Paralelismo irregular: sections y task

sections: bloques **fijos** distintos

```
#pragma omp parallel sections
{
    #pragma omp section
    leer_entrada();
    #pragma omp section
    preparar_tablas();
}
```

task: unidades **dinámicas**

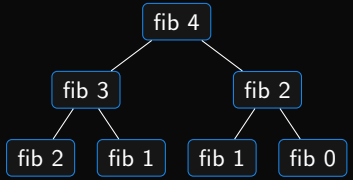
```
#pragma omp parallel
#pragma omp single
{
    for (Nodo* p = lista; p; p = p->
        sig) {
        #pragma omp task firstprivate(
            p)
        procesar(p);          // 1
            task por nodo
    }
}
```

- ➔ Un hilo (`single`) **genera** las tasks y todo el equipo las ejecuta cuando queda libre.
- ➔ Ideal para listas, árboles, recursión y problemas cuyo tamaño no se conoce de antemano.

Tasks recursivas: Fibonacci

```
long fib(int n) {
    if (n < 2) return n;
    long a, b;
    #pragma omp task shared(a) if (n > 20)
    a = fib(n - 1);
    #pragma omp task shared(b) if (n > 20)
    b = fib(n - 2);
    #pragma omp taskwait        // espera a
        las hijas
    return a + b;
}

int main() {
    long r;
    #pragma omp parallel
    #pragma omp single
    r = fib(32);
    std::printf("fib(32) = %ld\n", r);   //
        2178309
}
```



Cada nodo es una task. `if (n > 20)` es un **umbral**: bajo ese tamaño la task se ejecuta de inmediato en el mismo hilo (*undeferred*), sin esperar en la cola; `final(n<=20)` evita además crear sub-tasks.

Reto individual 5: umbral de tasks

Reto 5 – ¿Cuántas tasks es demasiado?

1. Compila el `fib` con tasks y mide $fib(40)$ con umbrales $n > 10, 15, 20, 25, 30$ y sin umbral.
2. Compara con la versión secuencial. ¿A partir de qué umbral hay speedup?
3. Escribe un `mergesort` recursivo con `task` y un umbral de tamaño; ordena 10^7 enteros y mide con 1, 2, 4 y 8 hilos.
4. ¿Por qué las tasks **muy pequeñas** empeoran el tiempo?

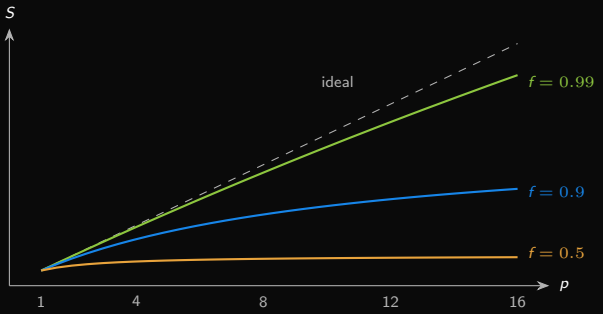
6
Rendimiento

Speedup, eficiencia y la ley de Amdahl

$$S(p) = \frac{T_1}{T_p} \qquad E(p) = \frac{S(p)}{p}$$

$$S(p) \leq \frac{1}{(1-f) + \frac{f}{p}}$$

$f =$ fracción paralelizable. Aunque $p \to \infty$, $S \leq 1/(1-f)$: con $f = 0.9$ nunca pasarás de $10\times$.



★ Referencia

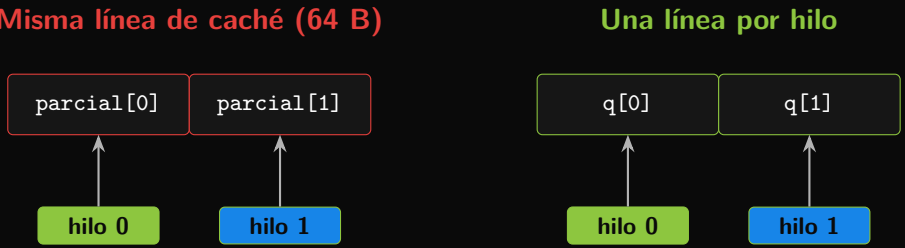
G. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, AFIPS 1967.

Resultados medidos

Programa	1 hilo	2 hilos	Speedup
pi ($N = 2 \cdot 10^8$), tiempo	0.298 s	0.159 s	1.9×
matmul `ikj` $n = 1024$, tiempo	0.678 s	0.343 s	2.0×
matmul `ikj` $n = 1024$, GFLOP/s	3.17	6.26	2.0×

- Medido con `g++ 14 -O2 -fopenmp` en una máquina virtual de **2 núcleos**: solo sirve para ilustrar; **mide en los nodos de `guane`**.
- Ambos programas son *casi* 100% paralelos y no tienen contención: por eso el speedup se acerca al ideal.
- Usa `omp_get_wtime()` (reloj de pared), no `clock()` (suma el tiempo de CPU de todos los hilos).

False sharing: compartir sin querer



Cada escritura **invalida** la línea en la caché del otro núcleo: los hilos se pelean aunque usen variables distintas. Medido con 2 hilos: compartida **0.292 s** vs una línea por hilo **0.037 s** ($\approx 8\times$).

➔ **Solución**: acumula en una variable *local*, o **alinea** (`alignas(64)`) cada dato por hilo; mejor aún: usa `reduction`.

NUMA, *first touch* y afinidad



- En nodos de varios sockets, cada socket tiene su **memoria local**; acceder a la del otro es más lento.
- **First touch**: la página se ubica en el socket del hilo que la **escribe primero**. Inicializa los arreglos **en paralelo** con el mismo reparto que usarás luego.

```
export OMP_PLACES=cores          # un hilo por nucleo fisico
export OMP_PROC_BIND=close       # hilos vecinos en el mismo socket (o: spread)
lscpu | grep -i numa             # ver la topologia del nodo
```

Vectorización: omp simd

```
#pragma omp parallel for simd
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];            // SAXPY: hilos + instrucciones
        vectoriales
```

- ➔ Dos niveles de paralelismo en un núcleo: **hilos** (`parallel for`) y **vectores** dentro de cada hilo (`simd`).
- ➔ Ayuda al compilador: `-O3 -march=native` y sin dependencias; `#pragma omp simd reduction(+:s)` para sumas.
- ➔ Revisa si se vectorizó: `-fopt-info-vec-optimized`.

★ Ojo con `-march=native`

Compila en el mismo tipo de nodo en el que ejecutarás: las instrucciones vectoriales cambian entre CPUs (por ejemplo, entre nodos de `legacy` y de `amd`).

7
OpenMP en el cluster

Script SLURM para OpenMP

```
#!/bin/bash
#SBATCH --job-name=pi_omp
#SBATCH --partition=legacy
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8            # nucleos = hilos
#SBATCH --time=00:10:00
#SBATCH --output=pi_%j.out

module load gnu14
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK     # hilos = nucleos reservados
export OMP_PROC_BIND=close OMP_PLACES=cores

./pi
```

- **Un proceso**, muchos núcleos: `--ntasks=1 --cpus-per-task=N`. OpenMP **no** sale del nodo.
- Consulta los núcleos de cada nodo con `scontrol show node guane01` antes de elegir *N*.

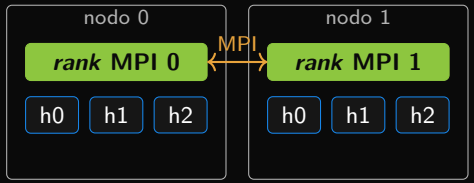
Barrido de escalabilidad

```
#!/bin/bash
#SBATCH --partition=legacy --nodes=1 --ntasks=1
#SBATCH --cpus-per-task=16 --exclusive
#SBATCH --output=scal_%j.out

module load gnu14
for p in 1 2 4 8 16; do
    export OMP_NUM_THREADS=$p
    echo -n "hilos=$p  "
    ./pi
done
```

- Reserva el nodo con `--exclusive` para no medir con vecinos ruidosos.
- Repite el script al menos 3 veces (o añade un bucle interno) y reporta mínimo o mediana.
- Guarda las salidas y grafica speedup y eficiencia (Python/matplotlib o `gnuplot`).

Más allá: OpenMP + MPI y GPUs



Híbrido: MPI entre nodos, OpenMP dentro de cada nodo (módulo 5).

```
#pragma omp target teams \
    distribute parallel for \
    map(to: x[0:n]) map(tofrom: y[0:n])
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
```

`target` hace *offload* del bucle a una GPU. Requiere un compilador con *offload* activado: revisa `module avail`. Para GPU, ver CUDA (módulo 6) y AMD/ROCm (módulo 7).

8
Retos grupales

Reto grupal: difusión de calor 2D

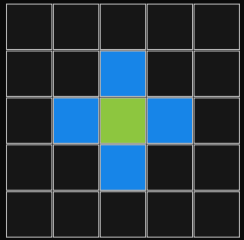
★ Reto grupal – Equipos de 3 o 4 personas

Simulen la difusión de calor en una placa con el método de Jacobi (*grid* $N \times N$, bordes a temperatura fija):

1. Versión **secuencial** en C++23 con `std::vector` y verificación contra una solución conocida.
2. Versión OpenMP con `parallel for collapse(2)` y `reduction(max:err)` para el criterio de convergencia.
3. Inicialización en paralelo (*first touch*) y prueba de `OMP_PROC_BIND`.
4. Barrido de $p = 1, 2, 4, 8, 16$ y de tamaños $N = 512, 1024, 4096$ con `sbatch --exclusive`.
5. Gráficas de **speedup** y **eficiencia**; identifiquen qué limita la escalabilidad (ancho de banda de memoria, sincronización).

Esquema del reto grupal

```
for (int it = 0; it < maxit; ++it) {
    double err = 0.0;
    #pragma omp parallel for collapse
        (2) \
            reduction(max:err)
    for (int i = 1; i < N-1; ++i)
      for (int j = 1; j < N-1; ++j) {
        Tn[i*N+j] = 0.25 * (T[(i-1)*N+j
            ] +
            T[(i+1)*N+j] + T[i*N+j-1] +
            T[i*N+j+1]);
        err = std::max(err,
            std::abs(Tn[i*N+j]-T[i*N+j
                ]));
      }
    std::swap(T, Tn);
    if (err < tol) break;
}
```



Cada celda nueva es el **promedio de sus 4 vecinas** (patrón *stencil*). Se lee `T` y se escribe `Tn`: sin *races*.

Entregables y criterios

Entregables

- ➔ Código secuencial y OpenMP con `Makefile`.
- ➔ Scripts `sbatch` del barrido.
- ➔ Tabla y gráficas de speedup y eficiencia.
- ➔ Informe corto: qué limita el escalado.

Criterios

- ➔ Resultados idénticos (o dentro de tolerancia) entre secuencial y paralelo.
- ➔ Cero *races*: `default(none)` y revisión de datos compartidos.
- ➔ Mediciones repetidas y con nodo exclusivo.
- ➔ Análisis con la ley de Amdahl y la jerarquía de memoria.

Reutilización: paralelicen también el producto de matrices de su `minila` (módulo 3) y compárenlo con la versión secuencial.

Referencias

- OpenMP Architecture Review Board, *OpenMP Application Programming Interface* (especificación y ejemplos): openmp.org.
- B. Barney, *OpenMP*, Lawrence Livermore National Laboratory: hpc-tutorials.llnl.gov/openmp.
- B. Chapman, G. Jost, R. van der Pas, *Using OpenMP*, MIT Press, 2008.
- R. van der Pas, E. Stotzer, C. Terboven, *Using OpenMP – The Next Step*, MIT Press, 2017.
- G. Amdahl, *Validity of the single processor approach...*, AFIPS Spring Joint Computer Conf., 1967.
- U. Drepper, *What Every Programmer Should Know About Memory*, 2007 (cachés y NUMA).
- Documentación de GCC: gcc.gnu.org/wiki/openmp (estado del soporte de OpenMP en `libgomp`).

Cierre

★ OpenMP en una línea

Un `pragma` paraleliza un bucle, pero la **corrección** (*races*), el reparto del trabajo (*scheduling*) y la memoria (*false sharing*, NUMA) deciden el *speedup*.

¡Gracias!

sc3.operaciones@uis.edu.co