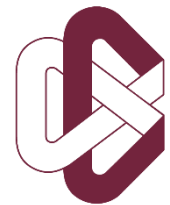




**CONAHCYT**

CONSEJO NACIONAL DE HUMANIDADES  
CIENCIAS Y TECNOLOGÍAS



**CIMAT**

# OPENMP, PARTE 2

Francisco J. Hernández López

fcoj23@cimat.mx



# SINCRONIZACIÓN

- **Exclusión mutua**
  - Utilizado para controlar el acceso a alguna variable compartida
  - La directiva **critical**, permite que un hilo a la vez tenga permitido ejecutar cierto código
- **Sincronización de eventos**
  - Para señalar un evento a través de los múltiples hilos
  - La directiva **barrier**, define un punto en el que cada hilo espera a que lleguen todos los demás
  - Una vez que llegan todos los hilos en ese punto, entonces todos pueden continuar su ejecución
  - Con esto se garantiza que todo el código anterior a la barrera lo han realizado todos los hilos

# RESTRICCIONES EN LOS CICLOS

```
for (index = start ; index < end ; increment_expr)
```

- El comportamiento del ciclo debe ser calculable en tiempo de ejecución, según los parámetros: *start*, *end* y el *increment\_expr*
- *start*, *end*: pueden ser alguna expresión numérica cuyo valor no cambia durante la ejecución del ciclo
- *increment\_expr*: debe cambiar el valor de *index* la misma cantidad después de cada iteración

No están permitidos dentro del ciclo:

- *exit* o *goto* (Fortran)
- *break* o *goto* (C)

<i>Operator</i>	<i>Forms of increment_expr</i>
++	<i>index++</i> or <i>++index</i>
--	<i>index--</i> or <i>--index</i>
+=	<i>index += incr</i>
-=	<i>index -= incr</i>
=	<i>index = index + incr</i> or <i>index = incr + index</i> or <i>index = index - incr</i>

# CICLOS ANIDADOS

```
subroutine sums(a, M, N)
integer M, N, a(0:M, N), i, j
```

```
!$omp parallel do
do j = 1, N
  a(0, j) = 0
  do i = 1, M
    a(0, j) = a(0, j) + a(i, j)
  enddo
enddo
end
```

```
subroutine smooth(a, M, N)
integer M, N, a(0:M + 1, 0:N), i, j
```

```
do j = 1, N
!$omp parallel do
  do i = 1, M
    a(i, j) = (a(i - 1, j - 1) + a(i, j - 1) + &
              a(i + 1, j - 1))/3.0
  enddo
enddo
end
```

# UNIR CICLOS ANIDADOS

- OpenMP puede juntar multiples ciclos anidados en uno solo y entonces particionar el trabajo entre los hilos disponibles, usando la clausula: **collapse**

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < K; i++)
    for (int j = 0; j < M; j++)
      {
        C[i][j] = 0;
        for (int k = 0; k < L; k++)
          C[i][j] += A[i][k] * B[k][j];
      }
```

# DIRECTIVA SECTIONS

- Especifica la ejecución en paralelo de algún bloque de código secuencial
- Pueden llamar diferentes funciones en paralelo

```
#pragma omp sections [clause[[,] clause]...] new-line
{
    [#pragma omp section ]
        structured block
    [#pragma omp section
        structured block ]
    ...
}
```

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            func_a();
        #pragma omp section
            func_b();
    }
}
```

# PARALELISMO ANIDADO

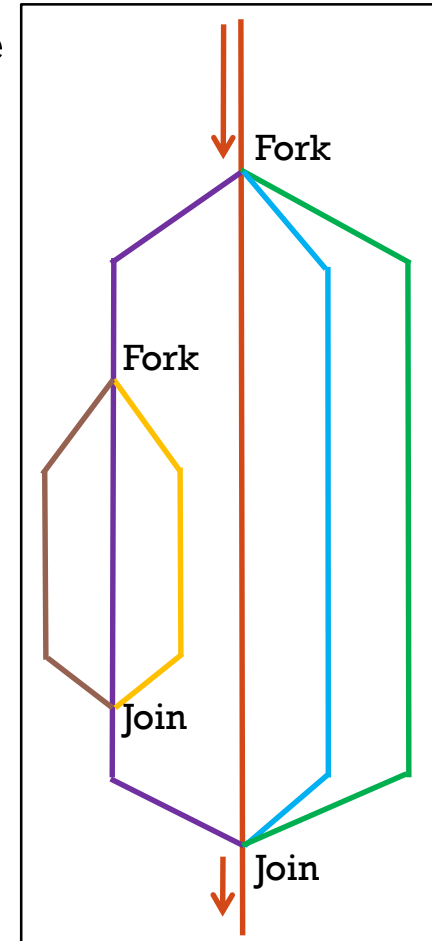
- Podemos agregar una región paralela dentro de otra, habilitando el paralelismo anidado con:

```
omp_set_nested(ban_nested);
```

- `ban_nested=1`, habilita el paralelismo anidado
- `ban_nested=0`, deshabilita el paralelismo anidado

Multiplicación de  
Matrices

```
#pragma omp parallel private(row,col,i)
{
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
        #pragma omp parallel shared(MA, MB, MC, size)
        {
            #pragma omp for schedule(static)
            for (col = 0; col < size; col++) {
                MC[row][col] = 0.0;
                for (i = 0; i < size; i++)
                    MC[row][col] += MA[row][i] * MB[i][col];
            }
        }
    }
}
```



# ACCESO EFICIENTE A LA MEMORIA

- En C/C++, los arreglos están almacenados en la memoria por filas (row-major), mientras que en Fortran se almacenan por columnas (column-major)
- Cuando un elemento del arreglo se transfiere de la memoria RAM a la caché los vecinos de ese elemento (que están en la misma fila, C/C++) también se transfieren a la caché
- Por lo tanto, el orden en que accedemos a los datos en un programa, es de vital importancia para alcanzar un buen rendimiento

```
for (int j=0; j<n; j++)  
    for (int i=0; i<n; i++)  
        sum += a[i][j];
```

```
for (int i=0; i<n; i++)  
    for (int j=0; j<n; j++)  
        sum += a[i][j];
```



# MÉTRICAS (*SPEEDUP*)

- La mejora en el tiempo de ejecución se expresa típicamente como la aceleración o ganancia del rendimiento (*speedup*)

$$speedup = \frac{t_{seq}}{t_{par}}$$

donde  $t_{seq}$  y  $t_{par}$ , son los tiempos que puede tomar una computadora para realizar cierto procesamiento de forma secuencial y paralelo, respectivamente.

- Estos tiempos, podrían estar influenciados por:
  - Habilidad del programador
  - Elección del compilador
  - Habilitar banderas de compilación (-O2, -fast\_math, ...)
  - El Sistema Operativo
  - El tipo del sistema de archivos de los datos
  - La hora del día (Ej. cuando usamos un servidor compartido)

# REGLAS PARA MEDIR LOS TIEMPOS

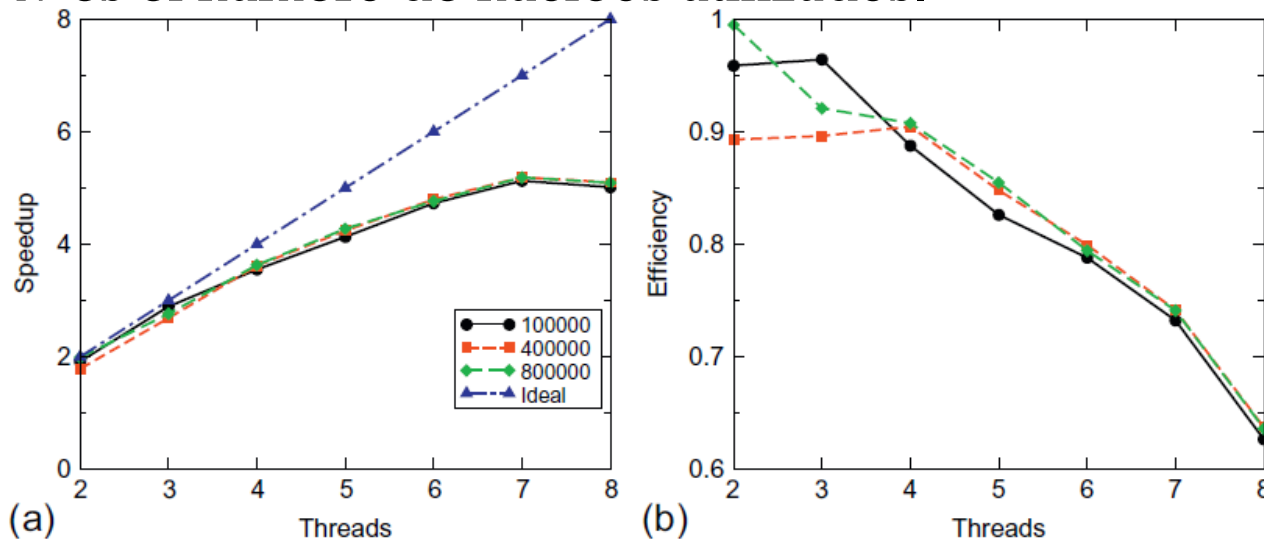
- El programa secuencial y paralelo deben probarse en plataformas de software y hardware idénticas y en condiciones similares
- El programa secuencial debe ser la solución más rápida del problema en cuestión

# MÉTRICAS (EFICIENCIA)

- Formalmente definida como:

$$efficiency = \frac{speedup}{N} = \frac{t_{seq}}{N t_{par}}$$

donde  $N$  es el número de núcleos utilizados.



Curvas de *speedup* y eficiencia para un programa que calcula la integral definida de una función aplicando la regla del trapecio. Estos resultados reportados en [Barlas, Gerassimos], fueron obtenidos en una CPU i7 950 quad-core y promediando 10 ejecuciones.

Barlas, Gerassimos. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.

Cómputo Paralelo, Francisco J. Hernández-López

Ene-Jun 2024

# LEY DE AMDAHL (1967)

$$speedup = \frac{t_{seq}}{t_{par}} = \frac{T}{(1 - \alpha)T + \frac{\alpha T}{N}} = \frac{1}{1 - \alpha + \frac{\alpha}{N}}$$

$T$  tiempo secuencial

$\alpha$  parte del código que puede paralelizarse

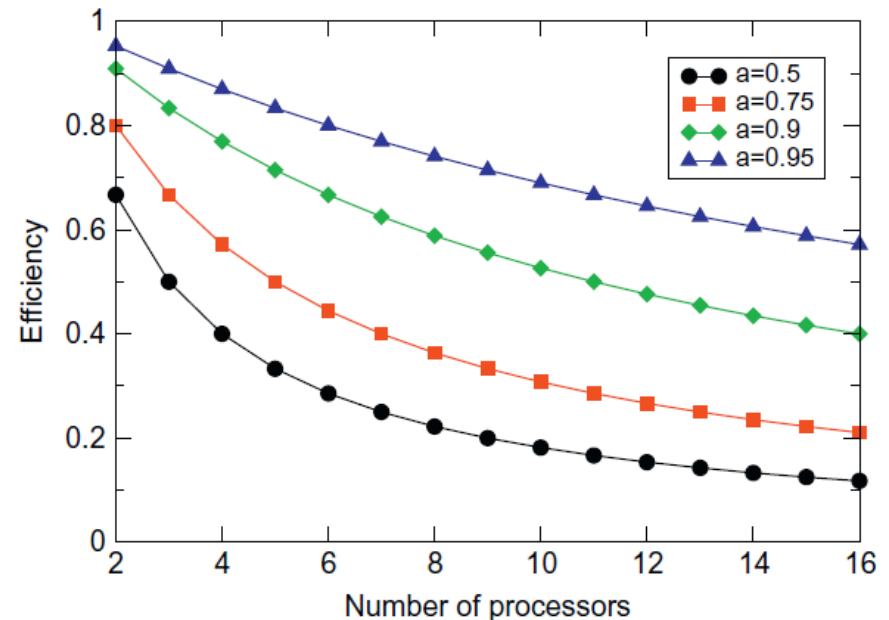
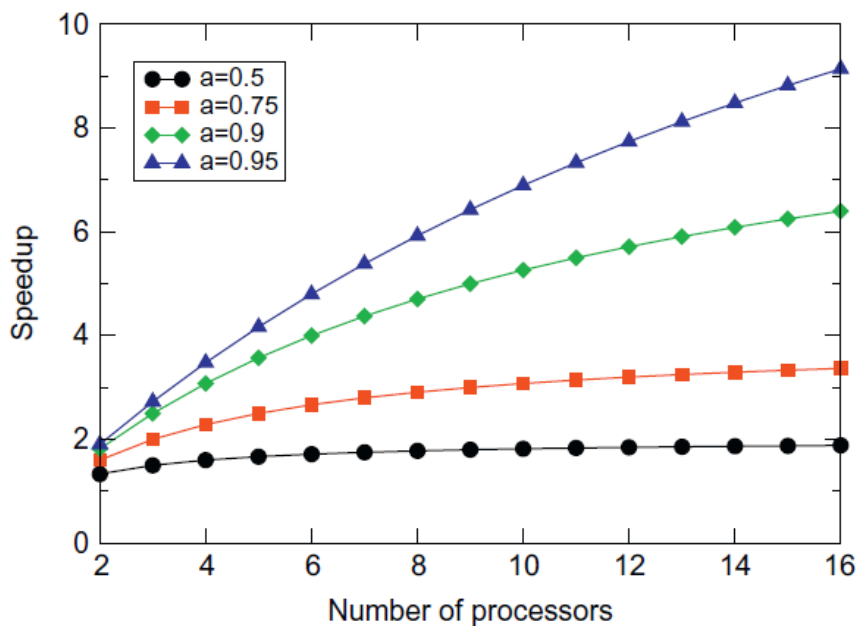
$1 - \alpha$  parte del código que tiene que ejecutarse de forma secuencial

$N$  número de procesadores

- Ignorando los costos de particionamiento, comunicación y coordinación, Cuando  $N \rightarrow \infty$ ,  $speedup = \frac{1}{1 - \alpha}$

# LEY DE AMDAHL (SPEEDUP Y EFICIENCIA)

- Curvas de speedup y eficiencia para diferentes valores de  $\alpha$



# LEY DE GUSTAFSON-BARSIS (1988)

$$speedup = \frac{t_{seq}}{t_{par}} = \frac{(1 - \alpha)T + N\alpha T}{T} = (1 - \alpha) + N\alpha$$

$T$  tiempo que requiere el programa en paralelo

$\alpha$  parte del código que puede paralelizarse

$1 - \alpha$  parte del código que tiene que ejecutarse de forma secuencial

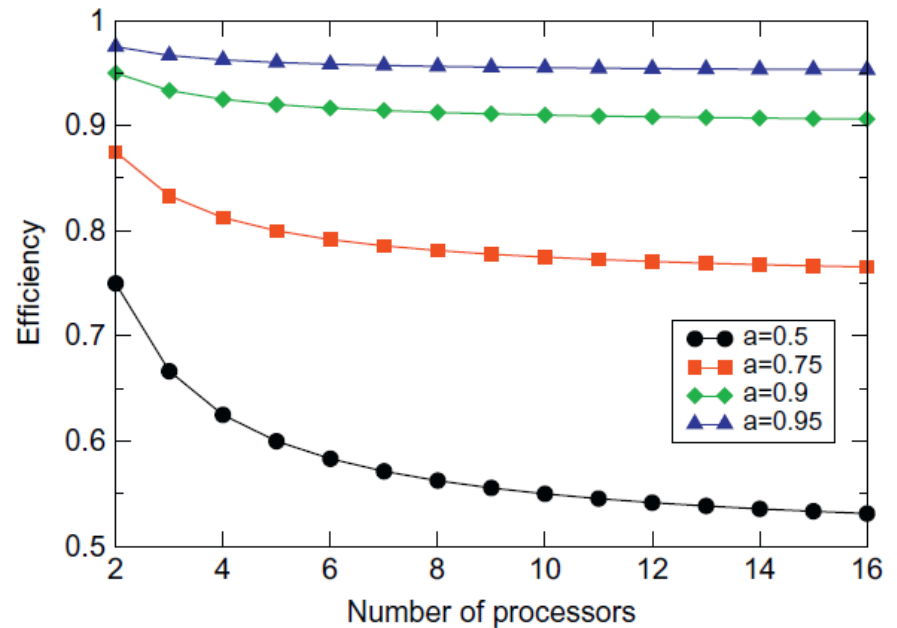
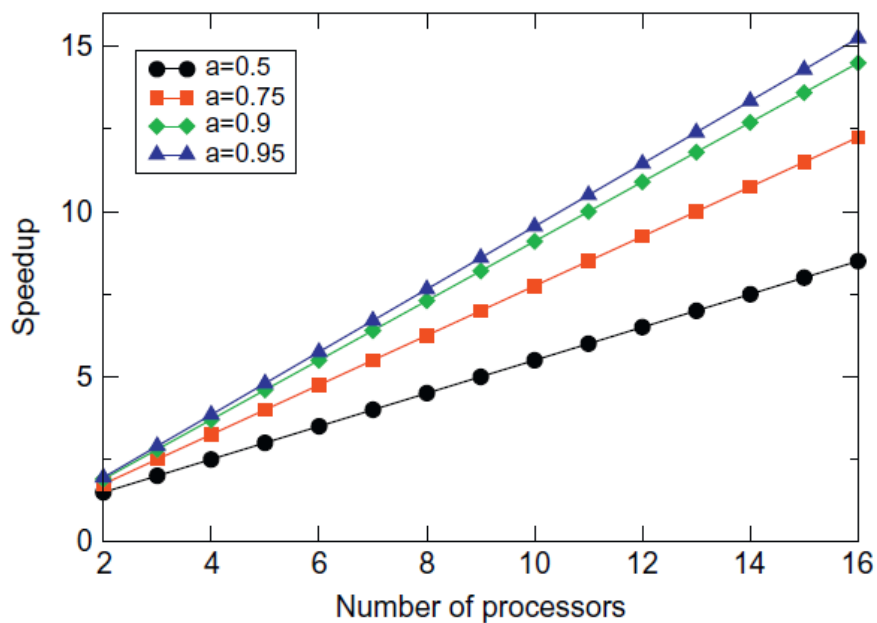
$N$  número de procesadores

$$efficiency = \frac{1 - \alpha}{N} + \alpha$$

- Ignorando los costos de particionamiento, comunicación y coordinación, Cuando  $N \rightarrow \infty$ ,  $efficiency = \alpha$

# LEY DE GUSTAFSON-BARSIS

- Curvas de speedup y eficiencia para diferentes valores de  $\alpha$



# GRACIAS POR SU ATENCIÓN

Francisco J. Hernández-López

[fcoj23@ciimat.mx](mailto:fcoj23@ciimat.mx)

WebPage:

[www.ciimat.mx/~fcoj23](http://www.ciimat.mx/~fcoj23)

