

# SISTEMAS PARALELOS

---

Clase 4 – Programación en memoria compartida // OpenMP

Prof. Dr Enzo Rucci



FACULTAD DE INFORMATICA



UNIVERSIDAD  
NACIONAL  
DE LA PLATA

# Agenda de la clase anterior

- Fundamentos de programación en memoria compartida
- Estándar Pthreads

# Agenda de esta clase

- Estándar OpenMP

# PROGRAMACIÓN EN MEMORIA COMPARTIDA // OPEN MULTI-PROCESSING

---

# Modelo de programación basado en directivas: OpenMP

- Modelos basados en threads → primitivas de bajo nivel.
- Modelos basados en directivas → constructores de alto nivel  
→ Idea básica: liberar al programador del manejo explícito de hilos.
- OpenMP:
  - Es un estándar para programación paralela basado en directivas que puede ser usado con C, C++ y Fortran.
  - Tiene 3 componentes primarios: directivas, funciones de librerías y variables de entorno.
  - Sus directivas proveen soporte para concurrencia, sincronización y manejo de datos obviando el uso explícito de locks, variables condición, alcance de los datos e inicialización de threads.
  - Sus directivas son traducidas a código Pthreads

# Modelo de programación basado en directivas: OpenMP

- OpenMP fue diseñado en 1997 por el consorcio *OpenMP Architecture Review Board* (OpenMP ARB) y aun hoy es mantenido por el mismo.
- La motivación principal radicaba en la dificultad de escribir programas paralelos de gran escala usando las herramientas del momento, como por ejemplo Pthreads → El objetivo era diseñar un estándar para programas de memoria compartida que pudieran ser desarrollados con un mayor nivel de abstracción
- OpenMP sigue una filosofía *incremental* de desarrollo

# Modelo de programación basado en directivas: OpenMP

- Inicialmente, las especificaciones para Fortran y C eran lanzadas en forma separada. A partir del 2005, son lanzadas juntas.

Fecha	Versión
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5
Nov 2018	OpenMP 5.0

# Modelo de programación basado en directivas: OpenMP

- Soporte de compiladores:

Vendedor	Compilador	Soporte
GNU	GCC	Desde GCC 6.1, OpenMP 4.5 es soportado completamente para C y C++. Compilar con <i>-fopenmp</i>
Intel	ICC	Desde versiones 18.0, OpenMP 4.5 es soportado completamente para C, C++ y Fortran. Compilar con <i>-qopenmp</i>

- Más info de soporte en: <http://www.openmp.org/resources/openmp-compilers-tools/>

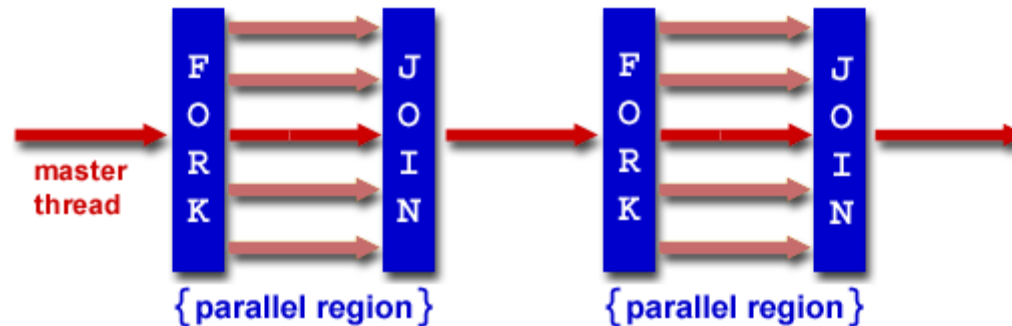


# OpenMP: Características básicas

- Sintaxis de las directivas.

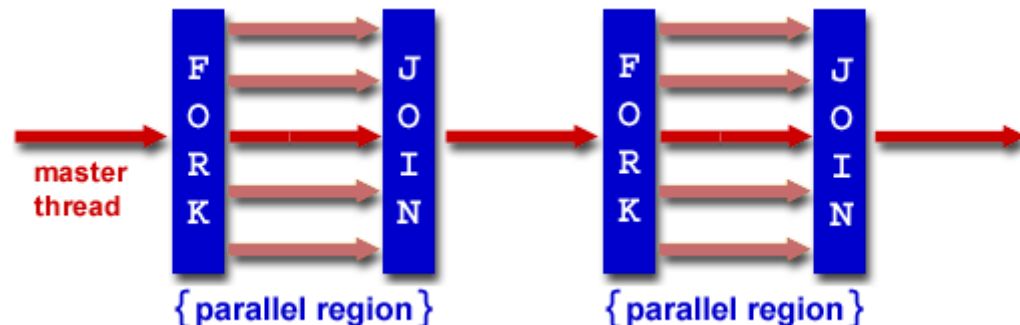
```
#pragma omp nombre_directiva [lista de cláusulas]
```

- Modelo Fork-Join



# OpenMP: Modelo *Fork-Join*

- Comienza con un único hilo (hilo master).
- **Fork**: Al encontrar un constructor paralelo (o directiva paralela), el hilo master crea un grupo de hilos
- El bloque encerrada por el constructor de la región paralela es ejecutada en paralelo entre todos los hilos.
- **Join**: cuando el conjunto de hilos finaliza el bloque paralelo, se sincronizan y terminan, continuando únicamente el hilo master.



# OpenMP: Constructor *parallel*

- Es el constructor más importante
- Permite especificar un bloque de código que será ejecutado en paralelo (*región paralela*)
- Asegura la creación de un equipo de hilos aunque la distribución del trabajo dentro de la región paralela es responsabilidad del programador
- Dentro de la región paralela, cada hilo mantiene un ID único (el ID 0 siempre corresponde al hilo master)
- Al final de la región paralela, hay una barrera implícita → sólo el hilo master continúa con la ejecución

```
#pragma omp parallel [lista de cláusulas]
{ ... }
```

# OpenMP: Constructor *parallel* – Cláusulas *private* y *firstprivate*

- Admite cláusulas que determinan cuáles datos serán privados a cada hilo y cuáles serán compartidos entre todos los hilos de una región paralela
- Las variables privadas de un hilo se especifican mediante la cláusula *private*:

```
#pragma omp parallel private(lista_de_variables)
{ ... }
```

- Esta cláusula crea una copia local a cada hilo de cada variable especificada respetando su tipo y tamaño
- Esta copia local sólo puede ser accedida y modificada por el hilo que la posee
- Variante: *firstprivate*

# OpenMP: Constructor *parallel* – Cláusulas *shared* y *default*

- Las variables que son compartidas entre todos los hilos de un equipo se especifican en la cláusula *shared*:

```
#pragma omp parallel shared(lista_de_variables)
{ ... }
```

- En este caso, todos los hilos podrán leer y modificar la variable original.
- Por defecto, todas las variables son compartidas → Para alterar este comportamiento, se puede emplear la cláusula *default*.

```
#pragma omp parallel default(shared|private|none)
{ ... }
```

# OpenMP: Constructor *parallel* – Cláusula *num\_threads*

- Para especificar el número de hilos a crear, se puede usar la cláusula *num\_threads*:

```
#pragma omp parallel num_threads(T)
{ ... }
```

- En caso de ausencia, el número de hilos a crear lo determina la variable de entorno OMP\_NUM\_THREADS

# OpenMP: Ejemplo de traducción a Pthreads

```
int a, b;
main() {
    [ // serial segment
    [ #pragma omp parallel num_threads (8) private (a) shared (b)
    { [ // parallel segment
    }
    [ // rest of serial segment
}
```

Sample OpenMP program

Code inserted by the OpenMP compiler

```
int a, b;
main() {
    [ // serial segment
    [ for (i = 0; i < 8; i++)
      pthread_create (....., internal_thread_fn_name, ...);
    [ for (i = 0; i < 8; i++)
      pthread_join (.....);
    [ // rest of serial segment
}

void *internal_thread_fn_name (void *packaged_argument) {
    int a;
    [ // parallel segment
}
```

Corresponding Pthreads translation

# OpenMP: ¡Hola Mundo!

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

int main () {
    int nthreads, tid;

    #pragma omp parallel private(tid)
    {   tid = omp_get_thread_num();
        printf("¡Hola Mundo! Soy el hilo = %d\n", tid);
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Número de hilos = %d\n", nthreads);
        }
    }
}
```

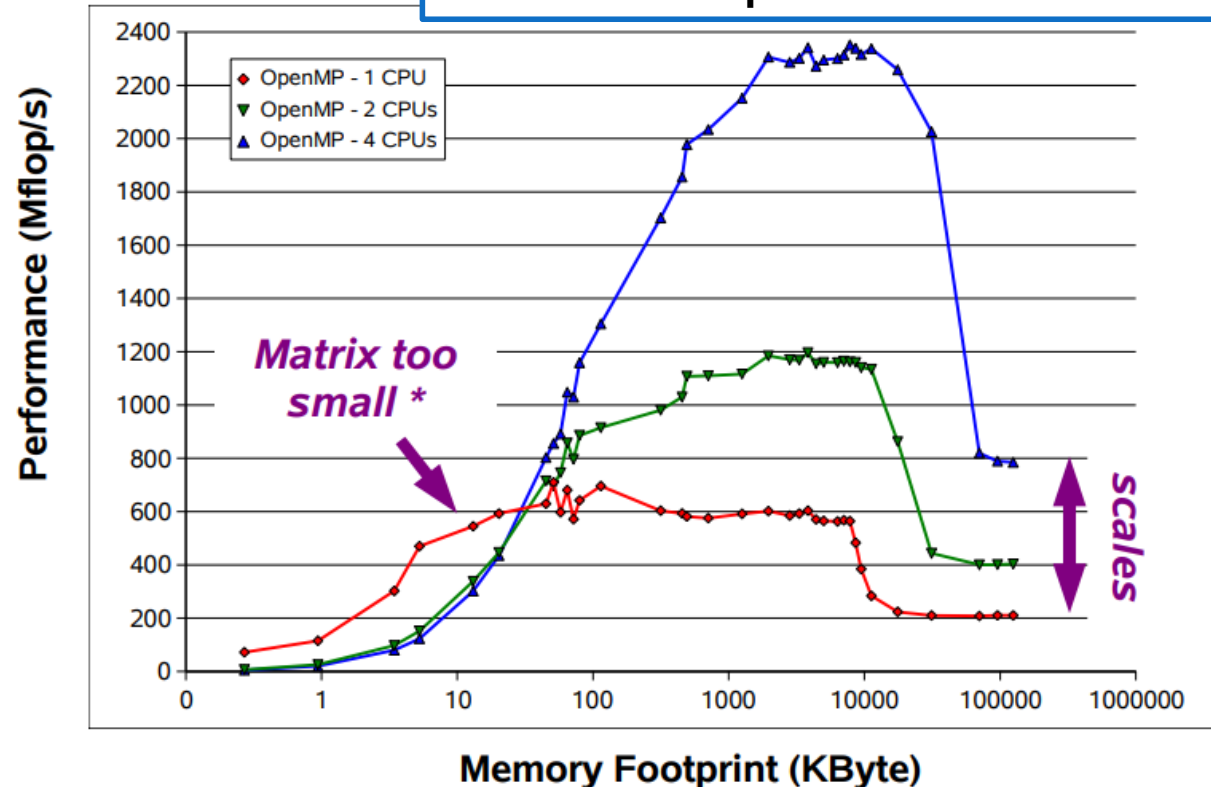
- En este ejemplo, cada hilo imprime su ID mientras que el master también imprime el número de hilos generados



# OpenMP: Constructor *parallel* – Cláusula *if*

- La cláusula *if* permite condicionar la generación de hilos a la evaluación de una expresión escalar
  - Si la evaluación de la expresión resulta falsa, entonces el código se ejecuta en forma secuencial.
  - Puede resultar útil para paralelizar sólo cuando vale la pena.

Programa OpenMP que computa una multiplicación matriz-vector

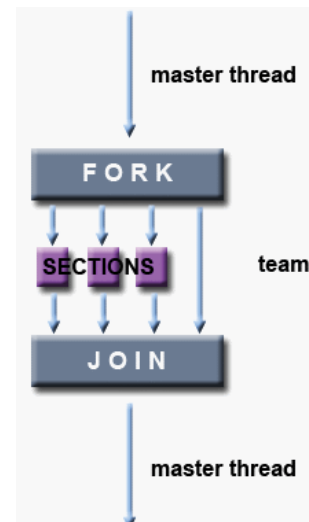
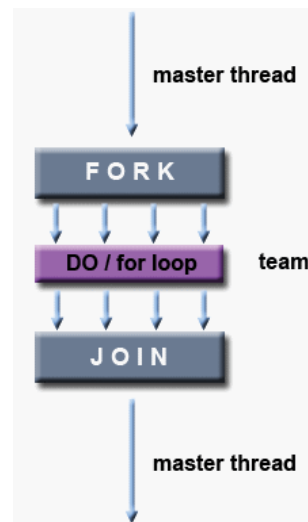


# OpenMP: Constructor *parallel* - ¿Cuánto hilos se generan?

- El número de hilos a generar por un constructor *parallel* está determinado por lo siguientes factores en orden de precedencia:
  1. Evaluación de la cláusula *if*
  2. Inclusión de la cláusula *num\_threads*
  3. Llamado a la función de librería *omp\_set\_num\_threads()*
  4. Seteo de la variable de entorno OMP\_NUM\_THREADS
  5. Decisión de la implementación

# OpenMP: Constructores para trabajo compartido

- La directiva *parallel* puede ser utilizada en conjunto con otras directivas para especificar concurrencia entre iteraciones y tareas (constructores de trabajo compartido) → No crea nuevos threads.
- Diferentes tipos de constructores.
  - Directiva *for* → Divide las iteraciones de un bucle entre los hilos (paralelismo de datos).
  - Directiva *sections* → Divide el trabajo en secciones separadas (paralelismo funcional).



# OpenMP: Constructor *for*

- Sintaxis:

```
#pragma omp for [lista de cláusulas]
for ( init_exp; check_exp; mod_exp)
```

- El uso de esta directiva impone algunas restricciones:
  - Las iteraciones deben ser independientes entre sí
  - El número de iteraciones debe ser conocido de antemano
  - La variable índice se vuelve privada por defecto y no puede ser modificada por los hilos dentro del bucle
  - No se puede usar *break* dentro de las iteraciones
- El bucle paralelo finaliza con una sincronización implícita entre todos los hilos que lo integran

# OpenMP: Constructor *for* – Cláusulas disponibles

- Cláusulas disponibles:
  - *shared*, *private*, *firstprivate*
  - *lastprivate* (lista de variables): funciona como *private*, sólo que la variable original queda con el valor de la última iteración del bucle
  - *reduction* (operador:variable, ...): realiza una operación de reducción usando el operador indicado con las múltiples copias de la variable correspondiente

# OpenMP: Constructor *for* – Cláusula *reduction*

Valid Operators and Initialization Values			
Operation	Fortran	C/C++	Initialization
Addition	+	+	0
Multiplication	*	*	1
Subtraction	-	-	0
Logical AND	.and.	&&	0
Logical OR	.or.		.false. / 0
AND bitwise	iand	&	all bits on / 1
OR bitwise	ior		0
Exclusive OR bitwise	ieor	^	0
Equivalent	.eqv.		.true.
Not Equivalent	.neqv.		.false.
Maximum	max	max	Most negative #
Minimum	min	min	Largest positive #

# OpenMP: Constructor *for* – Cláusula *reduction*

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

#define N 1000

int main () {
    int v[N], i, sum=0;
    /* Inicializar v */
    #pragma omp parallel
    {
        #pragma omp for reduction(+:sum)
        for (i=0; i<N; i++)
            sum += v[i];
    }
    ...
}
```

# OpenMP: Constructor *for* – Cláusula *nowait*

- Cláusulas disponibles:
  - *nowait*: evita la barra implícita al final del bucle

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;

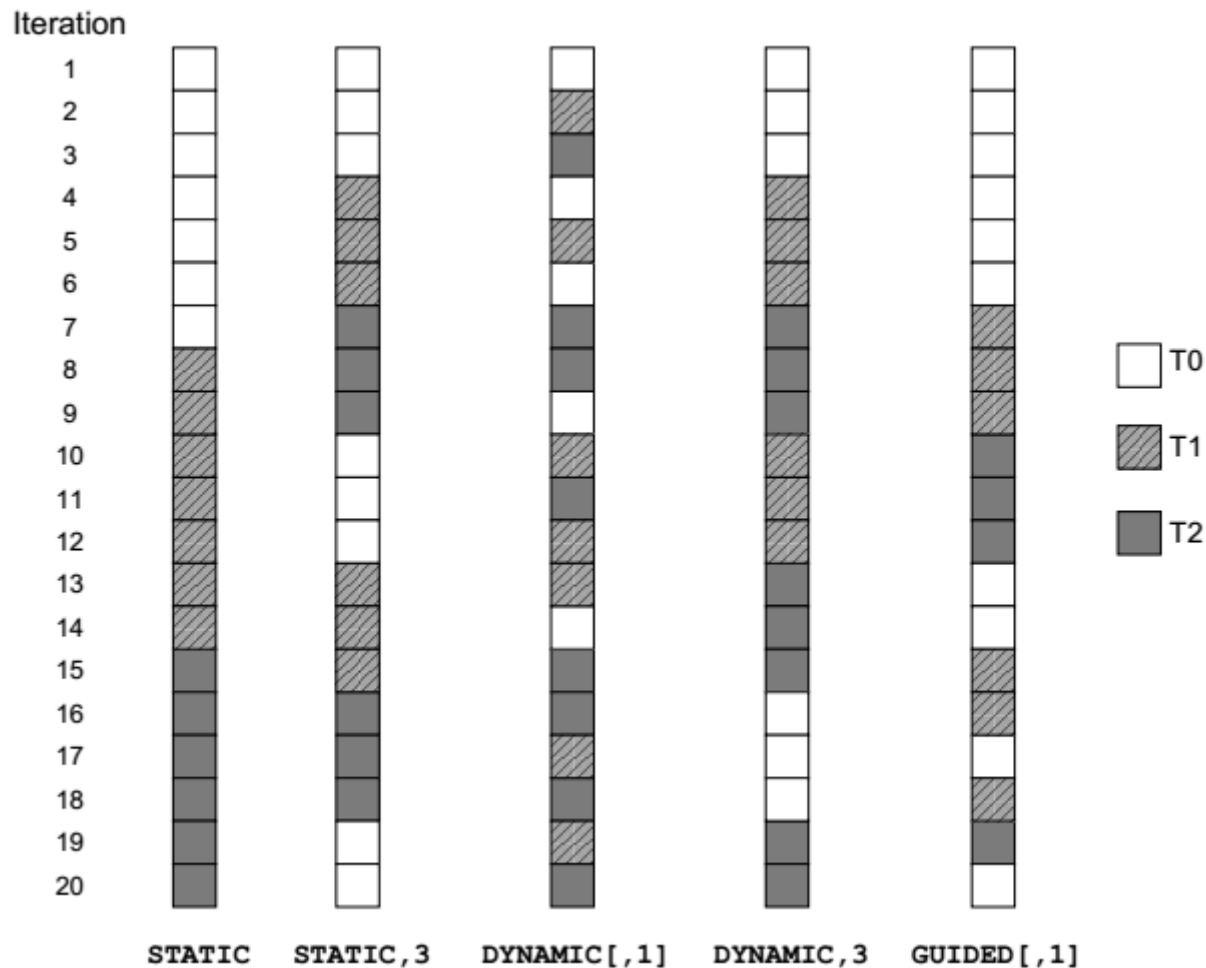
    #pragma omp for nowait
    for (i=0; i<m; i++)
        y[i] = sqrt(z[i]);
}
```



# OpenMP: Constructor *for* – Cláusula *schedule*

- `schedule(política [,chunk])`: especifica cómo se distribuyen las iteraciones entre los hilos.
  - *static*: divide en bloques de *chunk* iteraciones y las asigna en forma *round-robin*. Cuando *chunk* no se especifica, se dividen las iteraciones en bloques de tamaño aproximado.
  - *dynamic*: divide en bloques de *chunk* iteraciones y las asigna bajo demanda. Cuando *chunk* no se especifica, las iteraciones son asignadas de a 1.
  - *guided*: basado en *dynamic* pero decrementando *chunk* a medida que avanza el bucle. Cuando *chunk*=1, el bloque de iteraciones se asigna en forma proporcional a las iteraciones pendiente y los hilos que integran el bucle. Cuando *chunk* =  $k > 1$ , el bloque se asigna de igual manera pero nunca será menor a  $k$ .
  - *auto*: se delega la elección al compilador o al sistema
  - *runtime*: la planificación la determina la variable de entorno `OMP_SCHEDULE`

# OpenMP: Constructor *for* – Cláusula *schedule*



# OpenMP: Constructor *for* – Ejemplo

- Suma de dos vectores

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
{

    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];

}    /* end of parallel region */

}
```

# OpenMP: Constructor *sections*

- Sintaxis:

```
#pragma omp sections [lista de cláusulas]
{
    #pragma omp section
    { // bloque estructurado ... }
    [#pragma omp section]
    { // bloque estructurado ... }
    ...
}
```

- Útil para la distribución de trabajo no-iterativo. Por ejemplo, paralelismo funcional.
- Cada bloque de código indicado por la directiva *section* es independiente de los demás y es ejecutado una sólo vez por un único hilo, pudiendo hacerlo en paralelo con el resto de los hilos.
- Existe una barrera implícita al final de *sections*
- Cláusulas disponibles: *shared*, *private*, *firstprivate*, *lastprivate*, *reduction*, *nowait*

# OpenMP: Constructor *sections* – Ejemplo

- Reducción de  $a$  y  $b$  a suma/producto en  $c/d$ , respectivamente.

```
#pragma omp parallel shared(a,b,c,d) private(i)
{

    #pragma omp sections nowait
    {

        #pragma omp section
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];

        #pragma omp section
        for (i=0; i < N; i++)
            d[i] = a[i] * b[i];

    } /* end of sections */

} /* end of parallel region */
```

# OpenMP: Combinación de directivas

- Las directivas *for* y *sections* se pueden combinar con la directiva *parallel*:

```
#pragma omp parallel default (private) shared (n)
{
    #pragma omp for
    for (i=0; i<n; i++) {
        /* cuerpo del bucle paralelo */
    }
}
```

- Es equivalente a:

```
#pragma omp parallel for default (private) shared (n)
for (i=0; i<n; i++) {
    /* cuerpo del bucle paralelo */
}
```

# OpenMP: Combinación de directivas

- Ambos códigos son equivalentes:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        { tareaA(); }
        #pragma omp section
        { tareaB(); }
        #pragma omp section
        { tareaC(); }
        ...
    }
}
```

```
#pragma omp parallel sections
{
    #pragma omp section
    { tareaA(); }
    #pragma omp section
    { tareaB(); }
    #pragma omp section
    { tareaC(); }
    ...
}
```

- Se pueden usar las cláusulas tanto de *parallel* como de *for* y *sections*.

# OpenMP: Paralelismo anidado

- OpenMP permite habilitar el uso de paralelismo anidado:

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
                                     num_threads(2)
for (i = 0; i < dim; i++) {
    #pragma omp parallel for default(private) shared (a, b, c, dim) \
                                     num_threads(2)
    for (j = 0; j < dim; j++) {
        c[i,j] = 0;
        #pragma omp parallel for default(private) \
                                     shared (a, b, c, dim) num_threads(2)
        for (k = 0; k < dim; k++) {
            c[i,j] += a[i, k] * b[k, j];
        }
    }
}
```

Código OpenMP para  
multiplicar matrices cuadradas

- En este caso, cada directiva *parallel* genera un nuevo equipo de hilos.
- Se requiere que la variable de entorno OMP\_NESTED tenga valor TRUE; de otra forma el código es ejecutado por un único hilo.



# OpenMP: Constructores para sincronización

- OpenMP provee de constructores de alto nivel para diferentes tipo de sincronización:
  - Ejecución serial
  - Barreras
  - Secciones críticas
  - Atomicidad

# OpenMP: Constructor *single*

- Permite que un bloque de código sea ejecutado por un único hilo dentro de una región paralela
- Sintaxis:

```
#pragma omp single [lista de cláusulas]
{ /* bloque estructurado */ }
```

- El bloque es ejecutado por el primer hilo del equipo que llega a ese punto de ejecución; el resto de los hilos espera al final del bloque (hay una barrera implícita).
- Cláusulas disponibles: *private*, *firstprivate*, *nowait*

# OpenMP: Constructor *single* – Ejemplo

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

#define N 1000

int main () {
    float v[N], sum=0, avg;
    int i;
    /* Inicializar v */
    #pragma omp parallel
    {
        #pragma omp for reduction(+:sum)
        for (i=0; i<N; i++)
            sum += v[i];
        #pragma omp single
        { avg = sum / N; }
        ...
    }
    ...
}
```

Promedio de N números

# OpenMP: Constructor *master*

- Es una variante de la directiva *single*: el bloque de código es siempre ejecutado por el hilo master
- Sintaxis:

```
#pragma omp master  
{ /* bloque estructurado */ }
```

- A diferencia de *single*, no hay barrera implícita al final del bloque

# OpenMP: Constructor *barrier*

- Implementa un punto de sincronización global entre todos los hilos de un equipo (*barrera*)
- Sintaxis:

```
#pragma omp barrier
```

- Se debe usar con cuidado:
  - Puede causar deadlock
  - Incide en el rendimiento.

# OpenMP: Constructor *critical*

- Permite implementar regiones críticas en forma sencilla.
- Sintaxis:

```
#pragma omp critical [nombre]
{ /* bloque estructurado */ }
```

- Garantiza que, en cualquier punto de ejecución del programa, a lo sumo un hilo estará dentro de la sección crítica *nombre*.
- Si un hilo alcanza un bloque *critical* y ya hay otro en la misma, el hilo espera a que la sección crítica se libere.
- El nombre es opcional. Si no se especifica uno, entonces se usa un nombre por defecto que es el mismo para todas las secciones críticas que no tengan nombre (no conveniente).

# OpenMP: Constructor *critical* - Ejemplo

- Ejemplo: productor-consumidor con buffer de tamaño ilimitado

```
#pragma omp parallel sections private(task) num_threads (2)
{
    #pragma omp section
    {
        /* productor */
        task = producir ();
        #pragma omp critical
        { insertar_en_buffer(task); }
    }
    #pragma omp section
    {
        /* consumidor */
        #pragma omp critical
        { task = extraer_de_buffer(task); }
        consumir(task);
    }
}
```

# OpenMP: Constructor *atomic*

- Es una variante de la directiva *critical* para secciones críticas de una única instrucción
- Sintaxis:

```
#pragma omp atomic  
x bin_op = expr
```

- Esta directiva analiza la disponibilidad de instrucciones atómicas por hardware, por lo que podría producir mejor rendimiento que *critical*.
- Sin embargo, impone algunas restricciones para su uso:
  - *bin\_op* puede ser un operador aritmético o lógico.
  - Sólo la lectura (*load*) y la escritura (*store*) de *x* son atómicas; la evaluación de *expr* no lo es.
  - *expr* no puede contener una referencia a *x*.
- Uso poco frecuente debido a sus requisitos



# OpenMP: Constructor *ordered*

- Útil para aquellos casos en que resulta necesario ejecutar cierto segmento de código en el mismo orden en que lo haría la versión secuencial.
- Sintaxis:

```
#pragma omp ordered  
{ /* bloque estructurado */ }
```

- Se emplea en el ámbito de una directiva *for* o *parallel for*
  - Requiere incluir cláusula *ordered*

# OpenMP: Constructor *ordered* - Ejemplo

- Ejemplo: cálculo de la suma acumulativa de una lista

```
cumul_sum[0] = list[0];  
  
#pragma omp parallel for private (i) shared (cumul_sum, list, n) ordered  
for (i = 1; i < n; i++) {  
  
    /* otro procesamiento con list[i] */  
  
    #pragma omp ordered  
    { cumul_sum[i] = cumul_sum[i-1] + list[i]; }  
    ...  
}
```

- Tener en cuenta que la directiva *ordered* representa un punto de serialización en la ejecución → el bloque de código debe contener la mínima cantidad de instrucciones posibles
- Sólo tiene sentido si los hilos realizan trabajo significativo fuera del constructor *ordered*

# OpenMP: Directiva *flush*

- OpenMP adopta un modelo relajado de consistencia de memoria
  - Las variables suelen ser actualizadas en los registros o en la memoria caché, demorando su modificación en la memoria principal.
  - Si bien esto puede mejorar el rendimiento, también puede provocar una vista inconsistente de la memoria para un hilo
- La directiva *flush* representa un punto de sincronización de la memoria:
  - Todas las escrituras pendientes en memoria principal serán asentadas
  - Todas las lecturas pendientes serán realizadas desde memoria principal
- Sintaxis:

```
#pragma omp flush [(lista de variables)]
```

# OpenMP: Directiva *flush*

- No suele ser muy usada ya que muchos de las directivas OpenMP incluyen un *flush* implícito:
  - En la directiva *barrier*;
  - a la entrada y a la salida de *critical*, *ordered*, *parallel*, *parallel for* y *parallel sections*;
  - y a la salida de las directivas *for*, *sections* y *single*.
- Excepciones: la cláusula *nowait* excluye a *flush*; tampoco está presente a la entrada de *for*, *sections* y *single*; ni a la entrada o salida de *master*.

# OpenMP: Funciones de librería

- Además de las directivas, OpenMP soporta una serie de funciones que permite al programador controlar la ejecución del programa con mayor nivel de abstracción que Pthreads.
- Funciones básicas:

```
void omp_set_num_threads (int num_threads);
```

*Setea el valor de la variable de entorno OMP\_NUM\_THREADS, determinando el número de hilos que serán generando en las siguientes regiones paralelas que no especifiquen la clausula num\_threads*

```
int omp_get_num_threads ();
```

*Retorna el número de hilos de la región paralela actual*

# OpenMP: Funciones de librería

- Más funciones básicas:

```
int omp_get_max_threads ();
```

*Retorna la cantidad máximo de hilos que podrían generarse en una región paralela*

```
int omp_get_thread_num ();
```

*Retorna el ID del hilo que la invocó dentro de la región paralela actual*

```
int omp_get_num_procs ();
```

*Retorna el número de procesadores disponibles en el sistema*

```
int omp_in_parallel();
```

*Retorna True si el hilo que la invocó está dentro una región paralela; Falso en caso contrario*

# OpenMP: Funciones de librería

- Funciones para controlar y monitorizar la creación de hilos:

```
void omp_set_dynamic (int dynamic_threads);
```

*Habilita o inhabilita el ajuste dinámico del número de hilos a generar en las próximas regiones paralelas*

```
int omp_get_dynamic ();
```

*Retorna True si el ajuste dinámico del número de hilos está habilitado; Falso en caso contrario*

```
void omp_set_nested (int nested);
```

*Habilita o inhabilita el paralelismo anidado*

```
int omp_get_nested ();
```

*Retorna True si el paralelismo anidado está habilitado; Falso en caso contrario*

# OpenMP: Funciones de librería

- Funciones para controlar la planificación de hilos:

```
void omp_set_schedule(omp_sched_t kind, int chunk);
```

*Permite configurar dinámicamente la planificación a usar cuando se especifica runtime en la clausula schedule*

```
void omp_get_schedule(omp_sched_t * kind, int * chunk);
```

*Retorna la planificación que se usará cuando se especifica runtime en la clausula schedule*



# OpenMP: Funciones de librería

- OpenMP también ofrece funciones para el uso de *locks*, para aquellos casos en que las directivas *critical* y *atomic* no sean suficientes/convenientes.
- El tipo de dato para los *locks* es *omp\_lock\_t* y las funciones disponibles son:

```
void omp_init_lock (omp_lock_t *lock);  
void omp_destroy_lock (omp_lock_t *lock);  
void omp_set_lock (omp_lock_t *lock);  
void omp_unset_lock (omp_lock_t *lock);  
int omp_test_lock (omp_lock_t *lock);
```

- Funcionan en forma equivalente a los *locks* de Pthreads

# OpenMP: Funciones de librería

- OpenMP también ofrece funciones para exclusión mutua recursiva.
- El tipo de dato para los *locks* de esta clase es *omp\_nest\_lock\_t* y las funciones disponibles son:

```
void omp_init_nest_lock (omp_nest_lock_t *lock);  
void omp_destroy_nest_lock (omp_nest_lock_t *lock);  
void omp_set_nest_lock (omp_nest_lock_t *lock);  
void omp_unset_nest_lock (omp_nest_lock_t *lock);  
int omp_test_nest_lock (omp_nest_lock_t *lock);
```

- Funcionan en forma equivalente a los *locks* recursivos de Pthreads

# OpenMP: Variables de entorno

- OpenMP también cuenta con un conjunto de variables de entorno para ayudar a controlar la ejecución del programa paralelo.
- OMP\_NUM\_THREADS: especifica la cantidad de hilos por defecto que se crearán.
- OMP\_DYNAMIC: determina si el número de hilos puede ser modificado en forma dinámica.
- OMP\_NESTED: especifica si se permite el paralelismo anidado.
- OMP\_SCHEDULE: planificación para cuando la cláusula *schedule* es *runtime*.

# OpenMP 3.0: *Tasking*

- El uso de tareas fue introducido en la versión 3.0 (principal cambio).
- Una **tarea** es una unidad de trabajo (porción de código) cuya ejecución *puede* ser diferida en el tiempo. Se compone de:
  - Código a ejecutar
  - Entorno de datos
  - Variables de control internas
- Pensado para paralelizar problemas irregulares:
  - Bucles *while*
  - Bucles *for* que no tienen una cantidad conocida de iteraciones
  - Algoritmos recursivos
  - Entre otros
- En realidad, OpenMP < 3.0 ya soportaba el uso de tareas aunque de manera implícita. Por ejemplo, al ejecutar un constructor *parallel*.

# OpenMP 3.0: Constructor *task*

- El programador identifica las tareas encerrando los bloques de código correspondientes bajo la directiva *task* → Se asume que todas las tareas son independientes entre sí
- Sintaxis:

```
#pragma omp task [lista de cláusulas]
{ /* bloque estructurado */ }
```

- Cuando un hilo encuentra un constructor *task*, el sistema de ejecución genera una nueva tarea
- El momento en que esta tarea se ejecute dependerá del sistema de ejecución, el cual puede ser inmediato o diferido
- Se permite el anidamiento de tareas → Una tarea puede generar otras tareas

# OpenMP 3.0: Constructor *task*

- Cláusulas disponibles:
  - *shared, private, firstprivate, default*
  - *untied*: por defecto la tarea es ejecutada de inicio a fin por un mismo hilo (no necesariamente el que la generó).  
*untied* permite que la tarea pueda ser completada por más de un hilo.
  - *if (expresión)*: evalúa la expresión.
    - Si el resultado es verdadero, se genera una tarea.
    - Si el resultado es falso, se ejecuta el código inmediatamente

# OpenMP 3.0: Constructor *task* – Sincronización de tareas

- Barreras para hilos (explícitas o implícitas) → Todas las tareas generadas por un hilo de un equipo deben haberse completado para que el hilo pueda superar la barrera
- Barreras para tareas → Específicas para un hilo de un equipo. Sintaxis:

```
#pragma omp taskwait
```

- Al llegar a una directiva *taskwait*, el hilo se suspende hasta que todas sus tareas hijas se hayan completado (sólo considera las hijas, no sus descendientes).

# OpenMP 3.0: Constructor *task* – Ejemplo

```
#pragma omp parallel num_threads (T)  
{  
  
  #pragma omp task  
  foo();  
  
  #pragma omp barrier  
  
  #pragma omp single  
  {  
    #pragma omp task  
    bar();  
  }  
}
```

Genera T tareas foo()

Todas las tareas foo() deben haber terminado en este punto

Sólo se genera una tarea bar()

La tarea bar() debe haber terminado en este punto



# OpenMP 3.0: Constructor *task* – Reglas de alcance

- Si la cláusula *default* no fue especificada, entonces:
  - Las variables no especificadas son *firstprivate* por defecto (esto es así para garantizar su posible ejecución diferida).
  - Las variables que fueron especificadas como *shared* en la directiva inmediatamente anterior, mantienen su condición.

```
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A es *shared*  
B es *firstprivate*  
C es *private*

# OpenMP 3.0: Constructor *task* – Ejemplo: recorrido de una lista

```
List l = cargar_lista();  
while (l) {  
    procesar(l->dato);  
    l = l->next;  
}
```

- Recorrido clásico de una lista
- Se debe hacer algún tipo de procesamiento con cada elemento de la lista
- El procesamiento de cada elemento es independiente de los demás
- No puede resolverse con un *parallel for*

# OpenMP 3.0: Constructor *task* – Ejemplo: recorrido de una lista

```
#pragma omp parallel  
#pragma omp single  
{   List l = cargar_lista();  
    while (l) {  
        #pragma omp task  
        { procesar(l->dato); }  
        l = l->next;  
    }  
}
```

# OpenMP 3.0: Constructor *task* – Ejemplo: sucesión de Fibonacci

```
int fib ( int n ) {  
    int x,y;  
    if ( n < 2 ) return n;  
    x = fib(n-1);  
    y = fib(n-2);  
    return x+y;  
}
```

- Sucesión infinita de números naturales que comienza con los números 1 y 1, y a partir de ellos, cada término se obtiene sumando los dos anteriores
- Algoritmo recursivo para calcular la sucesión de Fibonacci
- No puede resolverse con un *parallel for*.

# OpenMP 3.0: Constructor *task* – Ejemplo: sucesión de Fibonacci

```
int fib ( int n ) {  
    int x,y;  
    if ( n < 2 ) return n;  
    #pragma omp task shared(x)  
    x = fib(n-1);  
    #pragma omp task shared(y)  
    y = fib(n-2);  
    #pragma omp taskwait  
    return x+y;  
}
```

```
int main () {  
    int n=30;  
    #pragma omp parallel shared(n)  
    {  
        #pragma omp single  
        printf ("fib(%d) = %d\n", n, fib(n));  
    }  
}
```

# OpenMP 3.0: *Tasking* - Resumen

- Deducir el alcance de una variable puede resultar difícil:
  - Las reglas por defecto difieren de la de otros constructores
  - Usar *default(none)* es recomendable
- Usar *tasking* para lo que fue pensado; no para paralelismo que OpenMP soporta adecuadamente
  - Por ejemplo, bucles estándar
  - *Tasking* tiene un overhead mayor
- El rendimiento de estos programas depende del sistema de ejecución
  - En general, se obtienen mejores resultados cuando el usuario es capaz de controlar el número y la granularidad de las tareas

# OpenMP 4.0: Nuevas características

- Soporte para aceleradores, como GPUs y Xeon Phi
  - Constructores *device*, *host device*, *target device*, ...
- Soporte para vectorización guiada
  - Constructor *simd*
- Cancelación de hilos
  - Constructor *cancel*
- Afinidad de hilos
  - Mapeo dinámico de hilos a núcleos
- Más en [www.openmp.org](http://www.openmp.org)

# Bibliografía usada para esta clase

- OpenMP. Sitio oficial. [www.openmp.org](http://www.openmp.org)
- OpenMP tutorial. Blaise Barney, Lawrence Livermore National Laboratory. <https://computing.llnl.gov/tutorials/openMP/>
- Capítulo 7, An Introduction to Parallel Computing. Design and Analysis of Algorithms (2da Edition). Grama A., Gupta A., Karypis G. & Kumar V. (2003) Inglaterra: Pearson Addison Wesley.
- Capítulo 6, Parallel Programming for Multicore and Cluster Systems. Rauber, T. & Rünger, G. (2010). EEUU: Springer-Verlag Berlin Heidelberg.
- Capítulo 6 y 7, Introduction to HPC for Scientists and Engineers. Hager, G. & Wellein, G. (2011) EEUU: CRC Press.
- Using OpenMP – Portable Shared Memory Parallel Programming. Chapman, B., Jost, G. & Van der Pas (2008). UK: MIT Press.