

Algoritmos Paralelos

(clase 27.10.15)

Prof. J.Fiestas

OpenMP (Open Multi-Processing)

API (Application Programming Interface), es el conjunto de subrutinas, funciones y procedimientos (o métodos en la programación orientada a objetos) que ofrece una cierta biblioteca para ser utilizada por otro software como una capa de abstracción.

Month/Year	Version
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

OpenMP (Open Multi-Processing)

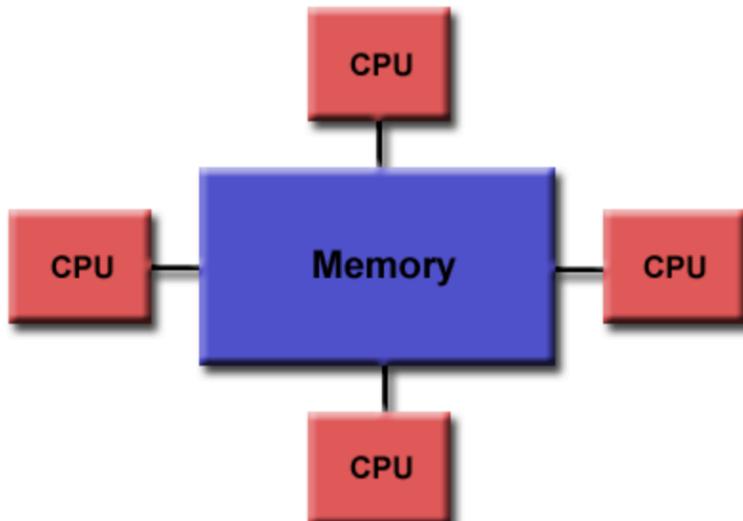
OPENMP es un API, que provee un modelo escalable, portable para aplicaciones de memoria compartida. Es aplicable en C/C++ y Fortran, en diferentes arquitecturas.

OPENMP utiliza directivas de compilador, librerias y variables de entorno

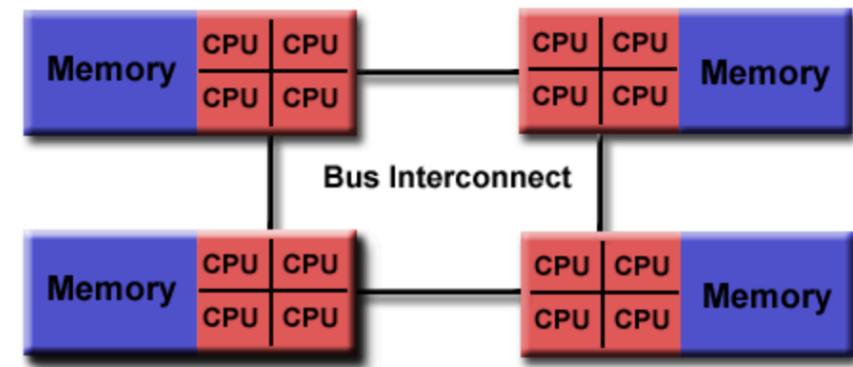
Month/Year	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0

OpenMP esta diseñado para maquinas multi-procesos y de memoria compartida (UMA, o NUMA)

OpenMP paraleliza usando hilos de ejecución o subprocessos (*threads*): unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo



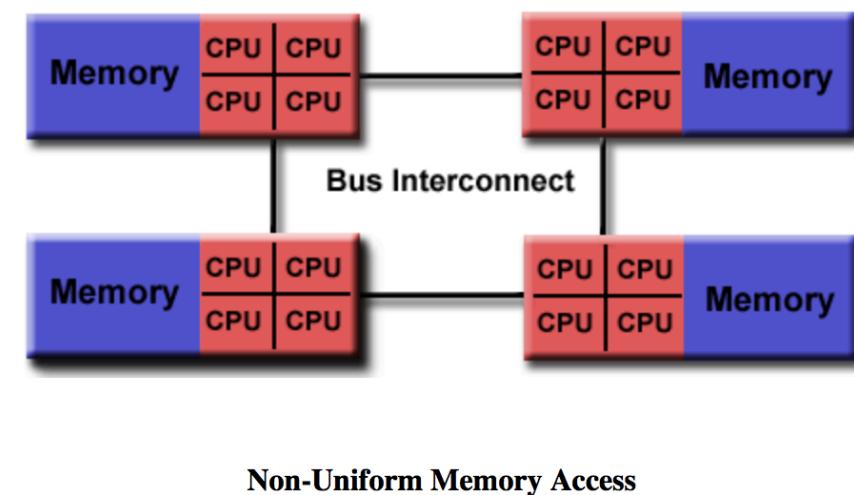
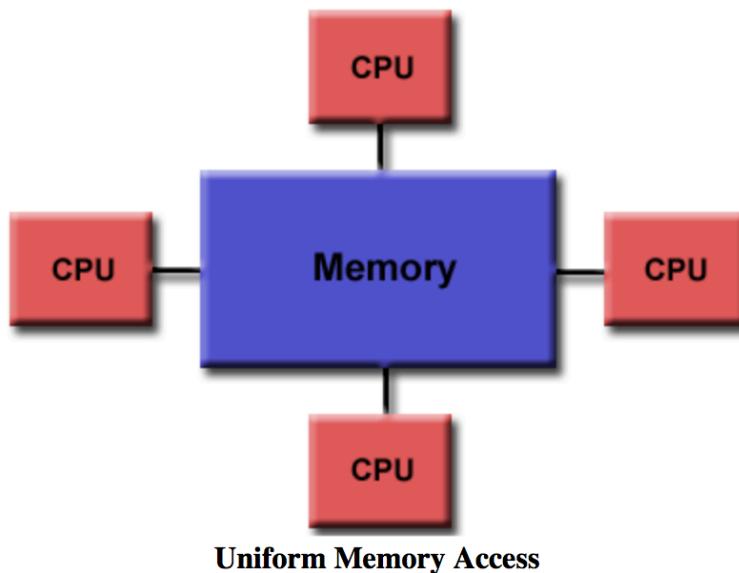
Uniform Memory Access



Non-Uniform Memory Access

La creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas concurrentemente.

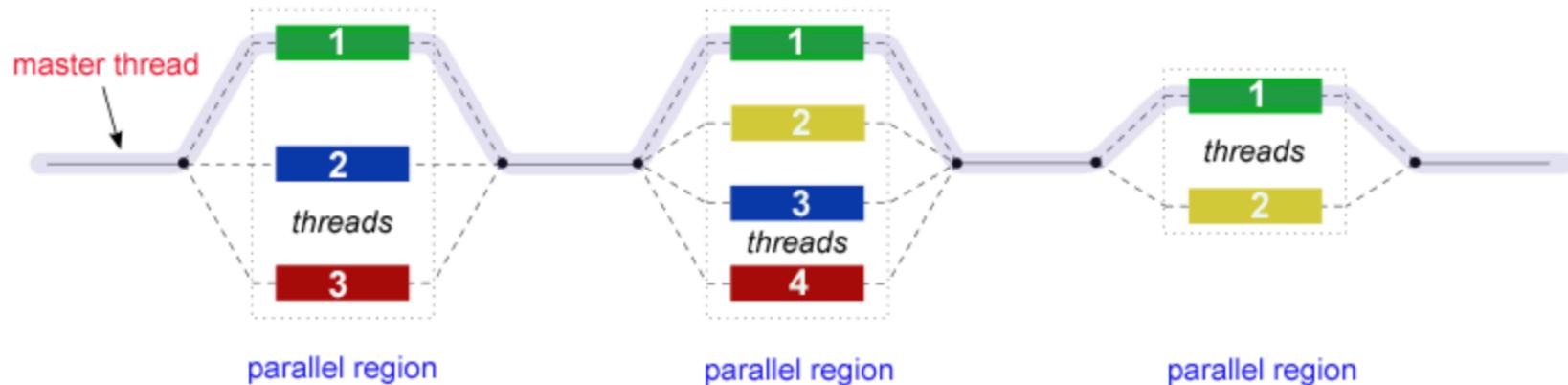
Threads existen solo ligados a procesos, y son típicamente igual al número de núcleos del procesador (sin embargo, el usuario define el número de threads)



Paralelismo en OpenMP es explícito, dando total control sobre el proceso. Puede implementarse añadiendo directivas a un proceso en serie, o insertando subrutinas para lograr niveles múltiples de paralelismo

Modelo Fork-Join:

Los procesos se inician como un proceso simple: *master thread*, que ejecuta secuencialmente hasta que se encuentra la primera región en paralelo.

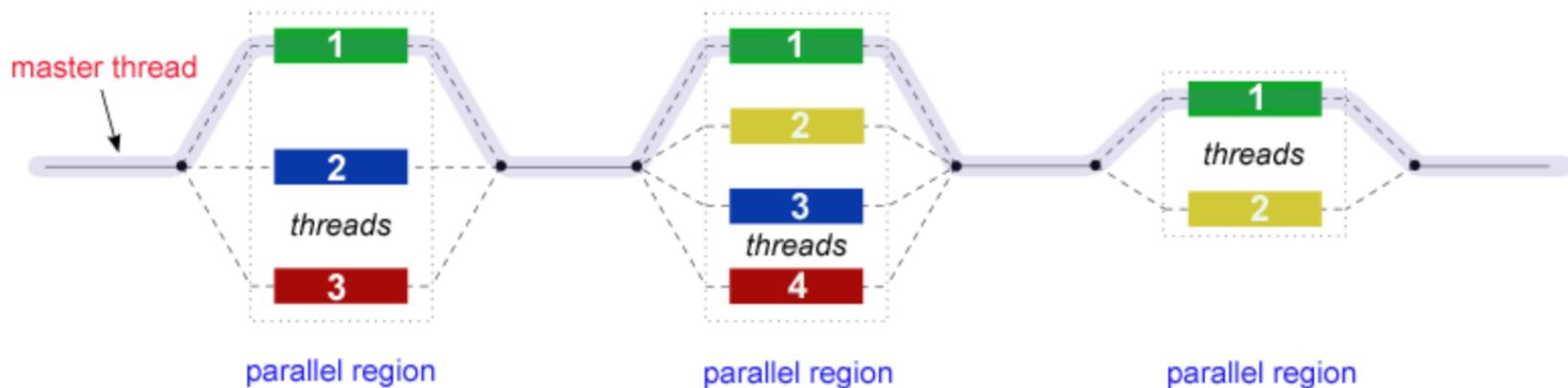


Modelo Fork-Join:

Los procesos se inician como un proceso simple: *master thread*, que ejecuta secuencialmente hasta que se encuentra la primera region en paralelo.

FORK: el *master thread* crea un grupo de *threads* en paralelo, que son ejecutados.

JOIN: cuando los *threads* completan su ejecución, se sincronizan y terminan, dejando solo el *master thread*



Directivas de compilación:

Son comentarios sintácticos en el código fuente y serán ignorados por el compilador si el usuario así lo decide (flag de compilación)

Estas inician una región de paralelismo en el código, dividen áreas del código entre *threads*, distribuyen iteraciones entre *threads*, serializan regiones del código, y sincronizan el trabajo entre *threads*.

```
#pragma omp parallel default(shared) private(beta,pi)
```



Nombre de la directiva



cláusulas

Variables de entorno:

Controlan la ejecución en paralelo en tiempo real, como:

Número de threads, cómo estan separadas las iteraciones, asignar threads a procesos, definiendo paralelismo entrelazado, threads dinámicos,

```
csh/tcsh  setenv OMP_NUM_THREADS 8  
sh/bash  export OMP_NUM_THREADS=8
```

Librerías:

Se utilizan para especificar y alinear el número de *threads*, y sus identificadores, y los *threads* dinámicos, paralelismo entrelazado, y controladores de tiempos de reloj.

```
#include <omp.h>
int omp_get_num_threads(void)
```

Estructura de OpenMP en C++

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

```
main () {
```

Código en serie

Sección en paralelo. Divide (fork) un grupo de hilos (threads).

Define tipos de variables

```
#pragma omp parallel private(var1, var2) shared(var3) {
```

Región paralela ejecutada por todos los hilos

.....

Los hilos se reúnen con el maestro

```
}
```

Continúa el código en serie

.....

```
}
```

Compiladores de OpenMP:

Compiler	Version	Supports
Intel C/C++, Fortran	14.0.3	OpenMP 3.1
GNU C/C++, Fortran	4.4.7	OpenMP 3.0
PGI C/C++, Fortran	8.0.1	OpenMP 3.0
IBM Blue Gene C/C++	12.1	OpenMP 3.1
IBM Blue Gene Fortran	14.1	OpenMP 3.1
IBM Blue Gene GNU C/C++, Fortran	4.4.6	OpenMP 3.0

Compiler / Platform	Compiler	Flag
Intel Linux Opteron/Xeon	<code>icc</code> <code>icpc</code> <code>ifort</code>	<code>-openmp</code>
PGI Linux Opteron/Xeon	<code>pgcc</code> <code>pgCC</code> <code>pgf77</code> <code>pgf90</code>	<code>-mp</code>
GNU Linux Opteron/Xeon IBM Blue Gene	<code>gcc</code> <code>g++</code> <code>g77</code> <code>gfortran</code>	<code>-fopenmp</code>
IBM Blue Gene	<code>bgxlC_r</code> , <code>bgcc_r</code> <code>bgxlC_r</code> , <code>bgxlC++_r</code> <code>bgxlC89_r</code> <code>bgxlC99_r</code> <code>bgxlF_r</code> <code>bgxlF90_r</code> <code>bgxlF95_r</code> <code>bgxlF2003_r</code>	<code>-qsmp=omp</code>
	*Be sure to use a thread-safe compiler - its name ends with <code>_r</code>	

Construcción de regiones en paralelo:

Regiones que serán ejecutadas en multiples *threads*, al encontrar la directiva PARALLEL

```
#pragma omp parallel [clause ...]
newline
    if (scalar_expression)
        private (list)
        shared (list)
        default (shared | none)
        firstprivate (list)
        reduction (operator: list)
        copyin (list)
        num_threads (integer-expression)

    structured_block
```

Construcción de regiones en paralelo:

El master es el *thread* con el número 0, a partir del cual, el código es duplicado y todos los *threads* lo ejecutarán.

Luego del final de la sección en paralelo, el master *thread* continúa la ejecución

```
#pragma omp parallel [clause ...]
newline
    if (scalar_expression)
        private (list)
        shared (list)
        default (shared | none)
        firstprivate (list)
        reduction (operator: list)
        copyin (list)
        num_threads (integer-expression)

    structured_block
```

```
#include <omp.h>
main () {
int nthreads, tid;
/* Fork a team of threads with each thread having a private tid
variable */
#pragma omp parallel private(tid) {
/* Obtain and print thread id */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
/* Only master thread does this */
if (tid == 0) {
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} /* All threads join master thread and terminate */
}
```

Constructores de trabajo compartido:

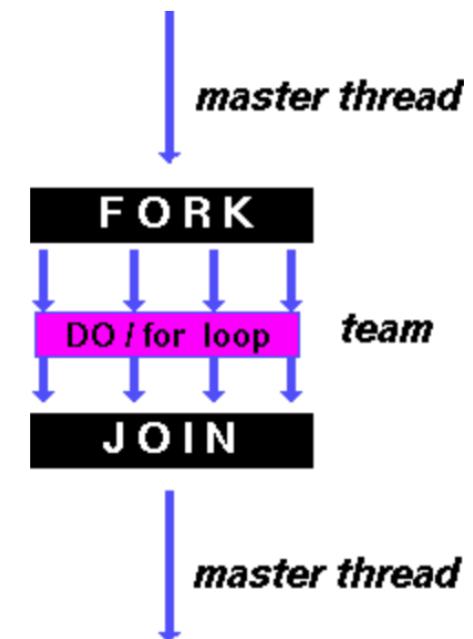
Dividen la ejecución de la región comprendida entre los miembros . Este no crea nuevos *threads*

Constructor Do/for:

Comparte iteraciones en un loop entre los procesos (*data parallelism*)

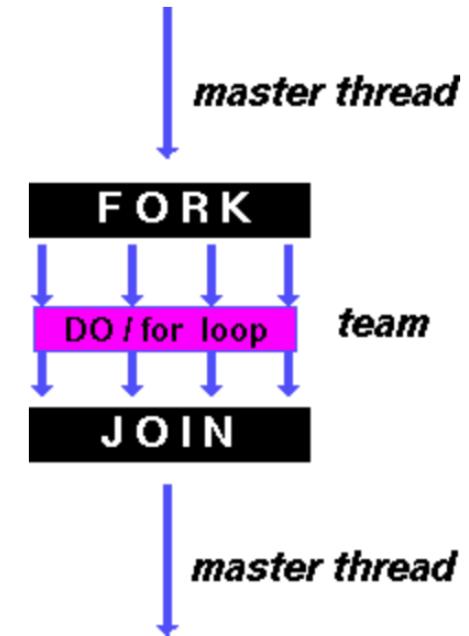
SCHEULE: cláusulas de ordenamiento de las iteraciones entre procesos

- **STATIC:** iteraciones son divididas en bloques de tamaño *chunk*, y añadidos a threads estáticamente. Si *chunk* no es dado, las iteraciones son divididas igualmente.



SCHEDULE: cláusulas de ordenamiento de las iteraciones entre procesos

- **DYNAMIC:** iteraciones son divididas en bloques de tamaño *chunk*, y añadidos a threads dinámicamente
- **GUIDED:** como DYNAMIC, pero cada bloque disminuye su tamaño cada vez que es asignado a un thread
- **RUNTIME:** la decisión de ordenamiento se hace en tiempo real por **OMP_SCHEDULE** (*chunk* no se define aquí)
- **AUTO:** el ordenamiento se delega al compilador



NO WAIT / nowait: si especificada, *threads* no se sincronizan al final del loop en paralelo

ORDERED: especifica que las iteraciones del loop deben ser ejecutadas como en una secuencia serial

```
#include <omp.h>
#define CHUNKSIZE 100
#define N    1000
main ()
{
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
#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 section */
}
```

COLLAPSE: especifica cuantos loops en un loop entrelazado deben colapsar en una larga iteración y divididos de acuerdo a SCHEDULE

```
#include <omp.h>
#define CHUNKSIZE 100
#define N    1000
main ()
{
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
#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 section */
}
```

SECTIONS:

Especifica que secciones del código se dividirán entre los threads. Cada SECTION se ejecuta una vez por un thread, secciones distintas se ejecutarán por threads distintos

Hay una barrera implícita al final de cada sección, a menos que se utilice NOWAIT/nowait

Nota:
no usar **goto** en las secciones

```
#pragma omp sections [clause ...] newline
    private (list)
    firstprivate (list)
    lastprivate (list)
    reduction (operator: list)
    nowait
{
    #pragma omp section  newline
        structured_block
    #pragma omp section  newline
        structured_block
}
```

Ejemplo de directiva SECTIONS

```
#include <omp.h>
#define N 1000
main ()
{ int i;
float a[N], b[N], c[N], d[N];
/* Some initializations */
for (i=0; i < N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35; }
#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 section */
}
```

SINGLE:

Especifica que la región del código incluída debe ser ejecutada por un solo thread.

Se usa, por ejemplo, cuando secciones no son consideradas seguras, como I/O

Threads que no ejecuten SINGLE, esperan hasta el final del bloque, a menos que se use NOWAIT/nowait

```
#pragma omp single [clause ...] newline
    private (list)
    firstprivate (list)
    nowait
structured_block
```

Constructores combinados:

Pueden combinarse en una sola directiva **PARALLEL**, seguida de directivas de trabajo compartido

```
#include <omp.h>
#define N    1000
#define CHUNKSIZE 100

main () {
int i, chunk;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel for \
shared(a,b,c,chunk) private(i) \
schedule(static,chunk)
for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

TASK:

Define una tarea, que puede ser ejecutada por el thread que la encuentre, o diferida a otro thread. Su ejecución puede ser controlada por directivas.

```
#pragma omp task [clause ...] newline
    if (scalar expression)
    final (scalar expression)
    untied
    default (shared | none)
    mergeable
    private (list)
    firstprivate (list)
    shared (list)
```

structured_block

Constructores de sincronización:

Sincronización es muchas veces necesaria entre *threads* en competencia de ejecución

MASTER:

Define una región que debe ser ejecutada solo por el *thread master*. Los otros *threads* evitan esta región

```
#pragma omp master newline  
structured_block
```

Constructores de sincronización:

Sincronización es muchas veces necesaria entre *threads* en competencia de ejecución

CRITICAL:

Define una región que debe ser ejecutada por un solo thread a la vez

Si un thread esta ejecutando una region CRITICAL y otro thread alcanza esta región, este último es bloqueado hasta que el primero termine su ejecución

```
#pragma omp critical [ name ] newline  
structured_block
```

Ejemplo de directiva CRITICAL

```
#include <omp.h>
main()
{
    int x;
    x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;
    } /* end of parallel section */
}
```

BARRIER:

Sincroniza todos los threads.

Si un thread la alcanza, va a esperar
a los demás, luego de lo cual,
podrán seguir ejecutando en
paralelo

```
#pragma omp barrier newline
```

TASKWAIT:

Define una espera (wait) en la ejecución de tareas secundarias generadas desde el inicio de la tarea actual del thread

```
#pragma omp taskwait newline
```

ATOMIC:

Especifica que la memoria debe ser actualizada en forma atómica, en vez de dejar a otros threads escribir en ella.

Actúa como una sección mini-CRITICAL

Solo se aplica a la línea que le sigue inmediatamente

```
#pragma omp atomic newline  
statement_expression
```

FLUSH:

Define un punto de sincronización, en el que el algoritmo debe proveer de una cuantificación consistente de la memoria. Las variables activas de cada thread son escritas en la memoria en ese momento.

```
#pragma omp flush (list) newline
```

La lista opcional contiene las variables que van a ser escritas en memoria (flushed), para evitar hacerlo con todas las variables

FLUSH:

```
#pragma omp flush (list) newline
```

FLUSH se usa particularmente con las siguientes directivas:
Barrier,
Parallel - al inicio y final
Critical – al inicio y final
Ordered – al inicio y final
For – al final
Sections – al final
Single – al final

ORDERED:

Especifica que iteraciones en el loop incluído, serán ejecutados en el mismo orden al de un proceso en serie.

Threads deberán esperar antes de ejecutar sus bloques de iteraciones (chunk) si las iteraciones no han sido aún completadas.

Se usa con un DO/for loop

Una iteración no debe ejecutar la misma directiva ORDERED mas de una vez, ni ejecutar mas de una directiva ORDERED

```
#pragma omp for ordered [clauses...]
(loop region)
#pragma omp ordered newline
structured_block
(end of loop region)
```

Algoritmos Paralelos

```
#include <omp.h>
int a, b, i, tid;
float x;
#pragma omp threadprivate(a, x);
main () {
/* Explicitly turn off dynamic threads */
omp_set_dynamic(0);
printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid) {
tid = omp_get_thread_num();
a = tid;
b = tid;
x = 1.1 * tid +1.0;
printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */
printf("Master thread doing serial work here\n");
printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid) {
tid = omp_get_thread_num();
printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */ }
```

THREADPRIVATE:

Se usa para mantener una variable global en cada thread durante la ejecución de regiones en paralelo

#pragma omp threadprivate (list)

1st Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 2: a,b,x= 2 2 3.200000

Thread 3: a,b,x= 3 3 4.300000

Thread 1: a,b,x= 1 1 2.100000

Master thread doing serial work here

2nd Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 3: a,b,x= 3 0 4.300000

Thread 1: a,b,x= 1 0 2.100000

Thread 2: a,b,x= 2 0 3.200000

Cláusulas de atributos de alcance de datos (data-sharing)

Dado que la memoria en OPENMP es compartida, las variables tambien lo son, por defecto.

Variables globales, i.e., variables estaticas (static)

Variables privadas, i.e., indices en loops

Estas directivas, proveen la capacidad de controlar el alcance de datos durante la ejecución de regiones en paralelo (que variables de la region serial del código son trasladadas a la región paralela, y cuales de ellas son visibles para todos los threads o son definidas como privadas)

Cláusulas de atributos de alcance de datos (data-sharing)

PRIVATE:

Un nuevo objeto del mismo tipo es declarado para cada thread
Toda referencia al objeto original es reemplazado con referencias
al nuevo objeto
Variables privadas no estan originalmente inicializadas para cada
thread

SHARED:

Declara variables en una lista para ser compartidas entre los threads
Una variable compartida existe solo en una locación de memoria,
cuya dirección pueden leer todos los threads.

Para acceder a las variables se recomienda usar secciones
críticas (CRITICAL)

shared (list)

DEFAULT:

Especifica el alcance por defecto de todas las variables
en toda region en paralelo

Solo puede ser usada una vez en una directiva

PARALLEL

default (shared | none)

FIRSTPRIVATE:

Combina PRIVATE con una inicialización automática de las variables en su lista

El valor de inicialización es el de los objetos originales

Antes de entrar al constructor de la región en paralelo

firstprivate (list)

LASTPRIVATE:

Combina PRIVATE con una copia de la variable original, de la última iteración o sección

EL valor de la variable original al final de la ejecución es el de la última iteración o sección

lastprivate (list)

COPYIN:

Provee una copia del valor de variables THREADPRIVATE para todos los threads

copyin (*list*)

COPYPRIVATE:

Asigna el valor de variables de un thread a todas las instancias de las variables privadas en los demás threads.
Se utiliza con SINGLE.

copyprivate (*list*)

REDUCTION:

Realiza una reducción de las variables en la lista. Para ello se copian las variables en cada thread. Finalmente, la reducción se aplica a todas las copias privadas de la variable compartida, y el valor final se escribe en la variable global compartida

reduction (operator: list)

Ejemplo de REDUCTION:

Iteraciones en el loop en paralelo se distribuyen en bloques del mismo tamaño a cada thread (SCHEDULE STATIC)
Al final del loop, todos los threads suman su valor a una variable global *result*, del master thread

Variables en el loop deben ser escalares (no arrays o estructuras), y deben ser declaradas SHARED

```
#include <omp.h>
main () {
    int i, n, chunk;
    float a[100], b[100], result;
    /* Some initializations */
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++)
    {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
#pragma omp parallel for \
    default(shared) private(i) \
    schedule(static,chunk) \
    reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
}
```

Cláusulas en directivas OPENMP

Directiva que no aceptan cláusulas:

MASTER

CRITICAL

BARRIER

ATOMIC

FLUSH

ORDERED

THREADPRIVATE

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Variables de entorno:

OMP_SCHEDULE

Se aplica solo a directivas DO, parallel for, y determinan el ordenamiento de iteraciones en el loop. I.e. `setenv OMP_SCHEDULE "guided, 4 "`

OMP_NUM_THREADS

Número máximo de threads. I.e. `setenv OMP_NUM_THREADS 8`

OMP_DYNAMIC

Activa/desactiva ajuste dinámico del número de threads accesibles para ejecución de regiones en paralelo (TRUE/FALSE). I.e. `setenv OMP_DYNAMIC_TRUE`

Variables de entorno:

OMP_PROC_BIND

Activa/desactiva threads ligados a procesos (TRUE/FALSE). I.e.

`setenv OMP_PROC_BIND TRUE`

OMP_NESTED

Activa/desactiva paralelismo anidado (TRUE/FALSE). I.e. `setenv`

`OMP_NESTED TRUE`

OMP_THREAD_LIMIT

Fija el número de threads a usarse en todo el programa. I.e.

`setenv OMP_THREAD_LIMIT 8`

Variables de entorno:

OMP_STACKSIZE

Controla el tamaño del stack para (non-Master) threads creados. Se da en kilobttes. I.e.

`setenv OMP_STACKSIZE 300K`

`setenv OMP_STACKSIZE 10M`

`setenv OMP_STACKSIZE 1G`

`setenv OMP_STACKSIZE 20000`

Variables de entorno:

OPENMP standard no especifica el tamaño del thread por defecto

Compiler	Approx. Stack Limit	Approx. Array Size (doubles)
Linux icc, ifort	4 MB	700 x 700
Linux pgcc, pgf90	8 MB	1000 x 1000
Linux gcc, gfortran	2 MB	500 x 500

En ciertos casos, un programa tiene mejor performance si sus threads estan ligados a procesadores/núcleos

```
setenv OMP_PROC_BIND TRUE  
setenv OMP_PROC_BIND FALSE
```

Ejercicio 1: variables privadas

Escribir un programa que modifique dos variables privadas `i,j` dentro de una directiva en paralelo, pero no cambie los valores de las variables originales.

Para ello, realice `assert(*ptr_i == 1, *ptr_j ==2)` dentro de la zona en paralelo, donde `ptr_i=&i`, y `ptr_j=&j`

Asimismo, realice un segundo `assert (i ==1 && j==2)` fuera de la region en paralelo.

Imprima los valores de `i` y `j` dentro y fuera de la región en paralelo, y observe como cambian los valores dentro pero no fuera.

Utilize 4 procesos para OPENMP

Ejercicio 2: ordenamiento

Utilizar el constructor de sincronización `ORDERED` para ordenar secuencialmente los índices de un array de 100 elementos, escritos de 5 en 5 y empezando de 0. Para ello declare `...for ORDERED ...` antes del for loop que imprima los valores del array. No se olvide de declarar `... ORDERED` justo antes de la impresion.

Algoritmos Paralelos

Ejercicio 3: Variables compartidas/privadas y reducción

Reescribir el siguiente programa con solo las directivas OPENMP siguientes:

#pragma omp parallel , antes de inicializar `a=0`

#pragma omp for reduction (+,a) , antes del `for`

Observar el resultado de la suma luego de varios intentos. Se mantiene el resultado?

Reescribir el programa con las directivas de sincronización OPENMP adecuadas, tal que:

`a` sea una variable compartida `shared(a)`, e

`i` sea una privada `private(i)`

La inicialización de `a` se haga solo por el *master thread*

La impresión de la suma la haga solo un thread directiva `single`

Compruebe si el resultado es ahora correcto

En caso contrario, donde sería adecuado incluir una barrera? #pragma omp barrier

```
#include <stdio.h>
int main (void) {
    int a, i;
    a = 0;
    for (i = 0; i < 10; i++) {
        a += i;
    }
    printf ("La suma es %d\n", a);
}
return 0;
}
```

Escribir un programa similar para hallar el mínimo y el máximo de un array de 10 elementos generados aleatoriamente entre 1 y 50