MEMORIA COMPARTIDA: OPENMP

RUMBO AL PARALELISMO

MODELOS PARA MEMORIA COMPARTIDA

Programa = conjunto de procesos/hilos concurrentes con acceso a espacios de direcciones compartidas y privadas.

- Programación muy similar al paradigma secuencial: no hay intercambios explícitos de información.
- Comunicación y sincronismo vía variables compartidas.
- El acceso concurrente a memoria es la clave del modelo.
- **Paralelismo de Datos vs Paralelismo de Tareas**.

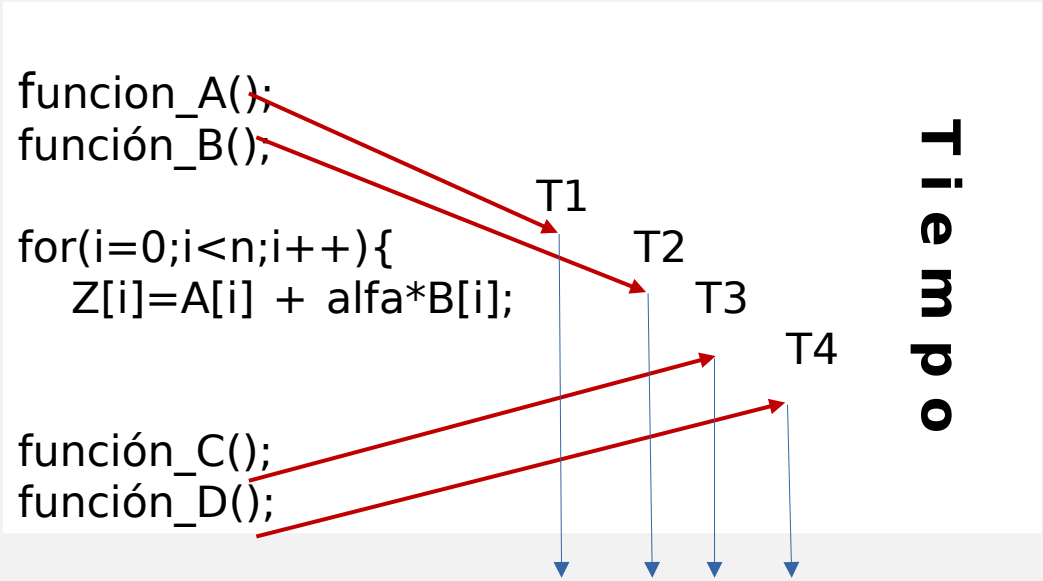
MODELOS PARA MEMORIA COMPARTIDA

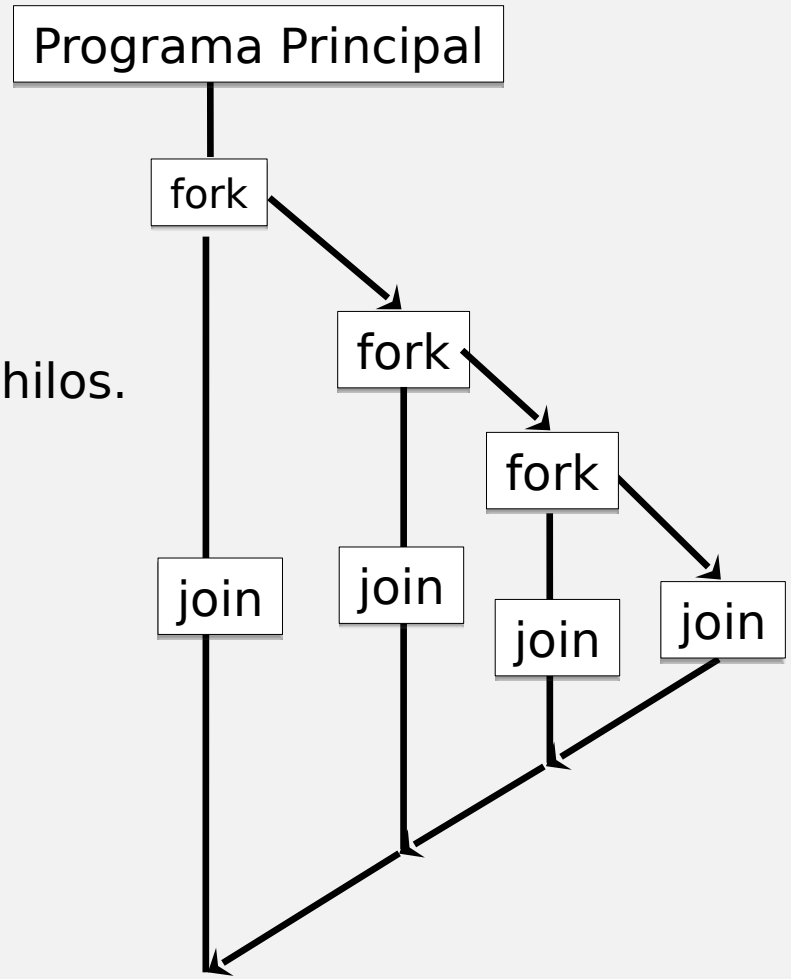
fork/join (Conway, 1963)

- De los primeros, permitiendo manejar tanto procesos
- como hilos.

Hilos

- Un proceso puede constar de múltiples hilos.
- Cada hilo tiene datos privados y comparte recursos con otros hilos.
- En ciertos momentos hay que realizar sincronizaciones.





MODELOS PARA MEMORIA COMPARTIDA

Hilos: Retos

- El problema de la *Race Condition*. No determinista y, por tanto, el resultado final puede ser incorrecto.
- Exclusión Mutua
  - Operaciones atómicas
    - Realizar de forma atómica (sin interrupción) las operaciones problemáticas y/o usar instrucciones especiales del procesador (p. ej. *test-and-set* o *compare-and-exchange*).
- Sección crítica
  - No más de un hilo ejecutándola dentro de ellas. Usar mecanismos de sincronización. Riesgo de interbloqueo (deadlocks).
- Sincronización
  - Normalmente mediante el uso de barreras y/o ejecución ordenada.

MODELOS PARA MEMORIA COMPARTIDA

Hilos POSIX (*pthreads*)

- Estándar en sistemas Unix. Basado en librerías. Problemas de portabilidad.
- Paralelismo explícito. Más orientado a paralelismo de tareas.

```
#include <pthread.h>
void ver(void *mensaje) {
   printf("%s\n", (char *)mensaje);        // Race Condition
}
int main() {
   pthread_t h1, h2;   char *m1="Hola", *m2="Mundo";   int error, *status;
   pthread_create(&h1,NULL,(void*)&ver,(void*)m1);
   pthread_create(&h2,NULL,(void*)&ver,(void*)m2);
   error = pthread_join(h1,(void *)&status);
   if (error && ((int)&status != h1))     ...
   error = pthread_join(h2,(void *)&status);
   if (error && ((int)&status != h2))     ...
}
```

MODELOS PARA MEMORIA COMPARTIDA

Hilos Java

ORIENTADO A OBJETOS

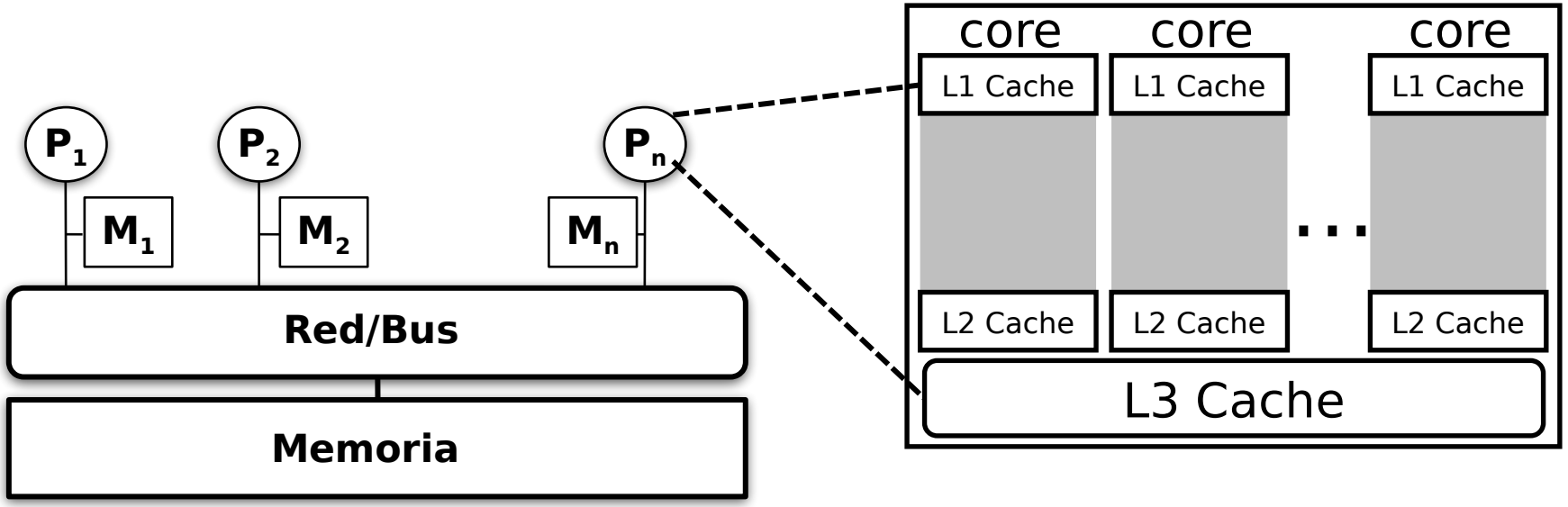
```
public class Hola extends Thread {
   public void run() {
      System.out.println(“Hola desde un hilo");
   }
   public static void main(String args[]) {  (new Hola()).start();  }
}
```

MÉTODOS SINCRONIZADOS

```
public class SynchronizedCounter {
   private int cont = 0;
   public synchronized void increment() {
      cont++;
   }
   public synchronized int value() {
      return cont;
   }
}
```

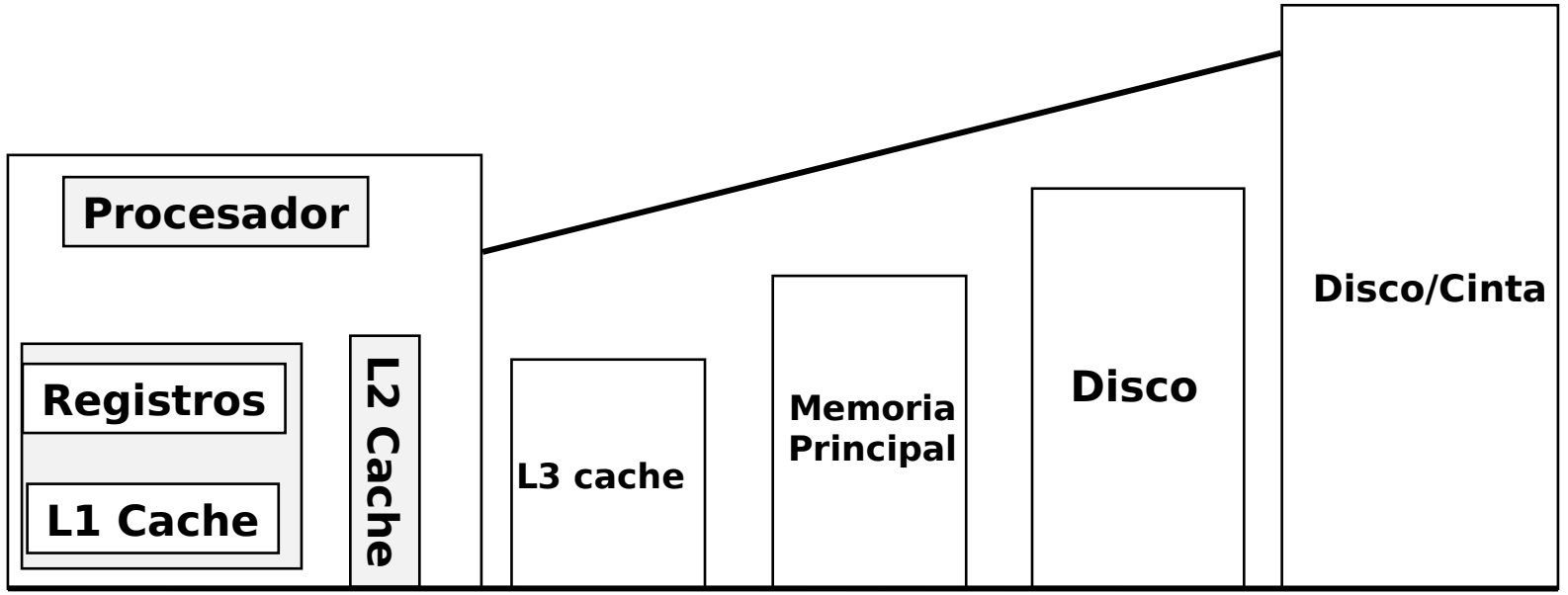
DETALLES DE ARQUITECTURA RELEVANTES: CADA *NODO*

- Sistemas con un único espacio "*lógico*" de direcciones de memoria accesible por todas las unidades de proceso. El acceso a las *distintas* memorias puede ser a diferente velocidad.
- No hay un modelo universal, como sucede con las secuenciales (abstracción *von Neumann*), pero se ha avanzado considerablemente.
- Multiprocesadores, multinúcleo, coprocesadores.



DETALLES DE ARQUITECTURA RELEVANTES: LA MEMORIA Y SU ACCESO

1. Jerarquías de Memoria



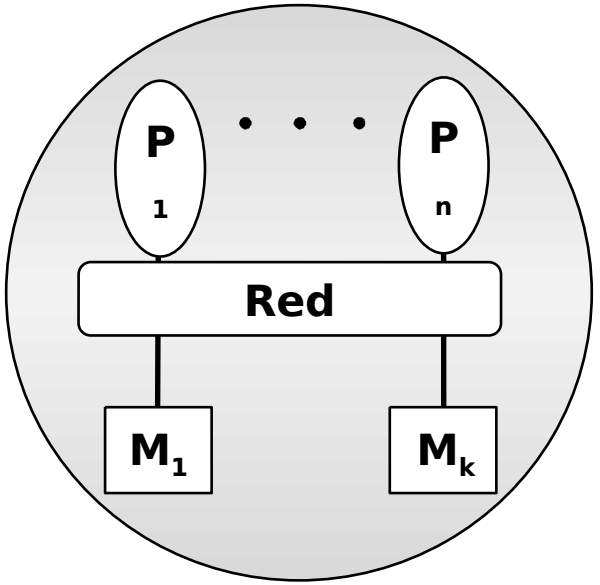
Speed	~ 1 ns	~ 10 ns	~ 100 ns	~ 10 ms	~ 10 sec
Size	Kbytes	Mbytes	Gbytes	Tbytes	Tbytes

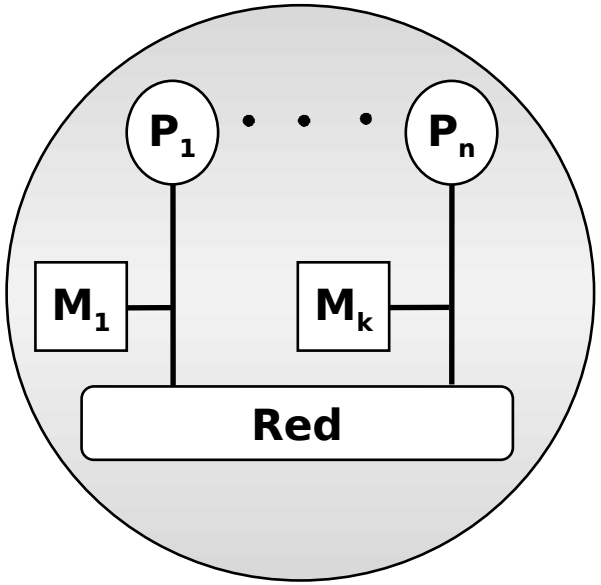
Los valores pueden cambiar respecto a los actuales pero las diferencias entre magnitudes se mantienen

DETALLES DE ARQUITECTURA RELEVANTES: LA MEMORIA Y SU ACCESO

2. Arquitectura de Memoria

UMA (*Uniform Memory Access*). El tiempo de acceso es el mismo para todos los procesadores / direcciones de memoria. También llamados SMPs (*symmetric multiprocessors*)





NUMA (*Non-Uniform Memory Access*). El tiempo de acceso a los datos depende de dónde estén almacenados. Escala mejor (precio y facilidad). **ccNUMA**, **COMA**, etc.

DETALLES DE ARQUITECTURA RELEVANTES: LA MEMORIA Y SU ACCESO

3. Coherencia y Consistencia de la información (CC)
   - El espacio único de direcciones accesible a varios procesos/hilos/etc. concurrentes es una **fuente de conflictos**.
   - La masiva presencia de las jerarquías de memoria añaden complejidad.
   - Quien programa, o el hardware/software, deben garantizar la CC.
   - Los hábitos de programación agrandan/disminuyen los conflictos.

Inicio:	**flag=0** **data=0**

P_1	P_2
data = 1	**Si flag=1**
flag = 1	**…= data**

P_2 lee *flag*	*data* para P_2
0	**0**

DETALLES DE ARQUITECTURA RELEVANTES: LA MEMORIA Y SU ACCESO

3. Coherencia y Consistencia de la información (CC)
   - El espacio único de direcciones accesible a varios procesos/hilos/etc. concurrentes es una **fuente de conflictos**.
   - La masiva presencia de las jerarquías de memoria añaden complejidad.
   - Quien programa, o el hardware/software, deben garantizar la CC.
   - Los hábitos de programación agrandan/disminuyen los conflictos.

Inicio:	**flag=0** **data=0**

P_1	P_2
data = 1	**Si flag=1**
flag = 1	**…= data**

P_2 lee *flag*	*data* para P_2
0	**0**
0	**1**

DETALLES DE ARQUITECTURA RELEVANTES: LA MEMORIA Y SU ACCESO

3. Coherencia y Consistencia de la información (CC)
   - El espacio único de direcciones accesible a varios procesos/hilos/etc. concurrentes es una **fuente de conflictos**.
   - La masiva presencia de las jerarquías de memoria añaden complejidad.
   - Quien programa, o el hardware/software, deben garantizar la CC.
   - Los hábitos de programación agrandan/disminuyen los conflictos.

Inicio:	**flag=0** **data=0**

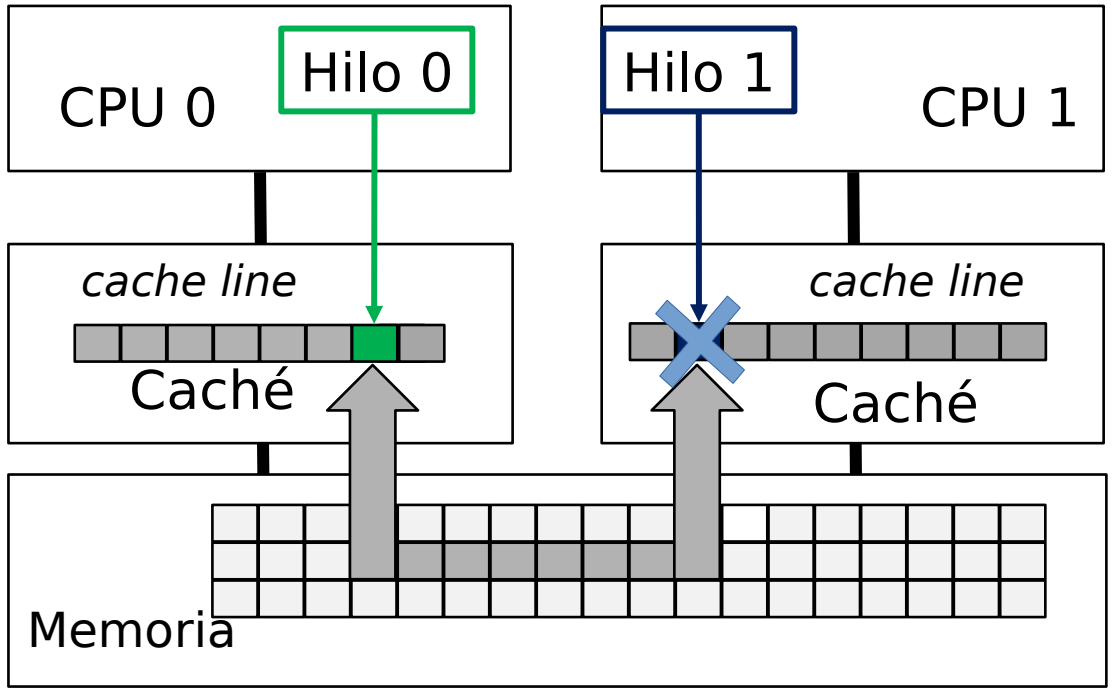
P_1	P_2
data = 1	**Si flag=1**
flag = 1	**…= data**

P_2 lee *flag*	*data* para P_2
0	**0**
0	**1**
1	**1**

DETALLES DE ARQUITECTURA RELEVANTES: LA MEMORIA Y SU ACCESO

4. Contención. Se accede a datos comunes o datos que están en el mismo bloque de memoria. Si son lecturas puede no haber problema (distinto al problema de coherencia).
5. *False Sharing*. En sistemas con CC si un hilo modifica un dato (que solo él necesita) que está en el mismo bloque de caché con datos necesarios para otros hilos. Para mantener la coherencia se invalida todo el bloque.

```
En region paralela:
  k = obtener_mi_id();
  S[k] = 0.0;
  for (i=0;i<n;i++)
    S[k] += b[i][k];
```



DETALLES DE ARQUITECTURA RELEVANTES: LA MEMORIA Y SU ACCESO

Positivo

- El espacio único de direcciones, unido a las herramientas actuales, facilita la programación.
- Cambiar de un multiprocesador a un multinúcleo, o viceversa, no implica cambios en el código.

Negativo

- Todos los conceptos comentados no afectan a la corrección del algoritmo (salvo la CC) pero sí al rendimiento.
- Mayor dificultad que en secuencial para controlar la localidad de los datos.
- Cambiar de un multiprocesador a un multinúcleo, o viceversa, sí puede afectar al rendimiento.
- Es fácil generar código correcto pero difícil que sea eficiente.

PRESENTACIÓN DE OPENMP

OpenMP
Enabling HPC since 1997

The OpenMP API specification for parallel programming

Home | Specifications | Blog | Community | Resources | News & Events | About

OpenMP
Enabling HPC since 1997

The OpenMP API specification for parallel programming

Home | Specifications | Blog | Community | Resources | News & Events | About

OpenMP Books

Home > Resources > OpenMP Books

- OpenMP Common Core: Making OpenMP Simple Again – by Tim Mattson, Helen He, Alice Koniges (2019)
- Using OpenMP – The Next Step – by Ruud van der Pas, Eric Stotzer and Christian Terboven (2017)
- Using OpenMP – Portable Shared Memory Parallel Programming – by Chapman, Jost, and Van Der Pas (2007)
- Parallel Programming in OpenMP – by Rohit Chandra et al.
- Parallel Programming in C with MPI and OpenMP – by Michael J. Quinn.
- Parallel Programming Patterns: Working with Concurrency in OpenMP, MPI, Java, and OpenCL – by Timothy G. Mattson, Berna Massingill and Beverly Sanders
- An Introduction to Parallel Programming with OpenMP, PThreads and MPI – by Robert Cook
- The International Journal of Parallel Programming – Issues and articles devoted to OpenMP.

PRESENTACIÓN DE OPENMP

PRESENTACIÓN DE OPENMP

The OpenMP Architecture Review Board

NEC
Shin-ichi Okano

NVIDIA
Jeff Larkin

Oak Ridge National Laboratory
Oscar Hernandez

RWTH Aachen University
Dieter an Mey

Sandia National Laboratory
Stephen Olivier

Stony Brook University
Dr. Barbara Chapman

SUSE
Michael Matz

Texas Advanced Computing Center
Kent Milfeld

The University of Manchester
Antoniu Pop

University of Basel
Florina Ciorba

University of Bristol
Simon McIntosh-Smith

University of Delaware
Sunita Chandrasekaran

University of Tennessee
Piotr Luszczek

AMD
Greg Rodgers

Argonne National Laboratory
Kalyan Kumaran

ARM
Graham Hunter

ASC/Lawrence Livermore National Laboratory
Bronis R. de Supinski

Barcelona Supercomputing Center
Xavier Martorell

Brookhaven National Laboratory
Vivek Kale

cOMPunity
Yonghong Yan

Edinburgh Parallel Computing Centre
Mark Bull

Fujitsu
Naoki Sueyasu

Hewlett-Packard Enterprise
Deepak Eachempati

a Hewlett Packard Enterprise company

IBM
Kelvin Li

INRIA
Olivier Aumage

Intel
Xinmin Tian

Lawrence Berkeley National Laboratory
Helen He

Leibniz Supercomputing Centre
Volker Weinberg

Los Alamos National Laboratory
Jamal Mohd-Yusof

Maui High Performance Computing Center
Alice Koniges

Mentor Graphics
Catherine Moore

Micron
Randy Meyer

NASA
Henry Jin

PRESENTACIÓN DE OPENMP

Este documento se centra en las especificaciones totalmente soportadas por los compiladores utilizados. Especificaciones futuras pueden tener comportamientos diferentes.

PRESENTACIÓN DE OPENMP

- Proyecto iniciado por Silicon Graphics, Inc. (SGI).
- Es la referencia/estándar para memoria compartida.
- Diseñado para simplificar el paradigma Posix Thread. Pensado inicialmente para UMA (extendido a NUMA).

- Es una especificación formal. La última es la 5.2 (noviembre 2021).

- Paralelismo a nivel de bucle, tareas, SIMD, etc.
- Soportado por los compiladores mediante extensiones del lenguaje.
- Admite paralelismo anidado (su implementación es opcional).
- Admite hilos dinámicos (nuevamente la implementación es opcional).

- Basado en el modelo *fork-join*.

MODELO DE EJECUCIÓN DE OPENMP

- OpenMP no destruye los hilos inactivos, los deja en estado de *espera* para ser reutilizados en la siguiente región paralela.
- Los hilos creados por una directiva forman un equipo (***team***).

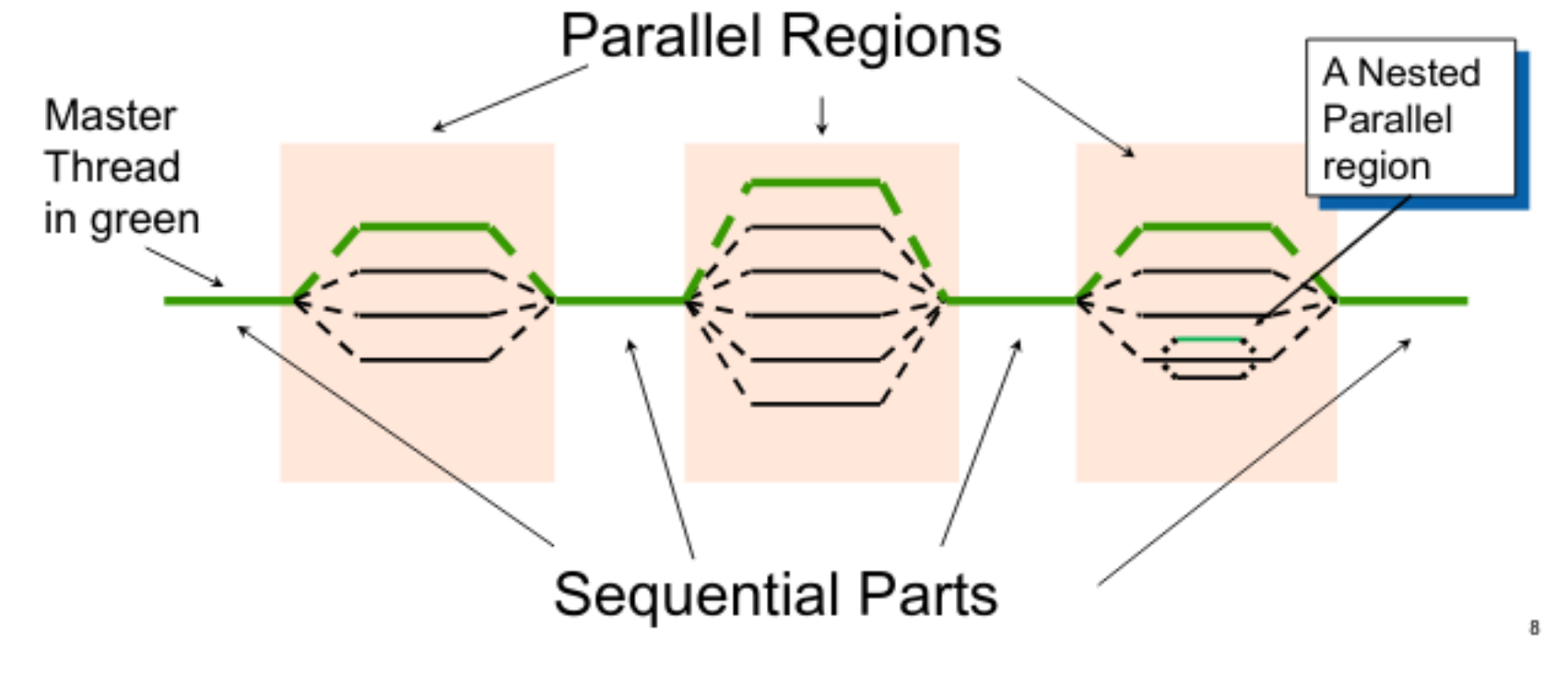
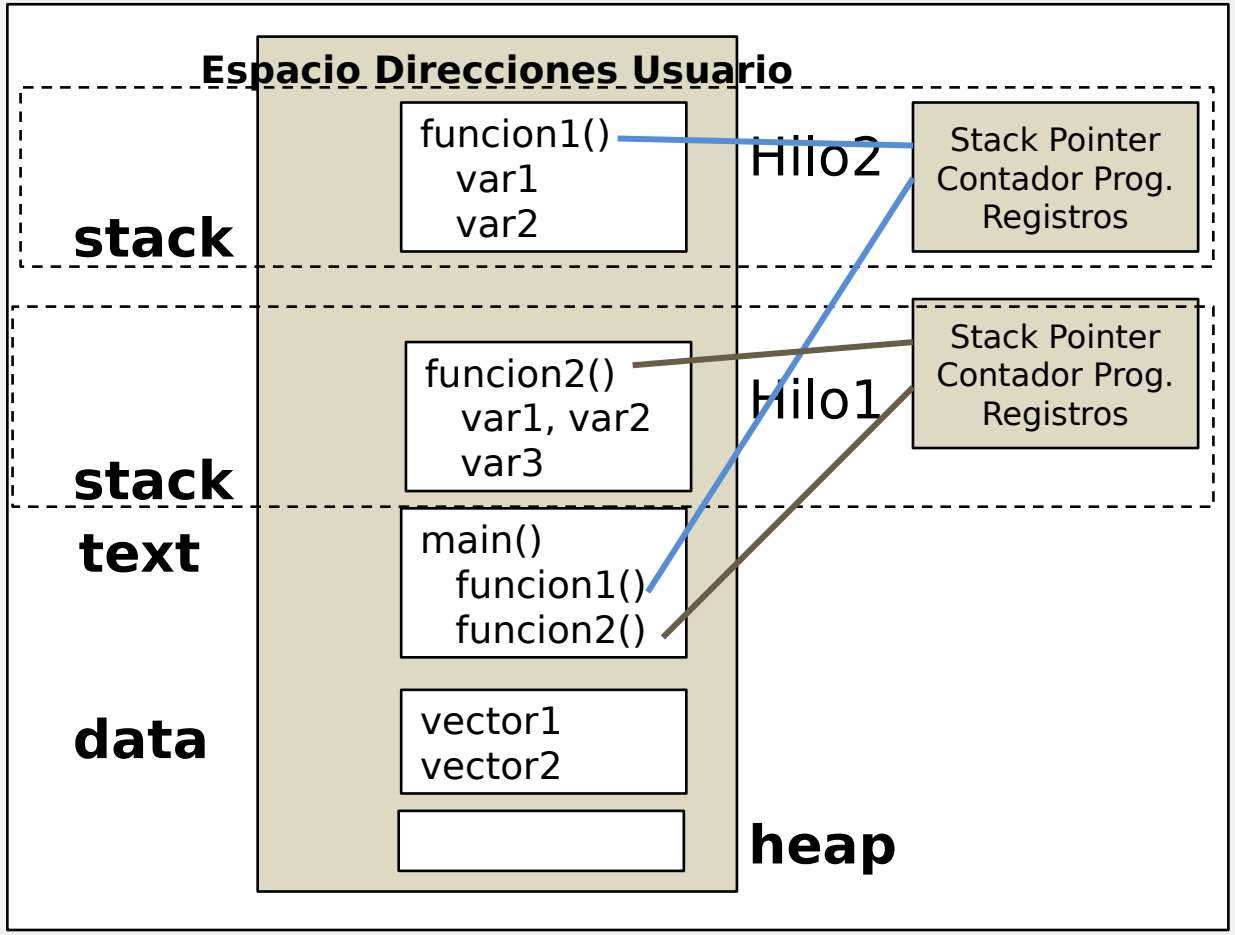


Imagen extraída de http://avcourt.org/documents/task-level-omp.pdf

MODELO DE MEMORIA DE OPENMP

- Memoria compartida con ***relaxed-consistency***: la "visión temporal" de la información que tienen los hilos puede no ser consistente con la información almacenada en memoria.
- Cada hilo tiene un contexto completo de ejecución.



MODELO DE PROGRAMACIÓN DE OPENMP

- La programación se basa en el uso de *directivas* (#pragma), en el uso de la API (*funciones*) y en el uso de *variables de entorno*.

```
void daxpy(int n, double a, double *x, double
*y, double *z)
{
   int j;
   #pragma omp parallel for
   for (j=0; j<n; j++)
        z[j] = a*x[j] + y[j];
}
```

- Facilita la migración. El compilador puede no soportar las directivas: se ejecuta igualmente en secuencial.
- Permite la paralelización incremental.
- Tanto para paralelismo de grano fino como grueso.
- Permite optimizaciones provenientes del compilador.

OPENMP AL COMPLETO

OpenMP

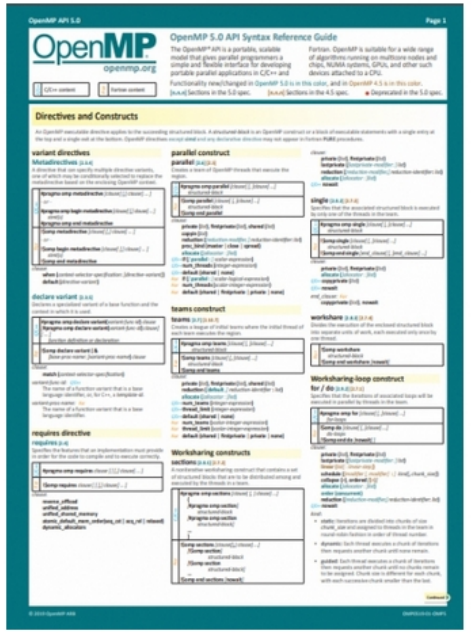
The OpenMP API specification for parallel programming

Home | Specifications | Community | Resources | News & Events | About

Reference Guides

Home > Resources > Reference Guides

These 8- and 12-page documents provide a quick reference to OpenMP with section numbers that refer you to where you can find greater detail in the full specification.



OpenMP 5.1

- OpenMP 5.1 Reference Guide (Nov 2020) PDF (optimized for web view)

OpenMP 5.0

- OpenMP 5.0 Reference Guide (May 2019) PDF (optimized for web view)
- OpenMP 5.0 Reference Guide (May 2019) PDF (optimized for local printing)
- OpenMP 5.0 Reference Guide (May 2019) Purchase Lulu.com print-on-demand hard copy
- OpenMP 5.0 Tasking Reference Guide (Sep 2019) PDF

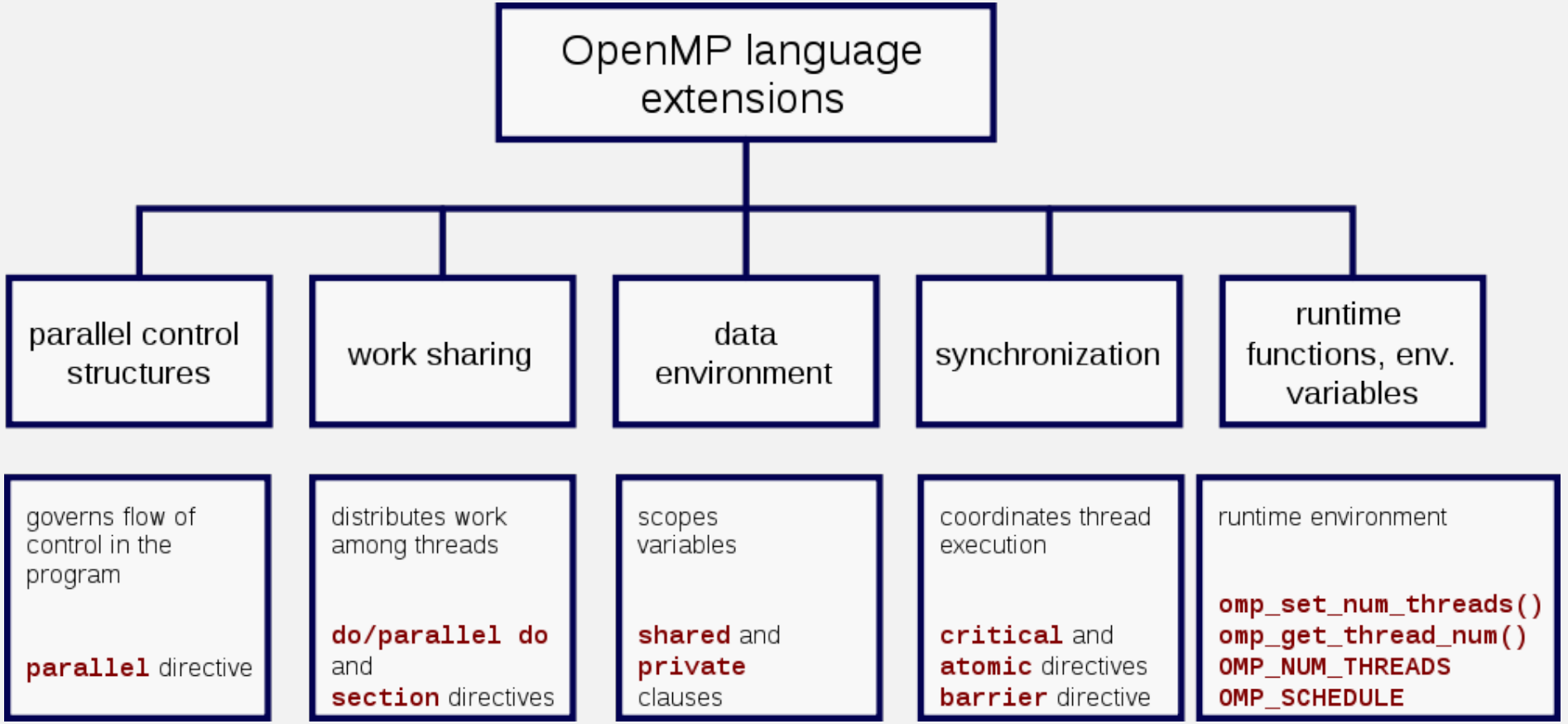
OpenMP 4.5

- OpenMP 4.5 Reference Guide – C/C++ (Nov 2015) PDF
- OpenMP 4.5 Reference Guide – Fortran (Nov 2015) PDF

OpenMP 4.0

- OpenMP 4.0 Reference Guide – C/C++ (October 2013 PDF)
- OpenMP 4.0 Reference Guide – Fortran (October 2013 PDF)

OPENMP AL COMPLETO



Importante

- El acceso a los *structured blocks* no puede ser consecuencia de un salto.
- El punto de salida no puede ser un salto fuera de los *structured blocks*.

OPENMP: VARIABLES DE ENTORNO

- **OMP_NUM_THREADS** Fija el número de hilos para regiones paralelas.

  export OMP_NUM_THREADS=8

- **OMP_NESTED**: Activa/desactiva el paralelismo anidado. Por defecto no.

  export OMP_NESTED=true

- **OMP_SCHEDULE** Fija el planificador para el paralelismo de bucles. Sólo afecta a los bucles con planificación runtime.

  export OMP_SCHEDULE=“guided,4”

- **OMP_DYNAMIC** Activa/desactiva, en función de la carga del sistema, el ajuste dinámico del número de threads para las regiones paralelas.

  export OMP_DYNAMIC=false

  false (estático) El número de hilos lo fija el programador. Si no hay la cantidad solicitada la ejecución falla.

  true (dinámico) El usuario establece el número máximo, el sistema los usados.

- Etc.

RECURSOS DE PROGRAMACIÓN EN OPENMP

DIRECTIVAS

```
#pragma omp <directiva> [clausula [...]]
```

- Pueden tener 0, 1 o varias cláusulas.
- Las cláusulas permiten modificar el comportamiento de la directiva, como por ejemplo aspectos tales como:
  - La paralelización condicional: cláusula *if(expresión)*, no soportada por todas las directivas.
  - El grado de concurrencia, como *num_threads, nowait, ordered,* etc. No todas soportadas por todas las cláusulas.
  - Los mecanismos para la gestión de datos, como *private, firstprivate, lastprivate, copyin, shared, default o reduction*. Nuevamente, no todas usables en todas las cláusulas.

RECURSOS DE PROGRAMACIÓN EN OPENMP

DIRECTIVAS

```
#pragma omp <directiva> [clausula [...]]
```

COMPILACIÓN CONDICIONADA

```
#ifdef _OPENMP
  ...
#endif
```

RECURSOS DE PROGRAMACIÓN EN OPENMP

DIRECTIVAS

```
#pragma omp <directiva> [clausula [...]]
```

COMPILACIÓN CONDICIONADA

```
#ifdef _OPENMP
  ...
#endif
```

FUNCIONES

```
#include <omp.h>
...
iam = omp_get_thread_num();
```

LAS FUNCIONES (API)

- int **omp_get_dynamic**(void)

  Devuelve **0** si no está activado el ajuste dinámico de hilos.
- void **omp_set_dynamic**(int valor)

  Si ***valor*** es **0** el *runtime* no ajusta dinámicamente el número de hilos.
- int **omp_get_nested**(void)

  Devuelve **0** si no está activado el paralelismo anidado.
- void **omp_set_nested**(int valor)

  Si valor es **0** se desactiva el paralelismo anidado.
- int **omp_in_parallel**(void)

  Devuelve **0** si es llamada fuera de una región paralela.
- int **omp_get_num_procs**(void)

  Devuelve el número de procesadores físicos disponibles.
- int **omp_get_max_threads**(void)

  Devuelve el número máximo de hilos que se pueden usar.

LAS FUNCIONES (API)

- void **omp_set_num_threads**(int num_threads)

  Establece el número de hilos para las regiones paralelas.
- int **omp_get_num_threads**(void)

  Devuelve el número de hilos de la región parelela actual.
- int **omp_get_thread_num**(void)

  Devuelve el identificador (entre 0 y n-1) del hilo.
- Etc.

- El número de hilos se puede modificar con variables de entorno, funciones y la cláusula *num_threads*. Las prioridades, de mayor a menor, y el ámbito, de menor a mayor, son:

  1. cláusula *num_threads*,
  2. funciones y
  3. variables de entorno.

RECURSOS DE PROGRAMACIÓN EN OPENMP

DIRECTIVAS

```
#pragma omp <directiva> [clausula [...]]
```

COMPILACIÓN CONDICIONADA

```
#ifdef _OPENMP
  ...
#endif
```

FUNCIONES

```
#include <omp.h>
...
iam = omp_get_thread_num();
```

COMPILACIÓN

```
Gcc:    gcc -fopenmp
Intel:  icc  -[f,q]openmp                o        -[f,q]openmp-simd
```

EJEMPLO: VARIABLES DE ENTORNO

```
#include <omp.h>
int main () {
  int nth, tid;

  #pragma omp parallel private(nth, tid)
  {
     #ifdef _OPENMP
        tid  = omp_get_thread_num();
        nth = omp_get_num_threads();
     #endif
     printf("Soy %d de %d hilos\n", tid, nth);
  }
}
```

```
gcc -fopenmp
export OMP_NUM_THREADS = 4
./32
```

Resultado de la ejecución

Soy 0 de 4 hilos
Soy 1 de 4 hilos
Soy 3 de 4 hilos
Soy 2 de 4 hilos

EJEMPLO: VARIABLES DE ENTORNO + FUNCIONES

```
#include <omp.h>
int main () {
   int nth, tid;
   omp_set_num_threads(2);
   #pragma omp parallel private(nth, tid)
   {
      #ifdef _OPENMP
         tid  = omp_get_thread_num();
         nth = omp_get_num_threads();
      #endif
      printf("Soy %d de %d hilos\n", tid, nth);
   }
}
```

```
export OMP_NUM_THREADS = 4
./33
```

Resultado de la ejecución:

Soy 0 de 2 hilos
Soy 1 de 2 hilos

EJEMPLO: VARIABLES DE ENTORNO + FUNCIONES + CLÁUSULAS

```
#include <omp.h>
int main () {
  int nth, tid;
  omp_set_num_threads(2);
      #pragma omp parallel private(nth, tid)
num_threads(3)
  {
     #ifdef _OPENMP
        tid  = omp_get_thread_num();
        nth = omp_get_num_threads();
     #endif
     printf("Soy %d de %d hilos\n", tid, nth);
  }
}
```

export **OMP_NUM_THREADS** = 4
./34

Resultado de la ejecución:

Soy 0 de 3 hilos
Soy 2 de 3 hilos
Soy 1 de 3 hilos

CONSTRUCTOR PARALLEL

```
#pragma omp parallel [cláusulas]
    structured block
```

- La tarea del hilo *maestro* de la región secuencial que ejecuta la directiva crea un grupo (*team*) de hilos.
- Durante el alcance de la directiva (región paralela) el número de hilos en el *team* no cambia: estático.
- El hilo maestro es el *0*. El resto de hilos se numeran de *1* a *n-1*.
- Creado el *team*, el hilo maestro crea implícitamente tantas tareas como hilos tenga el equipo.
- A cada tarea se le asigna una fracción del *structured block*.
- *structured block* es un bloque estructurado estándar en el lenguaje de programación (en C con llaves si …).
- Cada tarea es asignada a un hilo diferente del *team* en modo **tied**.
- Suspende la ejecución de la tarea creadora y cada hilo inicia la ejecución de su tarea.
- No hay sincronismo implícito de los hilos dentro de la región.
- Hay sincronismo implícito de salida. El maestro continúa y el resto se destruyen/inactivos.
- Los hilos pueden ejecutar diferentes sentencias mediante comprobaciones lógicas.
- Admite anidamiento (otro constructor *parallel*) si el paralelismo anidado está activado (*omp_set_nested(1)* o *OMP_NESTED=true*).

CONSTRUCTOR PARALLEL: CLÁUSULAS

```
#pragma omp parallel [cláusulas]
    structured block
```

- *if(expresión escalar)* Se paraleliza la región ⟺ se cumple la condición.
- *default(share|none)* Define el alcance por defecto de las variables. Si se usa *default(none)* todo es privado.
- *private(lista)* Lista de variables que serán privadas. El valor de ellas al inicio/fin de región es indefinido.
- *shared(lista)* Lista de variables compartidas por todos los hilos.
- *firstprivate(lista)* Lista de privadas que se inicializan con el último valor almacenado antes de la directiva.
- *num_threads(expresión)* Fija el número de hilos en la región paralela.
- *copyin(lista)* Copia el valor de las variables indicadas en la lista del hilo maestro al resto. Dichas variables solo pueden aparecer en un *copyin* y tienen que ser privadas (o *threadprivate*, se verá).
- *reduction(operador: lista)* Realiza una operación de reducción de forma segura sobre las variables escalares especificadas utilizando el operador indicado. Las variables *se declaran*, implícitamente, privadas.

  Los operadores soportados son: +, ∗, -, &, |, ˆ, &&, ||, max, min. Se inicializa al elemento neutro del operador, esto es, a 0, 1, 0, ~0, 0, 0, 1, 0, menor valor y mayor valor, respectivamente.

CONSTRUCTOR PARALLEL: CLÁUSULAS

```
#pragma omp parallel [cláusulas]
    structured block
```

Añadida en la especificación 4.0

- *proc_bind(master | close | spread)* Controla la política de afinidad del constructor parallel. **Master** → todo el team al mismo *place* que el hilo maestro. **Close** → los hilos del equipo se asignan a *places* cercanos al del hilo maestro. **Spread** → crear una distribución esparcida de los hilos en los *places*.

  Place es una abstracción (thread, core, socket) o una lista de números no negativos (igual que para *affinity*). Ver *OMP_PLACES* para mayor información.

  Añadida en la especificación 4.5

- *if([parallel:] expresión escalar)* Permite aplicar selectivamente esta cláusula a uno o varios constructores.

CONSTRUCTOR PARALLEL

```
#include <omp.h>
int main ()
{
   int c = 98;
   #pragma omp parallel private(c)
   {
      printf("c: %d \n", c);
   }
   return 0;
}
```

¿ Resultado ejecución 3 hilos ?

c: 0 o cualquier valor

c: 0 o cualquier valor

c: 0 o cualquier valor

```
#include <omp.h>
int main ()
{
   int c = 98;
   #pragma omp parallel firstprivate(c)
   {
      printf("c: %d \n", c);
   }
   return 0;
}
```

¿ Resultado ejecución 3 hilos ?

c: 98

c: 98

c: 98

CONSTRUCTOR PARALLEL

```
#include <omp.h>
int main () {
  int oyo, iyo, onp, inp;
  omp_set_nested(valor);
  #pragma omp parallel num_threads(2) private(oyo, onp)
  {
    oyo=omp_get_thread_num();  onp=omp_get_num_threads();
    printf("Out: (%d, %d)\n", oyo, onp);
     #pragma omp parallel num_threads(2) private(iyo, inp)
     {
       iyo=omp_get_thread_num();  inp=omp_get_num_threads();
       printf("Inner: (%d, %d / %d, %d)\n", oyo, onp, iyo, inp);
     }
  }
  return 0;
}
```

valor = 0
¿Resultado?
Out: (0, 2)
Out: (1, 2)
Inner: (0, 2 / 0, 1)
Inner: (1, 2 / 0, 1)

valor = 1
¿Resultado?

Out: (0, 2)
Out: (1, 2)
Inner: (0, 2 / 0, 2)
Inner: (0, 2 / 1, 2)
Inner: (1, 2 / 0, 2)
Inner: (1, 2 / 1, 2)

CONSTRUCTOR PARALLEL

```
#include <omp.h>
int main () {
  int yo, iyo, iiyo, np, inp, iinp;
  omp_set_nested(0);
  printf("yo  np  iyo  inp  iiyo  iinp\n");
  #pragma omp parallel num_threads(2)
  {
     yo = omp_get_thread_num(); np = omp_get_num_threads();
     #pragma omp parallel private(iyo, inp) num_threads(3)
     {
        iyo = omp_get_thread_num();  inp = omp_get_num_threads();
        #pragma omp parallel private(iiyo, iinp) num_threads(3)
        {
           iiyo = omp_get_thread_num();  iinp = omp_get_num_threads();
           printf("%2d %3d %3d  %3d %4d  %4d\n", yo, np, iyo, inp, iiyo, iinp);
        }
     }
  }
```

¿Resultado?

CONSTRUCTOR PARALLEL

```
yo  np  iyo  inp  iiyo  iinp
--------------------------------
 0   2    0    1     0     1
 1   2    0    1     0     1
```

```
yo  np  iyo  inp  iiyo  iinp
--------------------------------
 1   2    0    1     0     1
 1   2    0    1     0     1
```

Ambas salidas son factibles, depende de la implementación (compilador). Lo relevante (semántica) es el número de hilos desplegados, no quién los despliega.

```
#include <omp.h>
int main () {
  int yo, iyo, iiyo, np, inp, iinp;
  omp_set_nested(1);
  printf("yo  np  iyo  inp  iiyo  iinp\n");
  #pragma omp parallel num_threads(2)
  {
        ...
```

¿Y ahora?

CONSTRUCTOR PARALLEL

yo	np	iyo	inp	iiyo	iinp
1	2	0	3	0	3
1	2	0	3	0	3
1	2	0	3	1	3
1	2	0	3	1	3
1	2	1	3	0	3
1	2	1	3	1	3
1	2	1	3	2	3
1	2	0	3	2	3
1	2	2	3	1	3
1	2	2	3	0	3
1	2	0	3	2	3
1	2	2	3	2	3
1	2	2	3	0	3
1	2	2	3	1	3
1	2	1	3	0	3
1	2	1	3	1	3
1	2	1	3	2	3
1	2	2	3	2	3

yo	np	iyo	inp	iiyo	iinp
0	2	0	3	0	3
1	2	0	3	0	3
0	2	2	3	0	3
0	2	0	3	2	3
0	2	0	3	1	3
1	2	0	3	1	3
0	2	2	3	1	3
1	2	0	3	2	3
0	2	2	3	2	3
1	2	1	3	1	3
1	2	1	3	2	3
0	2	1	3	2	3
0	2	1	3	1	3
0	2	1	3	0	3
1	2	1	3	0	3
1	2	2	3	0	3
1	2	2	3	2	3
1	2	2	3	1	3

CONSTRUCTOR PARALLEL

export OMP_MAX_ACTIVE_LEVELS=x>=0

```
0  1  0  1  0  1
```

x=0

```
1  2  0  1  0  1
1  2  0  1  0  1
```

x>0

```
0  1  0  1  0  1
```

x=0

```
1  2  0  1  0  1
1  2  0  1  0  1
```

x=1

x=2

x>2

```
0  2  0  3  0  1
0  2  2  3  0  1
0  2  1  3  0  1
1  2  0  3  0  1
1  2  1  3  0  1
1  2  2  3  0  1
```

```
1  2  0  3  0  3  |  1  2  0  3  1  3
0  2  0  3  0  3  |  1  2  2  3  2  3
0  2  2  3  2  3  |  1  2  2  3  0  3
0  2  2  3  0  3  |  1  2  2  3  1  3
1  2  0  3  2  3  |  0  2  1  3  2  3
0  2  0  3  1  3  |  ..............................
```

```
#include <omp.h>
int main () {
  int yo, iyo, iiyo, np, inp,
iinp;
  omp_set_nested(0);
            ...
```

```
#include <omp.h>
int main () {
  int yo, iyo, iiyo, np, inp,
iinp;
  omp_set_nested(1);
            ...
```

WORKSHARING CONSTRUCTS (WC)

Propiedades

- Un WC es un constructor de trabajo compartido.
- No hay sincronismo de entrada pero sí de salida, salvo uso de *nowait*.
- Los WC no implican ejecución paralela.

Tipos

De bucles (*for*), de secciones (*sections*) y *single*.

Cláusulas

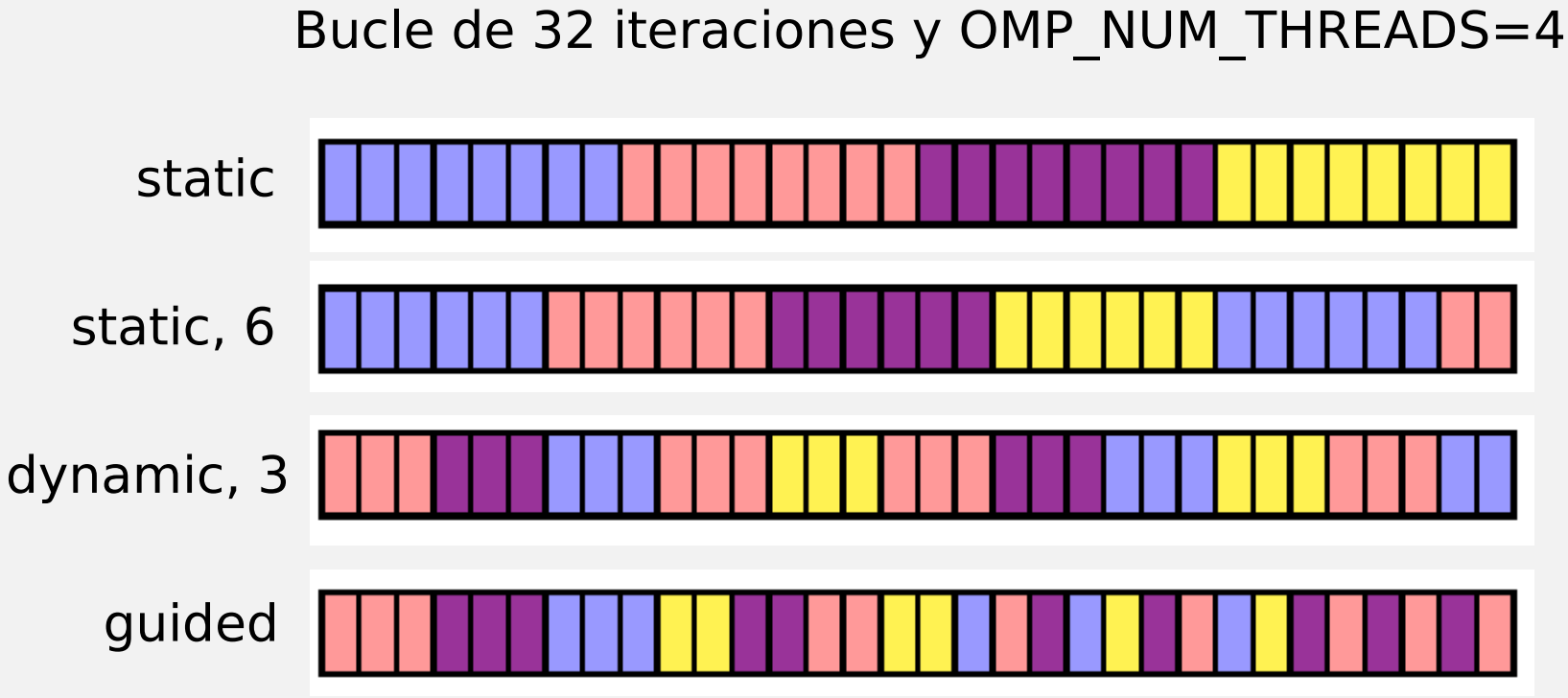
- *private*, *firstprivate*, *reduction*, *shared* y *nowait* (ya vistas)
- *lastprivate(lista)* El valor final de las variables de la lista es el que tendrían al final de la ejecución del secuencial equivalente.
- *ordered* Para ordenar secuencialmente las “iteraciones” entre los hilos.
- *collapse(valor)* Número de bucles léxicamente consecutivos que serán asociados al WC.

WORKSHARING CONSTRUCTS (WC)

Cláusulas

- *schedule(Tipo [, Chunk_size])* Controla cómo se distribuye el trabajo entre los hilos. *Chunk_size* es opcional y expresa el tamaño de los bloques en que se divide el trabajo.
  - Tipo ***static*** Por defecto. Antes de iniciar la ejecución divide el trabajo en un número de bloques que:
    - Si no se usa *Chunk_size* será el resultado de dividir el rango entre el número de hilos. Todos los bloques serán de igual tamaño salvo el último que puede ser menor.
    - Si se usa *Chunk_size* será el resultado de dividir el rango entre el *Chunk_size* resultando más/menos bloques que hilos. Se hace una asignación cíclica (*round-robin*).
  - Tipo ***dynamic*** Cada hilo recibe un bloque y al terminar solicita más (asignación bajo demanda: *first-come first-served*). Si no se usa *Chunk_size* tamaño del bloque = 1. Más sobrecarga pero mejor balanceo.
  - Tipo ***guided*** Como *dynamic* pero el tamaño va decreciendo exponencialmente (proporcional a la carga de trabajo que falte por realizar y al número de hilos) desde un valor inicial (depende de la implementación) hasta un final (*Chunk_size* o 1 si no se especifica).
  - Tipo ***runtime*** Se decide en tiempo de ejecución en función de la variable de entorno OMP_SCHEDULE.
  - Tipo ***auto*** Se delega en el compilador/entorno. El programador no tiene ningún tipo de control.

WORKSHARING CONSTRUCTS (WC)



WC FOR

```
#pragma omp for [cláusulas]
    for loop
```

- Útil en descomposiciones de datos “regulares”: Paralelismo de datos.
- No implica ejecución paralela, simplemente la ejecución de las iteraciones por los hilos del equipo (*team*). Qué y cuántas iteraciones ejecuta cada hilo depende del gestor, la política seleccionada y de las cláusulas usadas.
- Por defecto la variable de control es privada. El resto compartidas.
- Si se usa *collapse(n)* los espacios de iteración de los *n* primeros *for* se fusionarán en uno mayor que será distribuido entre los hilos según las políticas usadas. La ejecución secuencial de las iteraciones de los *for* asociados determina cómo será distribuido el nuevo espacio entre los hilos.

WC FOR

```
#include <omp.h>
int main ()
{
  int i;

 #pragma omp parallel [private(i)]
   #pragma omp for schedule(dynamic, 1)
   for (i=0; i<10; i++)
       printf("(%d, %d, %d)\n", omp_get_thread_num(), omp_get_num_threads(), i);

  return 0;
}
```

(0, 8, 1)	**(5, 8, 7)**
(1, 8, 2)	**(6, 8, 4)**
(2, 8, 5)	**(7, 8, 0)**
(3, 8, 3)	**(7, 8, 8)**
(4, 8, 6)	**(7, 8, 9)**

Pero también...

Ojo con el compilador y la planificación seleccionada

(0, 8, 1)	**(0, 8, 7)**
(0, 8, 2)	**(0, 8, 4)**
(0, 8, 5)	**(0, 8, 0)**
(0, 8, 3)	**(0, 8, 8)**
(0, 8, 6)	**(0, 8, 9)**

WC FOR

```
#include <omp.h>
int main ()
{
  int i, j, k;
  #pragma omp parallel num_threads(4)
    #pragma omp for private(j, k)
      for (i=0; i<2; i++)
        for (j=0; j<3; j++)
          for (k=1; k<2; k++)
              printf("%d con (i, j, k)=(%d, %d, %d)\n", omp_get_thread_num(), i, j, k);
  return 0;
}
```

```
0 con (i, j, k)=(0, 0, 1)
0 con (i, j, k)=(0, 1, 1)
0 con (i, j, k)=(0, 2, 1)
1 con (i, j, k)=(1, 0, 1)
1 con (i, j, k)=(1, 1, 1)
1 con (i, j, k)=(1, 2, 1)
```

WC FOR

```
#include <omp.h>
int main ()
{
  int i, j, k;
  #pragma omp parallel num_threads(4)
    #pragma omp for private(j, k) collapse(2)
      for (i=0; i<2; i++)
        for (j=0; j<3; j++)
          for (k=1; k<2; k++)
              printf("%d con (i, j, k)=(%d, %d, %d)\n",
omp_get_thread_num(), i, j, k);
  return 0;
}
```

0 con (i, j, k)=(0, 0, 1)
0 con (i, j, k)=(0, 1, 1)
2 con (i, j, k)=(1, 1, 1)
3 con (i, j, k)=(1, 2, 1)
1 con (i, j, k)=(0, 2, 1)
1 con (i, j, k)=(1, 0, 1)

Diferente según las políticas del compilador

0 con (i, j, k)=(0, 0, 1)
0 con (i, j, k)=(0, 1, 1)
1 con (i, j, k)=(0, 2, 1)
1 con (i, j, k)=(1, 0, 1)
2 con (i, j, k)=(1, 1, 1)
2 con (i, j, k)=(1, 2, 1)

WC FOR

```
#include <omp.h>
int main ()
{
  int yo, i;
  #pragma omp parallel private(yo, i) num_threads(4)
  {
    yo = omp_get_thread_num();
     #pragma omp for
       for (i=0;i<4;i++)  printf("%d en orden con i %d\n", yo,
i);
  }
  return 0;
}
```

0 en orden con i 0
1 en orden con i 1
3 en orden con i 3
2 en orden con i 2

Cualquiera de ellas u otras

0 en orden con i 1
1 en orden con i 0
3 en orden con i 2
2 en orden con i 3

WC FOR

```
#include <omp.h>
int main ()
{
  int yo, i;
  #pragma omp parallel private(yo, i) num_threads(4)
  {
    yo = omp_get_thread_num();
    #pragma omp for ordered
       for (i=0;i<4;i++)  printf("%d en orden con i %d\n", yo, i);
  }
  return 0;
}
```

```
0 en orden con i 0
1 en orden con i 1
3 en orden con i 3
2 en orden con i 2
```

WC FOR

schedule([modificador [, modificador] :] …)

- Nuevo calificador desde la especificación 4.5. **modificador** puede ser: *monotonic* (los *chunks* se asignan a los hilos siguiendo un orden lógico iterativo incremental, *nonmonotonic* (cualquier orden) o *simd*. Defecto *monotonic*.
- Ver documentación de la especificación 4.5 para más detalle.

linear(list [: step list])

- Nueva cláusula desde la especificación 4.5. Ver documento de la especificación para más información.

WC SECTIONS

```
#pragma omp sections [cláusulas]
{
    [#pragma omp section]
        structured block
    [#pragma omp section]
        structured block
    ...
}
```

- Útil en descomposiciones funcionales: constructor no iterativo.
- Si el número de hilos es menor alguno ejecutará varias secciones. Si es mayor algunos no ejecutarán nada.
- La política de asignación de las secciones a los hilos depende de la implementación. Sincronismo implícito al final salvo que se use *nowait*.
- Todos los *structured block* deben ir precedidos por la directiva *section*, exceptuando el primero para el que es opcional. Cada *section* se ejecuta una sola vez.

WC SECTIONS

```
#include <omp.h>
int main () {
    #pragma omp parallel
        #pragma omp sections nowait
        {
            printf("Hola desde %d\n", omp_get_thread_num());
          #pragma omp section
            printf("¿Qué tal desde %d?\n", omp_get_thread_num());
          #pragma omp section
            printf("Regular desde %d?\n", omp_get_thread_num());
        }
    return 0;
}
```

export OMP_NUM_THREADS=1
./55
¿Resultado de la ejecución?
Hola desde 0
¿Que tal desde 0?
Regular desde 0?

export OMP_NUM_THREADS=4
./55
¿Resultado de la ejecución?
¿Que tal desde 1?
Hola desde 0
Regular desde 3?

WC SINGLE

#pragma omp single [cláusulas]
structured block

- El primer hilo que alcance el constructor ejecuta el código. El resto esperan en la espera implícita de salida de *single*, salvo uso de *nowait*. Si se usa la cláusula *copyprivate* el hilo hace difusión de los valores de las variables de la lista al resto de hilos.

```
#include <omp.h>
int main () {
  int yo;
  #pragma omp parallel private(yo)
  {
    yo = omp_get_thread_num();
    #pragma omp single
      printf("Soy %d actuando en solitario dentro de single\n",yo);
  }
  return 0;
}
```

COMBINED CONSTRUCTORS (CC)

- Combinan constructores de paralelismo con otro tipo de constructores (WC, SIMD, TASK, etc.)

- Las cláusulas admisibles (y su semántica) para cada CC serán las de los mecanismos representados, pudiendo haber exclusión, como por ejemplo, la cláusula *nowait* que no puede usarse con el CC *for*.

```
#pragma omp parallel for
[cláusulas]
      for loop

#pragma omp parallel sections
[cláusulas]
{
      sections
}
```

CC FOR

```
suma = 0;
#pragma omp parallel for
for (i=0; i<n; i++)
    suma = suma + x[i]*y[i];
```

Incorrecto: *race condition* en suma (serialización) y el final valor indefinido.

```
suma = 0;
#pragma omp parallel for private(suma)
for (i=0; i<n; i++)
    suma = suma + x[i]*y[i];
```

Incorrecto: suma en los hijos puede no ser inicializada y al final su valor = 0

```
suma = 0;
#pragma omp parallel for reduction(+:suma)
for (i=0; i<n; i++)
    suma = suma + x[i]*y[i];
```

¿Analizamos su rendimiento? ¿Por dónde empezar?

CC FOR

```
#pragma omp parallel for private(j)
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      ...
```

```
for (i=0; i<n; i++)
   #pragma omp parallel for
   for (j=0; j<m; j++)
      ...
```

- En el primero las iteraciones de *i* se reparten entre los hilos. Cada hilo ejecuta el bucle *j* completo. En el segundo para cada *i* se activan y desactivan los hilos. Hay *n* sincronizaciones.
- En anidados habitualmente se paralelizan los externos por ser el coste de gestión de hilos menor.
- Si las dependencias de datos impiden paralelizar el bucle externo reorganizar los bucles, salvo impacto negativo en el uso de la caché.

CONSTRUCTOR SIMD (CSIMD)

```
#pragma omp simd [cláusulas]
    structured block
```

Desde la especificación 4.0, mejorado en la 4.5.

Varias iteraciones del bucle se ejecutan concurrentemente usando instrucciones SIMD: *vectorización*. Aplicable a bucles en *forma canónica*. Las iteraciones del bucle son (internamente) numeradas de 0 a n-1.

Cláusulas

- *collapse*, *reduction*, *lastprivate, linear* y *private* (ya comentadas)
- *simdlen(length)* Número de iteraciones a ejecutar concurrentemente.
- *safelen(length)* Dos iteraciones concurrentes no pueden estar a una distancia en su "numeración" mayor que *length*.
- *aligned(list [: alignment])* Los objetos de *list* se alinean en memoria al número de bytes indicado en *alignment*.

CSIMD

```
#include <omp.h>
int main () {
    ...
    #pragma omp simd
    for (i=0; i<1E+8; i++)
        MatA[i] += MatB[i]*MatC[i];
    ...
}
```

Intel Core i5-7600 3.50GHz
icc –O0 sin simd 0.0087 s
icc –O0 con simd 0.2114 s
icc –O3 sin simd 0.0087 s
icc –O3 con simd 0.2113 s

gcc7 –O0 sin simd 0.286 s
gcc7 –O0 con simd 0.285 s
gcc7 –O3 sin simd 0.283 s
gcc7 –O3 con simd 0.287 s

Intel Core i3-2100 CPU 3.10GHz
icc –O0 sin simd 0.0207 s
icc –O0 con simd 0.3419 s
icc –O3 sin simd 0.0207 s
icc –O3 con simd 0.3422 s

gcc7 –O0 sin simd 0.4691 s
gcc7 –O0 con simd 0.4625 s
gcc7 –O3 sin simd 0.4673 s
gcc7 –O3 con simd 0.4743 s

Intel Xeon E5420 2.50GHz
icc –O0 sin simd 0.0274 s
icc –O0 con simd 0.7148 s
icc –O3 sin simd 0.0274 s
icc –O3 con simd 0.7149 s

gcc7 –O0 sin simd 0.8066 s
gcc7 –O0 con simd 0.7973 s
gcc7 –O3 sin simd 0.7933s
gcc7 –O3 con simd 0.7974 s

CSIMD

```
#include <omp.h>
int main () {
    ...
    #pragma omp simd aligned(MatA, MatB, MatC: 32)
    for (i=0; i<n*n; i++)
        MatA[i] += MatB[i]*MatC[i];
    ...
    return 0;
}
```

Note: Intel CPUs have 64 byte cache lines, and 32 byte vectors, best alignment is 32 bytes

CSIMD

```
float fun(float a, float * b, float c);
{
    ...
   return ...;
}

int main () {
    float tmp;
    int i;
    #pragma omp simd
    for (i=0; i<n; i++)
        int j = Vec[i];
        Mat[j] += fun(tmp, &Vec[i],
Mat[j]);
    ...
    return 0;
}
```

- No vectoriza a menos que *fun* sea *inline*.
- Más aún (peor aún) solo es correcto si todos los elementos de *Vec* son distintos.
- Si los elementos del vector no son distintos se debe usar *#pragma omp ordered simd*.

CSIMD

```
#pragma omp declare simd uniform(a, b, valor) linear(i:1)
double fun1(double *a, double *b, double valor, int i) {
    double c;
    c = a[i] + b[i] + valor;
    return c;
}

int main () {
    double *a, *b; ...;
    #pragma omp simd
    for (i=0; i<n; i++)
        a[i] = fun1(a, b, 1.0, i);
    ...
    return 0;
}
```

- Cuando una función es potencialmente *inline* se puede vectorizar el bucle declarando la función como *vectorizable*.
- Se deben caracterizar los argumentos para ayudar al compilador: **uniform** → argumento constante y **linear(var: step)** → **var ± step**.

CSIMD

- Aunque sea solo un hilo el que ejecute el constructor *simd* hay ejecución *vectorizada* (paralelismo) de instrucciones.
- Es por ello que el programador debe garantizar la consistencia haciendo privadas ciertas variables (igual que en los otros constructores).

```
double work( double *a, double *b, int n )
 {
    int i;
   double  sum, tmp;

   sum = 0.0;
   #pragma omp simd private(tmp)
reduction(+:sum)
   for (i = 0; i < n; i++)
      sum += a[i] + b[i];
      tmp = ....
   return sum;
}
```

CC CSIMD

#pragma omp for simd [cláusulas]
structured block

- El *structured block* será ejecutado en paralelo por los hilos del *team* y, además, las instrucciones que así lo permitan usando *simd*. Las cláusulas admisibles son todas las de *for* y *simd*.

Ejemplo traspa 61 en Intel Core i5-7600 3.50GHz	
Normal con –O0	0.287-0.253 s (double – float)
Normal con –O3	0.288-0.252 s (double – float)
pragma omp parallel for con –O0	0.206-0.103 s (double – float)
pragma omp parallel for con –O3	0.206-0.103 s (double – float)
pragma omp simd con –O0	0.285-0.251 s (double – float)
pragma omp simd con –O3	0.282-0.251 s (double – float)
pragma omp for simd con –O0	0.286-0.250 s (double – float)
pragma omp for simd con –O3	0.285-0.250 s (double – float)

- El rendimiento final depende de la programación, las cláusulas, el compilador (ivdep), etc.

CONSTRUCTOR TASK (CTASK)

Útil para paralelismo no estructurado (bucles con condición de parada dependiente de los datos, tamaños no conocidos inicialmente, etc.)

Definiciones previas

- *Explicit task* Tarea generada por el constructor *task*.
- *Implicit task* Tarea generada por el constructor *parallel,* regiones de paralelismo implícito, etc.
- *Tied task* Tarea cuya ejecución (suspendida previamente) solo puede ser reanudada por el hilo que suspendió su ejecución. Por defecto.
- *Untied task* Tarea cuya ejecución puede ser reanudada por cualquier hilo del equipo.
- *Undeferred task* Tarea que suspende su ejecución, para crear y ejecutar una nueva. Reanuda su ejecución cuando termina la nueva.
- *Included task* Es del tipo *undeferred*. Ejecución inmediata por el hilo.

CTASK

Filosofía

- Los hilos crean las tareas y por defecto las ejecutan de inicio a fin (de forma inmediata o no). También puede diferir su ejecución a otro hilo.
- La ejecución de las tareas se puede suspender y reanudar en los *Task Scheduling Points* (TSP) = {*pragma task*, *pragma taskwait*, *barrier*, finalización de la tarea, barreras implícitas, etc.}
- Cuando un hilo alcanza un TSP puede conmutar, iniciar o reanudar cualquier tarea pendiente.
- La terminación correcta de las tareas se garantiza con los constructores de sincronización explícitos (p. ej. *taskwait* y *barrier*) o por los implícitos existentes en algunas directivas/constructores.
- Precaución con las variables *threadprivate* puesto que su contenido puede ser cambiado por cualquier tarea ejecutada por el mismo hilo.
- El comportamiento *shared* por defecto no es aplicable a *task* (recordar características del modelo *fork/join*).

CTASK

#pragma omp task [cláusulas]
structured block

No implica ejecución paralela. El hilo crea la tarea. Admite anidamiento. La región de la tarea interna (código) no forma parte de la tarea externa.

Cláusulas

- *firstprivate*, *shared*, *default*, *private* (ya vistas).
- *If([task:] exp)* Si se evalúa a falso se genera una *undeferred task*.
- *final(exp)* Si *exp* se evalúa a cierto la nueva tarea (y todas sus hijas) será *final task* (*omp_in_final()* permite saber si la tarea es, o no, final).
- *untied* Precaución: no hacer referencia al identificador del hilo en la tarea.
- *mergeable* Si la nueva tarea es *undeferred* o *included* se puede fusionar con la generadora. La fusión, o no, depende de la implementación.
- *priority(value)* Mayor valor, mayor prioridad. Cero es la menor prioridad.
- *depend(dependece-type: list)* Restricciones de ejecución: in, out, inout.

CTASK

#pragma omp taskyield

Suspende la ejecución de la tarea.

#pragma omp taskwait

Es un *TSP*. Suspende la ejecución de la tarea hasta que todas las tareas creadas previamente en la misma región paralela concluyan su ejecución.

```
#pragma omp taskgroup
    structured block
```

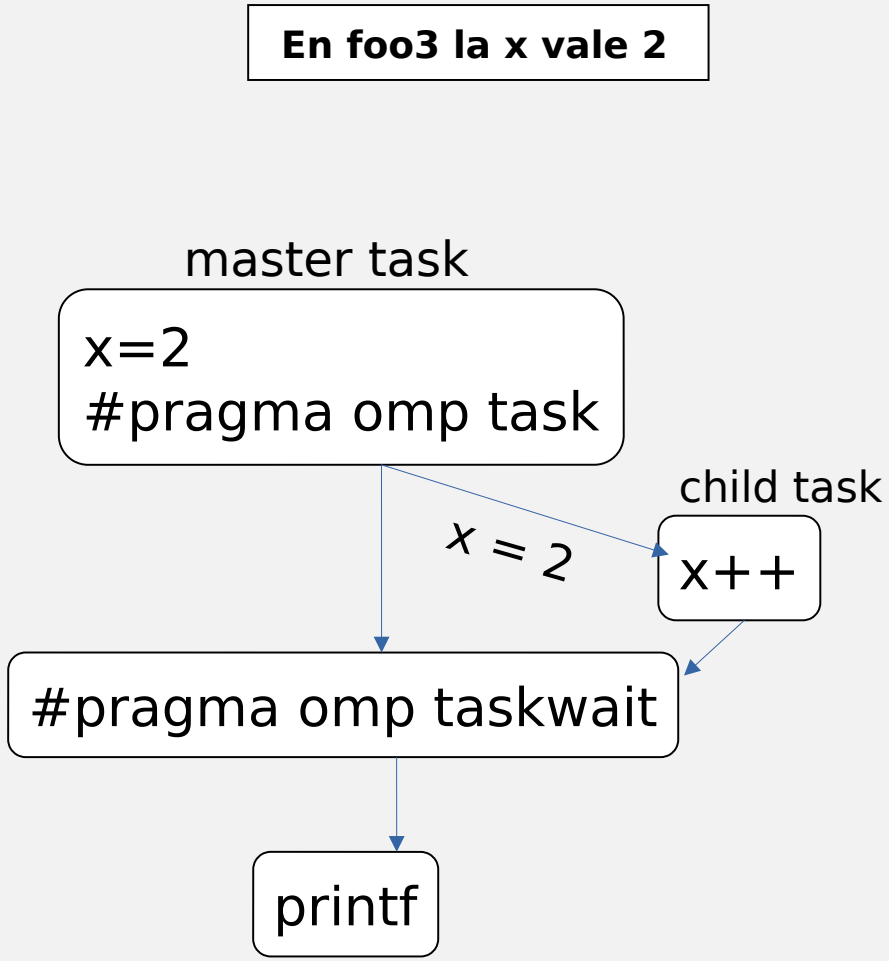
Todas las tareas generadas dentro de la región *taskgroup*, y todas sus descendientes, forman un grupo. Tiene un *TSP* al final, esto es, la tarea suspende su ejecución hasta que todas las tareas del grupo terminen su ejecución.

taskwait espera por todas las tareas previas adscritas a la misma región paralela. *taskgroup* solo espera por las de la región paralela que forman parte del grupo.

CTASK

```
#include <omp.h>
void foo3() {
   int x = 2;
   #pragma omp task
      x++;
   [ #pragma omp taskwait ]
   printf("En foo3 la x vale %d\n",x);
}

int main(int argc, char * argv[]) {
  foo3();
  return 0;
}
```



CTASK

```
#include <omp.h>
void foo3() {
   int x = 2;
   #pragma omp task
      x++;
   [ #pragma omp taskwait ]
   printf("En foo3 la x vale %d\n",x);
}
void foo2() {
   int x = 2;
   #pragma omp task mergeable
      x++;
   [ #pragma omp taskwait ]
   printf("En foo2 la x vale %d\n",x);
}
int main(int argc, char * argv[]) {
  foo2(); foo3();
  return 0;
}
```

En foo2 la x vale 2
En foo3 la x vale 2

En foo2 la x vale 3
En foo3 la x vale 2

foo2() *not safe*

Ambos resultados son factibles: depende de los detalles de implementación del compilador: fusión sí o no

CTASK

```
#include <omp.h>
void foo3() {
   int x = 2;
   #pragma omp task
      x++;
   [ #pragma omp taskwait ]
   printf("En foo3 la x vale %d\n",x);
}
void foo2() {
   int x = 2;
   #pragma omp task shared(x)
      x++;
   [ #pragma omp taskwait ]
   printf("En foo2 la x vale %d\n",x);
}
int main(int argc, char * argv[]) {
  foo2(); foo3();
  return 0;
}
```

En foo2 la x vale 3
En foo3 la x vale 2

CTASK

```
#include <omp.h>
int comp_fib_numbers(int n)
{
    int a, b;
    if (n < 2)
        return n;
    else
    {
        #pragma omp task shared(a) untied
            a = comp_fib_numbers(n-1);

        #pragma omp task shared(b) untied
            b = comp_fib_numbers(n-2);

        #pragma omp taskwait
            return a + b;
    }
}
```

```
int main(int argc, char * argv[])
{
    int result, N;
    N = atoi(argv[1]);

    #pragma omp parallel
        #pragma omp single nowait
            result = comp_fib_numbers(N);

   printf("fibo(%d) = %d\n", N, result);
   return 0;
}
```

CTASK

```
#define LIMIT 3                    /* limite para la profundidad en la recursión */
void Buscar(int pos, int n, char *state) {
    if (pos == n)
    {
        validar_solucion(state);
        return;
    }
    #pragma omp task final(pos > LIMIT) mergeable
    {
        char new_state[n];
        if  (!omp_in_final())
        {
            memcpy(new_state, state, pos );
            state = new_state;
        }
        state[pos] = 0;
        Buscar(pos+1, n, state );
    }
```

CTASK

```
int main(int argc, char * argv[]) {
   int x = 100;
   #pragma omp task
      x += 1;
   #pragma omp task
      x *= 2;
   printf("La x vale %d\n",x);

   #pragma omp parallel
      #pragma omp single
      {
         #pragma omp task
           x += 1;
        #pragma omp task
           x *= 2;
      }
    printf("Y ahora vale %d\n",x);
    return 0;
}
```

La x vale 100

Y ahora vale 202

Recordar

El comportamiento *shared* por defecto no es aplicable a *task* pero sí al resto

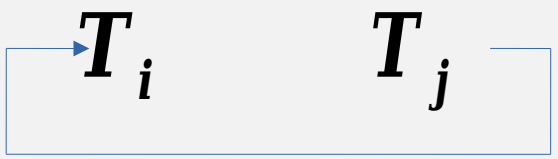
CTASK

```
#include <stdio.h>
#include <omp.h>
int main()
{
   int x = 100;
   printf("La x vale %d\n", x);
   #pragma omp parallel
      #pragma omp single
      {
         #pragma omp task depend(out: x)
            x += 1;
         #pragma omp task depend(in: x)
            x *= 2;
      }
   printf("Y ahora vale %d\n", x);
   return 0;
}
```

La x vale 100
Y ahora vale 202

CTASK

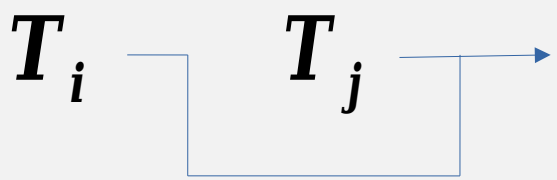
```
#include <stdio.h>
#include <omp.h>
int main()
{
   int x = 100;
   printf("La x vale %d\n", x);
   #pragma omp parallel
      #pragma omp single
      {
         #pragma omp task depend(in: x)
            x += 1;
         #pragma omp task depend(out: x)
            x *= 2;
      }
   printf("Y ahora vale %d\n", x);
   return 0;
}
```



La x vale 100
Y ahora vale 202

CTASK

```
#include <stdio.h>
#include <omp.h>
int main()
{
   int x = 100;
   printf("La x vale %d\n", x);
   #pragma omp parallel
      #pragma omp single
      {
         #pragma omp task depend(out: x)
            x += 1;
         #pragma omp task depend(out: x)
            x *= 2;
      }
   printf("Y ahora vale %d\n", x);
   return 0;
}
```



La x vale 100
Y ahora vale 202

CTASK

```
#include <stdio.h>
#include <omp.h>
int main()
{
   int x = 100;
   printf(“La x vale %d\n”, x);
   #pragma omp parallel
      #pragma omp single
      {
         #pragma omp task depend(in: x) priority(0)
            x += 1;
         #pragma omp task depend(out: x) priority(10) untied
            x *= 2;
      }
   printf(“Y ahora vale %d\n", x);
   return 0;
}
```

La x vale 100
Y ahora vale 202

CTASK

Orden	Operación	Dependencia	Resultado
1ª 2ª	x += 1 x *= 2	Out In	202
1ª 2ª	x += 1 x *= 2	In Out	202
1ª 2ª	x += 1 x *= 2	Out Out	202
1ª 2ª	x += 1 x *= 2	Out baja prioridad Out alta prioridad	202
1ª 2ª	x *= 2 x += 1	Out In	201
1ª 2ª	x *= 2 x += 1	In Out	201
1ª 2ª	x *= 2 x += 1	Out Out	201
1ª 2ª	x *= 2 x += 1	Out baja prioridad Out alta prioridad	201

CTASK

En resumen

- Además de las dependencias que establece la cláusula **depend** también el orden de creación de las tareas en el código fuente influye las dependencias.

¿Por qué los resultados no son los esperados?

- Porque si no hay tareas en las colas que tengan dependencias **out** o **inout** sobre la variable de una tarea previa en el código con dependencia **in** la tarea está lista para ejecutarse inmediatamente y la implementación, probablemente, lo hará incluso antes de que se encuentren las siguientes tareas.

¿Cómo solucionarlo?

- Colocando las tareas correctamente en el código.
- Usando dependencias tipo **inout**.
- Nota. Todos los programas de las transparencias previas seguirán imprimiendo 202/201 como hasta ahora, pero ya sabemos que están haciendo lo esperado (y formalmente correctos si se usa **inout**).

CC CTASK

#pragma omp taskloop [cláusulas]
for-loop

Las iteraciones del bucle, o bucles si se usa *collapse*, se ejecutarán en paralelo usando *tasks*. Por defecto *taskloop* crea de forma implícita un *taskgroup*.

Cláusulas

- *firstprivate*, *shared*, *default*, *private*, *lastprivate*, *collapse*, *final*, *priority*, *untied*, *mergeable*, *if* (ya vistas)

- *grainsize(exp)* El número (*n*) de iteraciones asignadas a las tareas estará comprendido entre: *min(exp, n.º iteraciones del bucle <=n<=2*exp)*.

num_tasks(num) El número de tareas es *min(exp, n.º iteraciones del bucle)*

- *nogroup* No crea la región *taskgroup*.

CC CTASK Y AVANZADOS

#pragma omp taskloop simd [cláusulas]
for-loop

Las iteraciones del bucle se ejecutarán en paralelo usando *tasks* y *simd*.

Las cláusulas serán las de *taskloop* y *simd*.

Avanzados o …

Device constructors (target)
Teams constructors
Distributed constructors
Combined (i.e. #pragma omp target teams distribute parallel for simd)

- Soporte *offload* pensado para el uso de varios dispositivos, incluidas aceleradoras.
- Fuera del alcance de este curso.

- OpenACC, OpenCL, OneAPI, NVIDIA DevTools, …

CONSTRUCTORES DE SINCRONIZACIÓN

```
#pragma omp master
    structured block
```

- Solo el hilo maestro ejecuta *structured block*. No hay esperas implícitas para el resto de los hilos.

```
#pragma omp barrier
```

- Espera o sincronización explícita de todos los hilos del equipo.

```
#pragma omp critical [(nombre) [int(expression)]]
    structured block
```

- *"... in OpenMP all unnamed critical sections are considered identical (if you prefer, there's only one lock for all unnamed critical sections), so that if one thread is in one [unnamed] critical section, no thread can enter any [unnamed] critical section. As you might guess, you can get around this by using named critical sections. In OpenMP, all the unnamed critical sections are mutually exclusive... "*

CONSTRUCTORES DE SINCRONIZACIÓN

```
#include <omp.h>
int main ()
{
   int yo;
   #pragma omp parallel private(yo)
   {
      yo = omp_get_thread_num();

      #pragma omp critical
      {
         printf("Soy %d al inicio de la seccion critica\n", yo);
         sleep(1);
         printf("Soy %d al final  de la seccion critica\n", yo);
      }
   }
   return 0;
}
```

```
export OMP_NUM_THREADS=8
./86
     Soy 0 al inicio de la seccion critica
     Soy 0 al final  de la seccion critica
     Soy 3 al inicio de la seccion critica
     Soy 3 al final  de la seccion critica
     Soy 2 al inicio de la seccion critica
     Soy 2 al final  de la seccion critica
     Soy 1 al inicio de la seccion critica
     Soy 1 al final  de la seccion critica
     Soy 4 al inicio de la seccion critica
     Soy 4 al final  de la seccion critica
     Soy 5 al inicio de la seccion critica
     Soy 5 al final  de la seccion critica
     Soy 6 al inicio de la seccion critica
     Soy 6 al final  de la seccion critica
     Soy 7 al inicio de la seccion critica
     Soy 7 al final  de la seccion critica
```

CONSTRUCTORES DE SINCRONIZACIÓN

max = min = 1.0; n=1000000;
“double a[1000000] = {0,2,4….}”

```
#pragma omp parallel for
num_threads(4)
for (i=0; i<n; i++)
{
   if (a[i] >= max)
   {
      #pragma omp critical
       if (a[i] >= max)
          max = a[i];
   }
   if (a[i] <= min)
   {
      #pragma omp critical
      if (a[i] <= min)
         min = a[i];
   }
}
```

(10^6/4) * 2“conflictos” sobre **max + min**

Tiempo con gcc: 14.2 sec.
Tiempo con icc: 27.8 sec.

CONSTRUCTORES DE SINCRONIZACIÓN

```
max = min = 1.0; n=1000000;
“double a[1000000] = 1.0”
#pragma omp parallel for
num_threads(4)
for (i=0; i<n; i++)
{
   if (a[i] >= max)
   {
      #pragma omp critical (maximo)
       if (a[i] >= max)
          max = a[i];
   }
   if (a[i] <= min)
   {
      #pragma omp critical (minimo)
      if (a[i] <= min)
         min = a[i];
   }
}
```

Tiempo con gcc: 14.2 sec.
Tiempo con icc: 27.8 sec.

Y ahora...

Tiempo con gcc: 11.6 sec.
Tiempo con icc: 14.8 sec.

El rendimiento es mejor.
Y la gestión de icc mucho peor, en este caso

CONSTRUCTORES DE SINCRONIZACIÓN

#pragma omp atomic [read | write | update | capture]
expression-stmt | structured block

Asegura la ejecución de varios hilos paralelos de forma atómica. Es más eficiente que *critical*.

expression-stmt debe ser:

- v=x si *read*
- *x*=expresión si *write*
- *x*++, *x*--, ++*x*, --*x*, *x binop* = expresión, ... si *update* o *none*
- v=*x*++, v=*x*--, v=++*x*, v=--*x*, v=*x binop* = expresión
  {v=*x*; *x binop* = expresión}, ... si *capture* siendo *binop* +, *, -, /, &, ^, |, <<, >>

Mirar manual para *structured block*.

FUNCIONES PARA MEDIR EL TIEMPOS

double omp_get_wtime(void)

- Retorna los segundos transcurridos desde un tiempo pasado.

double omp_get_wtick(void)

- Retorna la precisión del reloj usado por **omp_get_wtime**

```
#include <omp.h>
int main ()
{
  double start=0.0, end=0.0;
  start = omp_get_wtime();
      ... Trabajo a monitorizar ...
  end = omp_get_wtime();
  printf("Tiempo transcurrido %3.5E segundos\n",
end - start);
  return 0;
}
```

FUNCIONES DE BLOQUEO

void omp_init_lock(omp_lock_t *lock)

- Crear una llave o cerrojo. Se crea en estado no-bloqueado (*unlocked*).

void omp_destroy_lock(omp_lock_t *llave)

- Destruye una llave o cerrojo.

void omp_set_lock(omp_lock_t *llave)

- Bloquear una llave (poner en estado *locked*).

void omp_unset_lock(omp_lock_t *llave)

- Desbloquear un cerrojo (poner en estado *unlocked*).

int omp_test_lock(omp_lock_t *llave)

- Retorna cierto cuando bloquea la llave. Si no puede bloquear retorna falso y continúa, no es bloqueante.

FUNCIONES DE BLOQUEO

Las anteriores **NO** permiten establecer (*lock*) múltiples veces un cerrojo por la misma tarea. Las siguientes (xxx_**nest**_xxx), **SÍ**.

```
void omp_init_nest_lock(omp_lock_t *lock)

void omp_destroy_nest_lock(omp_lock_t *llave)

void omp_set_nest_lock(omp_lock_t *llave)

void omp_unset_nest_lock(omp_lock_t *llave)

int omp_test_nest_lock(omp_lock_t *llave)
```

En todas (normales y _*nest*_ cuando estén soportadas) los propietarios de los cerrojos son las regiones (regiones secuenciales, paralelas, etc.)

FUNCIONES DE BLOQUEO

```
#include <omp.h>
int main () {
  omp_lock_t llave;
  int yo, c=0;
  omp_init_lock(&llave);
  #pragma omp parallel private(yo)
num_threads(4)
  {
     yo = omp_get_thread_num();
     omp_set_lock(&llave);
       printf("%d paso primer lock\n", yo);
     omp_unset_lock(&llave);

     while (! omp_test_lock(&llave))
         printf(“%d espera segundo lock\n“, yo);
     c++;
     omp_unset_lock(&llave);
  }
  omp_destroy_lock(&llave);
  printf(“La c al final vale %d\n", c);    return 0;
}
```

```
./93
3 paso primer lock
1 paso primer lock
1 espera segundo lock
1 espera segundo lock
1 espera segundo lock
1 espera segundo lock
0 paso primer lock
0 espera segundo lock
0 espera segundo lock
1 espera segundo lock
1 espera segundo lock
1 espera segundo lock
1 espera segundo lock
1 espera segundo lock
1 espera segundo lock
2 paso primer lock
3 espera segundo lock
1 espera segundo lock
0 espera segundo lock
La c al final vale 4
```

FUNCIONES DE BLOQUEO

```
#include <omp.h>
int main () {
  omp_lock_t llave;
  int yo, c=0;
  omp_init_lock(&llave);
   #pragma omp parallel private(yo)
num_threads(4)
  {
     yo = omp_get_thread_num();
      omp_set_lock(&llave);
      omp_set_lock(&llave);
        printf("%d paso primer lock\n", yo);
      omp_unset_lock(&llave);
      omp_unset_lock(&llave);
     while (! omp_test_lock(&llave))
         printf(“%d espera segundo lock\n“, yo);
     c++;
      omp_unset_lock(&llave);
  }
  omp_destroy_lock(&llave);
  printf(“La c al final vale %d\n", c); …
```

```
./94
```

Bloqueo, no termina no imprime nada

FUNCIONES DE BLOQUEO

```
#include <omp.h>
int main () {
  omp_nest_lock_t llave;
  int yo, c=0;
  omp_init_nest_lock(&llave);
  #pragma omp parallel private(yo) num_threads(4)
  {
     yo = omp_get_thread_num();
     omp_set_nest_lock(&llave);
     omp_set_nest_lock(&llave);
       printf("%d paso primer lock\n", yo);
     omp_unset_nest_lock(&llave);
     omp_unset_nest_lock(&llave);
     while (! omp_test_nest_lock(&llave))
         printf("%d espera segundo lock\n", yo);
     c++;
     omp_unset_nest_lock(&llave);
  }
  omp_destroy_nest_lock(&llave);
  printf("La c al final vale %d\n", c); ...
```

```
./95
0 paso primer lock
0 espera segundo lock
0 espera segundo lock
0 espera segundo lock
0 espera segundo lock
3 paso primer lock
3 espera segundo lock
3 espera segundo lock
2 paso primer lock
2 espera segundo lock
2 espera segundo lock
2 espera segundo lock
2 espera segundo lock
2 espera segundo lock
1 paso primer lock
3 espera segundo lock
2 espera segundo lock
0 espera segundo lock
La c al final vale 4
```

MÁS CONSTRUCTORES /CLÁUSULAS ÚTILES

#pragma omp flush [(lista)]

Actualiza “la visión temporal de memoria” de las variables contenidas en la lista del hilo que la ejecuta (coherencia).

La “visión temporal” del resto de hilos no se modifica necesariamente (necesitan sus propios *flush*).

En *barrier*, al entrar/salir en/de *critical/ordered/parallel*, al salir de *for*, *sections*, *single* y de los CPWC hay *flush* implícito, salvo uso de *nowait*.

EJEMPLOS

```
#include <omp.h>
int main () {
  int yo, x=0;
  #pragma omp parallel private(yo)
  {
     yo = omp_get_thread_num();
     printf("Soy %d antes flush x vale %d\n",yo, x);
     #pragma omp master
     {
        x=99;
        #pragma omp flush(x)
        printf(“Master actualizo x a 99\n”);
     }
     while (x==0) {
         #pragma omp flush(x)
     }
     printf("Soy %d after flush x vale %d\n",yo,x);
  }
  return 0;
}
```

Compilado con gcc

```
Soy 3 antes flush x vale 0
Soy 1 antes flush x vale 0
Soy 0 antes flush x vale 0
Soy 3 after flush x vale 99
Master actualizo x a 99
Soy 0 after flush x vale 99
Soy 1 after flush x vale 99
Soy 2 antes flush x vale 0
Soy 2 after flush x vale 99
```

Compilado con icc

```
Soy 0 antes flush x vale 0
Master actualizo x a 99
Soy 0 after flush x vale 99
Soy 1 antes flush x vale 99
Soy 1 after flush x vale 99
Soy 2 antes flush x vale 99
Soy 2 after flush x vale 99
Soy 3 antes flush x vale 99
Soy 3 after flush x vale 99
```

MÁS CONSTRUCTORES /CLÁUSULAS ÚTILES

#pragma omp threadprivate (lista)

Variables a replicar a cada hilo.

"*Convierte*" variables globales en locales y persistentes a los hilos a través de las múltiples regiones paralelas.

Solo se inicializan al principio (en la creación o réplica).

EJEMPLOS

```
#include <omp.h>

int main ()
{
   int yo;
   int x=-99;
   #pragma omp parallel private(yo)
   {
        yo = omp_get_thread_num();
        printf("%d antes x vale %d\n",yo, x);
        x=yo;
        printf("%d actualiza x a %d\n", yo, x);
   }
   printf("La x al final vale %d\n", x);
   return 0;
}
```

```
0 antes x vale -99
0 actualiza x a 0
1 antes x vale 0
1 actualiza x a 1
3 antes x vale 1
3 actualiza x a 3
2 antes x vale 3
2 actualiza x a 2
La x al final vale 2
```

EJEMPLOS

```
#include <omp.h>
int x=-69;
#pragma omp threadprivate(x)
int main ()
{
  int yo;
  int x=-99;
  #pragma omp parallel private(yo)
  {
      yo = omp_get_thread_num();
      printf("%d antes x vale %d\n",yo, x);
      x=yo;
      printf("%d actualiza x a %d\n", yo, x);
  }
  printf("La x al final vale %d\n", x);
  return 0;
}
```

Debe ser así. Probar a cambiar de lugar para ver qué hace el compilador

```
0 antes x vale -99
0 actualiza x a 0
1 antes x vale -69
1 actualiza x a 1
3 antes x vale -69
3 actualiza x a 3
2 antes x vale -69
2 actualiza x a 2
La x al final vale 0
```

EJEMPLOS

```
#include <omp.h>
int x=-69;
#pragma omp threadprivate(x)
int main ()
{
  int yo;
  int x=-99;
  #pragma omp parallel private(yo) copyin(x)
  {
      yo = omp_get_thread_num();
      printf("%d antes x vale %d\n",yo, x);
      x=yo;
      printf(“%d actualiza x a %d\n“, yo, x);
  }
  printf(“La x al final vale %d\n", x);
  return 0;
}
```

```
0 antes x vale -99
0 actualiza x a 0
1 antes x vale -99
1 actualiza x a 1
3 antes x vale -99
3 actualiza x a 3
2 antes x vale -99
2 actualiza x a 2
La x al final vale 0
```

EJEMPLOS

```
#include <omp.h>
float  x=1.2, y=2.1;
#pragma omp threadprivate(x, y)

void Inicializa (float *a, float *b) {
  #pragma omp single copyprivate(x, y)
        scanf("%f %f %f %f", a, b, &x, &y);
}
int main () {
   float a=3.4, b=5.6;
   #pragma omp parallel
   {
      Inicializa(&a, &b);
      printf("La salida es: %f %f %f %f\n", a, b, x, y);
   }
   return 0;
}
```

```
./102
9 8 7 6
```

La salida es:

```
9.000 8.000 7.000 6.000
9.000 8.000 7.000 6.000
9.000 8.000 7.000 6.000
9.000 8.000 7.000 6.000
```

EJEMPLOS

```
#include <omp.h>
float  x=1.2, y=2.1;
#pragma omp threadprivate(x, y)

void Inicializa (float *a, float *b) {
   #pragma omp single copyprivate(x, y)
         scanf("%f %f %f %f", a, b, &x, &y);
}
int main () {
    float a=3.4, b=5.6;
    #pragma omp parallel private(a, b)
    {
        Inicializa(&a, &b);
        printf("La salida es: %f %f %f %f\n", a, b, x, y);
    }
    return 0;
}
```

```
./103
9 8 7 6
```

La salida es:

```
X.XXX X.XXX 7.000 6.000
9.000 8.000 7.000 6.000
X.XXX X.XXX 7.000 6.000
X.XXX X.XXX 7.000 6.000
```

EJEMPLOS

```
#include <omp.h>
float  x=1.2, y=2.1;
#pragma omp threadprivate(x, y)

void Inicializa (float *a, float *b) {
   #pragma omp single copyprivate(x, y)
         scanf("%f %f %f %f", a, b, &x, &y);
}
int main () {
    float a=3.4, b=5.6;
    #pragma omp parallel firstprivate(a, b)
    {
        Inicializa(&a, &b);
        printf("La salida es: %f %f %f %f\n", a, b, x, y);
    }
    return 0;
}
```

```
./104
9 8 7 6
```

La salida es:
3.400 5.600 7.000 6.000
3.400 5.600 7.000 6.000
3.400 5.600 7.000 6.000
9.000 8.000 7.000 6.000

EJEMPLOS

```
#include <omp.h>
int main ()
{
  int yo, i, x=-99;
  #pragma omp parallel private(yo, i) num_threads(4)
  {
    yo = omp_get_thread_num();
    printf("%d antes con x=%d\n", yo,  x);
    #pragma omp for lastprivate(x)
       for (i=0; i<8 ; i++) {
            x=yo*i;
            printf(“%d con i %d modifica x a %d\n“, yo, i, x);
       }
    printf("%d after for, x vale %d\n", yo, x);
  }
  printf(“La x al final vale %d\n", x);
  …
```

Compilado con icc

```
0 antes con x=-99
0 con i 0 modifica x a 0
0 con i 1 modifica x a 0
1 antes con x=-99
1 con i 2 modifica x a 2
1 con i 3 modifica x a 3
3 antes con x=-99
3 con i 6 modifica x a 18
3 con i 7 modifica x a 21
2 antes con x=21
2 con i 4 modifica x a 8
2 con i 5 modifica x a 10
2 after for, x vale 21
0 after for, x vale 21
1 after for, x vale 21
3 after for, x vale 21
La x al final vale 21
```

EJEMPLOS

```
#include <omp.h>
int main ()
{
  int yo, i, x=-99;
  #pragma omp parallel private(yo, i) num_threads(4)
  {
    yo = omp_get_thread_num();
    printf("%d antes con x=%d\n", yo,  x);
    #pragma omp for lastprivate(x)
       for (i=0; i<8 ; i++) {
           x=yo*i;
           printf(“%d con i %d modifica x a %d\n“, yo, i, x);
       }
    printf("%d after for, x vale %d\n", yo, x);
  }
  printf(“La x al final vale %d\n", x);
  …
```

Compilado con gcc

```
0 antes con x=-99
0 con i 0 modifica x a 0
0 con i 1 modifica x a 0
1 antes con x=-99
1 con i 2 modifica x a 2
1 con i 3 modifica x a 3
3 antes con x=-99
3 con i 6 modifica x a 18
3 con i 7 modifica x a 21
2 antes con x=-99
2 con i 4 modifica x a 8
2 con i 5 modifica x a 10
0 after for, x vale 21
3 after for, x vale 21
1 after for, x vale 21
2 after for, x vale 21
La x al final vale 21
```

MÁS CONSTRUCTORES /CLÁUSULAS ÚTILES

#pragma omp declare reduction(….)

Permite definir nuevos operadores de reducción para ser usados en la cláusula *reduction*.

```
int ParIdamin(const int N, const double *v) {
   int i, pos_min = 0;    double val_min = v[0];
   #pragma omp parallel
   {
      double val = DBL_MAX;    int pos = 1;
      #pragma omp for nowait
        for(i=1; i<N; i++)  if (v[i]<val) { val=v[i]; pos=i; }
      #pragma omp critical
        if (val < val_min) { val_min=val; pos_min=pos; }
   }
   return pos_min;
}
int main(int argc, char * argv[]) {
    double v[10]={9, 6, 7, 5, 3, 3, 3, 3 ,3, 3};
    printf("%d\n",ParIdamin(10, v)); …
```

¿Qué hace?

¿Y si quiero el primer/último?

MÁS CONSTRUCTORES /CLÁUSULAS ÚTILES

#pragma omp declare reduction(....) “para la posición del primer mínimo”

Permite definir nuevos operadores de reducción para ser usados en la cláusula *reduction*.

```
int ParIdamin(const int N, const double *v) {
   int i, pos_min = 0;    double val_min = v[0];
   #pragma omp parallel
   {
      double val = DBL_MAX;    int pos = 1;
      #pragma omp for nowait
        for(i=1; i<N; i++)  if (v[i]<val) { val=v[i]; pos=i; }
      #pragma omp critical
        if (val < val_min) { val_min=val; pos_min=pos; }
         else if ((val==val_min) && (pos < pos_min))
pos_min=pos;
   }
   return pos_min;
}
...
```

MÁS CONSTRUCTORES /CLÁUSULAS ÚTILES

#pragma omp declare reduction(….) “para la posición del primer mínimo”

¿Más Rápida?
No sé
Más portable seguro

```
int ParIdamin(const int N, const double *v)
{
  int i;
  PosMin data ={DBL_MAX, 0};

  #pragma omp parallel for reduction(MIN: data)
  for (i=0; i<N; i++)
     if (v[i] < data.val) { data.val=v[i];
data.pos=i; }

    return data.pos;
}
```

```
struct PosMin {
  float val;
  int    pos;
};
typedef struct PosMin PosMin;

#pragma omp declare reduction(MIN : PosMin : \
  omp_out = ((omp_in.val < omp_out.val) || ((omp_in.val == omp_out.val)  && \\
       (omp_in.pos < omp_out.pos))) ? omp_in : omp_out) \
       initializer(omp_priv = { FLT_MAX, 0 })
```

MÁS CONSTRUCTORES /CLÁUSULAS ÚTILES

```
#pragma omp ordered
    structured block
```

Los hilos ejecutan el **structured block** siguiendo el orden natural de las iteraciones del bucle en secuencial.

```
#pragma omp ordered [clause, [clause]]
    structured block
```

Igual que el anterior si no se ponen cláusulas o se usa la cláusula *threads*

Si se usa la cláusula *simd* el orden natural es a nivel *simd* o *simd loop*.

Si se usan las cláusulas *depend(source)* o *depend(sink: vec)* la ordenación se hace en base a la ordenación establecida en *source* o *sink* (ver manual para más información).

EJEMPLOS

```
#include <omp.h>
int main () {
  int yo, i;
  #pragma omp parallel private(yo, i) num_threads(4)
  {
    yo = omp_get_thread_num();
    #pragma omp for
      for (i=0;i<4;i++)
        printf("%d en ordered con i %d\n", yo, i);
  }
  return 0;
}
```

```
./111_1
0 en ordered con i 1
1 en ordered con i 0
3 en ordered con i 2
2 en ordered con i 3
```

Necesario

```
...
yo = omp_get_thread_num();
#pragma omp for ordered
  for (i=0;i<4;i++)
  #pragma omp ordered
    printf("%d en ordered con i %d\n", yo, i);
...
```

```
./111_2
0 en ordered con i 0
1 en ordered con i 1
2 en ordered con i 2
3 en ordered con i 3
```

EJEMPLOS

```
#include <omp.h>
int main () {
  int yo;
  #pragma omp parallel private(yo)
  {
    yo = omp_get_thread_num();
    #pragma omp single
      {printf("%d en solitario dentro del 1st single\n",yo); sleep(1);}

    #pragma omp master
      {printf("%d. Debo ser 0, estoy en master\n",yo);  sleep(1);}

    #pragma omp single
      {printf("%d en solitario dentro del 2nd single\n",yo); sleep(1);}

    #pragma omp single
      {printf("%d en solitario dentro del 3rd single\n",yo); sleep(1);}

    printf(“Thread %d, fuera de singles y masters\n",yo);
  }
  return 0;
}
```

```
export OMP_NUM_THREADS=8
./112
    0 en solitario dentro del 1st single
    2 en solitario dentro del 2nd single
    0. Debo ser 0, estoy en master
    0 en solitario dentro del 3rd single
    Thread 0 fuera de singles y masters
    Thread 5 fuera de singles y masters
    Thread 4 fuera de singles y masters
    Thread 3 fuera de singles y masters
    Thread 2 fuera de singles y masters
    Thread 6 fuera de singles y masters
    Thread 7 fuera de singles y masters
    Thread 1 fuera de singles y masters
```

EJEMPLOS

```
#include <omp.h>
int main () {
  int yo;
  #pragma omp parallel private(yo)
  {
    yo = omp_get_thread_num();
    #pragma omp single nowait
      {printf("%d en solitario dentro del 1st single\n",yo); sleep(1);}

    #pragma omp master
      {printf("%d. Debo ser 0, estoy en master\n",yo);  sleep(1);}

    #pragma omp single nowait
      {printf("%d en solitario dentro del 2nd single\n",yo); sleep(1);}

    #pragma omp single nowait
      {printf("%d en solitario dentro del 3rd single\n",yo); sleep(1);}

    printf(“Thread %d, fuera de singles y masters\n",yo);
  }
  return 0;
}
```

```
export OMP_NUM_THREADS=8
./113
     0 en solitario dentro del 1st single
     0. Debo ser 0, estoy en master
     1 en solitario dentro del 2nd single
     2 en solitario dentro del 3rd single
     Thread 3 fuera de singles y masters
     Thread 4 fuera de singles y masters
     Thread 5 fuera de singles y masters
     Thread 6 fuera de singles y masters
     Thread 7 fuera de singles y masters
     Thread 0 fuera de singles y masters
     Thread 1 fuera de singles y masters
     Thread 2 fuera de singles y masters
```

EJEMPLOS

```
#include <omp.h>
float work1(int i)
    return 1.0 * i;

float work2(int i)
    return 2.0 * i;

void atomic_example(float *x, float *y, int * index, int n)
{
    int i;
    #pragma omp parallel for shared(x, y, index, n)
    for (i=0; i<n; i++) {
        #pragma omp atomic update
            x[index[i]] += work1(i);
        y[i] += work2(i);                    /* NO LE AFECTA EL ATOMIC */
    }
}
```