

SIMD con hebras. OpenMP

August 29, 2022

¿Qué es OpenMP?

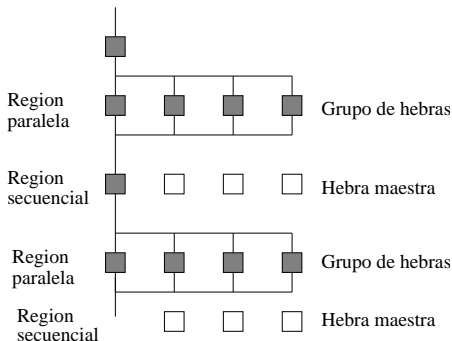
OpenMP es un estándar para programación paralela en sistema de memoria compartida

- Las aplicaciones desarrolladas con OpenMP son portables a todas las arquitecturas de memoria compartida que cuenten con un compilador *capaz de* OpenMP
- Permite **paralelización incremental**
- OpenMP extiende funcionalidades de lenguajes tradicionales a través de:
 - Nuevas **directivas** para el lenguaje
 - Funciones de librería
- Actualmente disponible para C/C++ y Fortran en muchos compiladores comerciales
- OpenMP no es un lenguaje!

THE

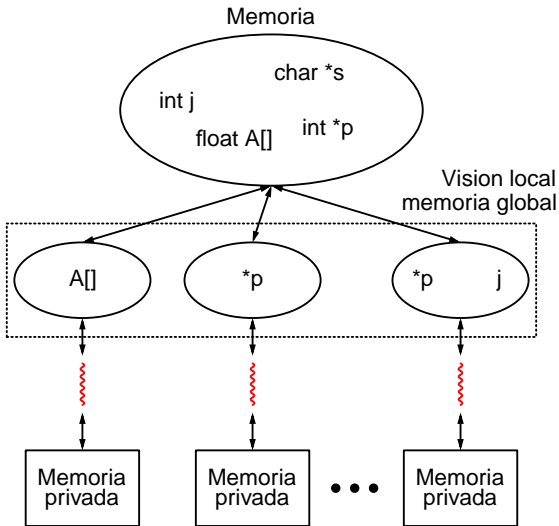
Modelo OpenMP de ejecución

- La ejecución comienza con la hebra inicial
- Comienza una construcción paralela
- La hebra (maestra) crea a las hebras hijas
- Término de la construcción paralela
 - Las hebras se sincronizan
 - Existencia implícita de una barrera
- La hebra maestra es la única que continua
- Comienza otra construcción paralela, etc



- Todas las hebras tienen acceso a un lugar donde almacenar y recuperar variables compartidas; este lugar es llamado **memoria**
- Cada hebra puede tener un **visión temporal propia de la memoria**
- Además, cada hebra puede tener su propia memoria no accesible a otras hebras; **memoria privada de hebra** o **threadprivate memory**
- Si varias hebras escriben una variable compartida sin sincronización, el valor final de la variable es no especificado
- Si una hebra lee una variable compartida y otra hebra la escribe sin sincronización, el valor leído por cualquier hebra es no especificado.

Journal of Management Education 36(7) 809–827



- Note que la operación **flush** NO garantiza que una hebra lea el valor actualizado de una variable
- El programador debe garantizar el siguiente orden:
 - La primera hebra escribe el valor de una variable en su visión temporal de la memoria
 - La primera hebra realiza un **flush**
 - La segunda hebra realiza un **flush**
 - La segunda hebra lee el valor de la variable

© 2011 Pearson Education, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or by any information storage or retrieval system, without prior written permission from the publisher.

- `#pragma omp parallel [clause[,] clause]...`
 bloque estructurado
- Algunas de las posibles cláusulas son
 - `private (list)+`
 - `shared (list)+`
- El orden en que aparecen las cláusulas no es importante

Algunas de las posibles cláusulas son

- **private** (list)+
- **shared** (list)+

- ```
#pragma omp parallel barrier
```



## Ejemplo

```

1 int val1;

2 int function1(int tid) {
3 float f;

4 f = tid*val1;
5 ...
6 return(f);
7 }

8 void main() {
9 int ntasks = 3;
10 int val3 = 2.0;
11 #pragma omp parallel num_threads(ntasks)
12 {
13 val3 = function1(omp_get_thread_num());
14 }

15 }

```

# Variables de medio ambiente

Variables que pueden ser “seteadas” antes de la ejecución del programa

- **OMP\_NUM\_THREADS**

- Define el número de hebras durante la ejecución del programa
- `setenv OMP_NUM_THREADS 16 (csh, tcsh)`
- `export OMP_NUM_THREADS=16 (sh, bash, ksh)`
- Cuando se ha especificado que el número de hebras se ajusta dinámicamente, esta variable indica el número máximo de hebras que se puede usar

- **OMP\_SCHEDULE**

- Define la política (tipo de planificación) para dividir un for-loop entre varias hebras.
- Opcionalmente, define el tamaño de la porción del loop
- `setenv OMP_SCHEDULE 'STATIC,4' (csh, tcsh)`
- `export OMP_SCHEDULE='STATIC,4' (sh, bash, ksh)`





---



---

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define f(A) (sqrt(1.0-(A*A)))
#define N 1000000000
```

```
main() {
 double sum=0.0, x, pi;
 int i;
 for (i=1;i<=N;i++) {
 x = drand48();
 sum = sum + f(x);
 }
 pi = 4.0*sum/N;
 printf("pi = %f\n", pi);
}
```

---



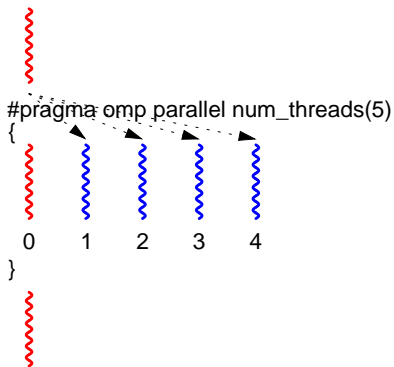
# Directiva parallel

- La estructura sintáctica fundamental para crear bloques paralelos es

```
pragma omp parallel [clause[,] clause]...
 bloque-estructurado
```
- donde las cláusulas pueden ser:
  - `if`(*expresion-escalar*)
  - `private`(*lista-variables*)
  - `firstprivate`(*lista-variables*)
  - `shared`(*lista-variables*)
  - `default`(*none*)
  - `copyin`(*lista-variables*)
  - `reduction`(*operador: lista-variables*)
  - `num_threads`(*expresion-entera*)

## Directiva parallel

- Cuando un hebra encuentra una construcción `parallel`, crea un equipo de hebras que ejecuta el bloque en forma paralela redundante
- La hebra que crea al equipo de hebras pasa a ser la hebra maestra del equipo, con número de hebra 0, en dicho equipo, mientras dure la región paralela
- Una hebra puede conocer su ID invocando a `omp_get_thread_num()`





Directiva **parallel**: número de hebras

- Para determinar el número de hebras requeridas para el bloque, se evalúan las siguientes reglas, en orden
  - 1 Si la cláusula **if** está presente y se evalúa a cero (falso), el bloque es serializado
  - 2 Si se ha alcanzado el nivel máximo de paralelismo, el bloque es serializado
  - 3 Si la cláusula **num\_threads** está presente, entonces el valor de la expresión de dicha cláusula es el número de hebras requeridas
  - 4 Si la función de librería **omp\_set\_num\_threads()** ha sido invocada, entonces ese es el número de hebras
  - 5 Si la variable de medioambiente **OMP\_NUM\_THREADS** ha sido definida, entonces el valor del argumento de dicha variable es el número de hebras requerida
  - 6 Si nada de lo anterior es verdad, queda definida por la implementación, es decir el compilador

## Ejemplo elección número de hebras

- El siguiente caso crea 5 hebras para el bloque

```
....
#pragma omp parallel num_threads(5)
```

- El siguiente caso genera 3 hebras

```
....
omp_set_num_threads(3);
#pragma omp parallel
```

- El siguiente caso?

```
....
#pragma omp parallel if(val) num_threads(12)
```

- ¿Qué pasa con el siguiente código?

```
....
omp_set_num_threads(3);
#pragma omp parallel num_threads(5)
```

- ¿Y con el siguiente?

```
....
#pragma omp parallel
omp_set_num_threads(3)
```

\_\_\_\_\_

# Paralelismo anidado

- Paralelismo anidado puede ser activado con:

- `OMP_NESTED=TRUE`
- `omp_set_nested()`

```
print_num_threads(int level) {
 printf("Nivel %d: %d threads, Soy %d\n", level,
 omp_get_num_threads(), omp_get_thread_num());
}
main() {
 omp_set_nested(1);
 #pragma omp parallel num_threads(2)
 {
 print_num_threads(1);
 #pragma omp parallel num_threads(3)
 print_num_threads(2);
 }
}
```

```
Nivel 1: 2 threads, Soy 0
Nivel 1: 2 threads, Soy 1
Nivel 2: 3 threads, Soy 1
Nivel 2: 3 threads, Soy 0
Nivel 2: 3 threads, Soy 1
Nivel 2: 3 threads, Soy 2
Nivel 2: 3 threads, Soy 2
Nivel 2: 3 threads, Soy 0
```

# Paralelismo anidado desactivado

```

print_num_threads(int level) {
 printf("Nivel %d: %d threads, Soy %d\n", level,
 omp_get_num_threads(), omp_get_thread_num());
}
main() {
 omp_set_nested(0);
 #pragma omp parallel num_threads(2)
 {
 print_num_threads(1);
 #pragma omp parallel num_threads(3)
 print_num_threads(2);
 }
}

```

```

Nivel 1: 2 threads, Soy 0
Nivel 2: 1 threads, Soy 0
Nivel 1: 2 threads, Soy 1
Nivel 2: 1 threads, Soy 0

```

- Paralelismo anidado está desactivado por defecto en la mayoría de los compiladores
- La creación de un número de hebras mucho más grande que el número de unidades de procesamiento puede implicar deterioro en rendimiento

## Ejemplo con parallel

En este ejemplo, el procesamiento de un arreglo es paralelizado en base al número de hebras y al largo del arreglo

```
#define N 100

main() {
 int imagen[N];
 int i, n, numth, mytid;

 #pragma omp parallel private(mytid, numth, n, i)
 {
 int j;
 mytid = omp_get_thread_num();
 numth = omp_get_num_threads();
 n = N/numth;
 i = mytid * n;
 for (j=i; j < i+n; j++) //Se divide a mano el trabajo
 imagen[j] = 2*imagen[j];
 }
}
```

openMP también provee directivas para repartir el trabajo en forma automática

## Atributos de compartición en `parallel`

El atributo de compartición de una variable referenciada dentro de una construcción se determina por una de las siguientes modalidades:

- 1 **Predeterminada:** Definidas antes de comenzar la construcción
- 2 **Explícitamente determinada:** Definidas por cláusulas de la construcción
- 3 **Implícitamente determinada:**

1. *Journal of the American Medical Association*, 2000; 283: 2689-2696.

- 1 Variables que aparecen en `threadprivate` son `threadprivate`
- 2 Variables automáticas declaradas en un scope dentro de la construcción son `private`
- 3 Objetos con almacenamiento dinámico son `shared`
- 4 La variable en un `for` o `parallel for` son `private`
- 5 Variables con almacenamiento estático (`static`) declaradas en un scope dentro de la construcción son `shared`



# Ejemplo compartición predeterminada

```
int main (void)
{
 int i;
 char *p = malloc(10);
 strcpy(p,"hola");
 printf("%s\n", p);
 #pragma omp parallel num_threads(5)
 {
 int tid = omp_get_thread_num();
 static int j = -1;
 if (tid == 0) {
 strcpy(p, "chao");
 j = 5;
 printf("j = %d\n", j);
 }
 for (i=0; i < 2; i++)
 printf("tid = %d, i = %d, %s, j= %d\n", tid, i, p, j);
 }
}
```

```
hola
tid = 4, i = 0, chao, j= 5
tid = 4, i = 1, chao, j= 5
tid = 1, i = 0, chao, j= -1
tid = 3, i = 0, chao, j= -1
tid = 2, i = 0, chao, j= -1
j = 5
tid = 0, i = 0, chao, j= 5
tid = 0, i = 1, chao, j= 5
```

La variable the for-loop "i" es compartida!



# Atributo explícito

```

int main (void)
{
 int i;
 char *p = malloc(10);
 strcpy(p,"hola");
 printf("%s\n", p);
 #pragma omp parallel num_threads(5) private(i)
 {
 int tid = omp_get_thread_num();
 static int j = -1;
 if (tid == 0) {
 strcpy(p, "chao");
 j = 5;
 printf("j = %d\n", j);
 }
 for (i=0; i < 2; i++)
 printf("tid = %d, i = %d, %s, j= %d\n", tid, i, p, j);
 }
}

```

tid = 2, i = 0, chao, j= -1  
 tid = 2, i = 1, chao, j= 5  
 tid = 1, i = 0, chao, j= -1  
 tid = 1, i = 1, chao, j= 5  
 tid = 4, i = 0, chao, j= -1  
 tid = 4, i = 1, chao, j= 5  
 j = 5  
 tid = 0, i = 0, chao, j= 5  
 tid = 0, i = 1, chao, j= 5  
 tid = 3, i = 0, chao, j= -1  
 tid = 3, i = 1, chao, j= 5

## Atributo de compartición implícito

Una variable referenciada dentro del alcance de una construcción, posee un atributo de compartición implícitamente determinada, es aquella que no aparece en la lista de cláusulas de compartición, no tienen atributos predeterminados.

- 1 En una región **parallel** o **task** el atributo está determinado por la cláusula **default**, si ésta está presente
- 2 Si no hay cláusula **default** en **parallel** la variable es **shared**
- 3 Es decir, cuando no aparece la cláusula **default** es equivalente a **default(shared)**
- 4 Para construcciones excepto **task**, la variable hereda el atributo que tiene en el scope mayor
- 5 En una construcción **task**, si no hay **default**, la variable es compartida y hereda esta propiedad en todos los scope menores incluidos en la construcción

- Todas las variables con atributos implícitos son **shared**

- Especifica que la referencia a cualquier variable no asume ningún defecto. Luego, si existe una referencia a una variable, debe cumplirse que

- 1 La variable está explícitamente listada en una cláusula de compartición
- 2 La variable está declarada en la construcción paralela
- 3 La variable es un objeto **const**
- 4 La variable es la variable que controla el loop
- 5 La variable es **threadprivate**

# private versus firstprivate

**private** establece cuáles variables son privadas a las hebras, y no especifica cómo éstas deben ser inicializadas.

**firstprivate** dice cuáles variables son privadas y especifica que deben ser inicializadas con el valor de la variable al comenzar el bloque paralelo

```
int main (void)
{
 int i = 10;
 omp_set_num_threads(3);
 #pragma omp parallel private(i)
 {
 printf("tid %d, i = %d\n",
 omp_get_thread_num(), i);
 i = omp_get_thread_num();
 }
 printf("i = %d\n", i);
}

tid 1, i = 32603
tid 2, i = 0
tid 0, i = 0
i = 10
```

```
int main (void)
{
 int i = 10;
 omp_set_num_threads(3);
 #pragma omp parallel firstprivate(i)
 {
 printf("tid %d, i = %d\n",
 omp_get_thread_num(), i);
 i = omp_get_thread_num();
 }
 printf("i = %d\n", i);
}

tid 2, i = 10
tid 1, i = 10
tid 0, i = 10
i = 10
```

## shared y el modelo de memoria OpenMP

Por defecto, todas las variables declaradas en el scope en que se encuentra la directiva **parallel** son globales en el bloque paralelo

El modelo de memoria de openMP requiere de mucho cuidado cuando se trabaja con variables **shared**

## shared y el modelo de memoria OpenMP

```
main() {
 int x = 2;
 #pragma omp parallel num_threads(10) shared(x)
 {
 if (omp_get_thread_num() == 0)
 x = 5;
 else
 printf("1: tid %d: x = %d\n",
 omp_get_thread_num(), x);
 #pragma omp barrier
 if (omp_get_thread_num() == 0)
 printf("2: tid %d: x = %d\n",
 omp_get_thread_num(), x);
 else
 printf("3: tid %d: x = %d\n", omp_get_thr
 }
}
```

```
1: tid 3: x = 2
1: tid 4: x = 2
1: tid 1: x = 2
1: tid 2: x = 5
1: tid 5: x = 5
1: tid 7: x = 5
1: tid 6: x = 5
1: tid 8: x = 5
1: tid 9: x = 5
2: tid 0: x = 5
3: tid 3: x = 5
3: tid 7: x = 5
3: tid 9: x = 5
3: tid 6: x = 5
3: tid 5: x = 5
3: tid 8: x = 5
3: tid 2: x = 5
3: tid 1: x = 5
3: tid 4: x = 5
```

La barrera sincroniza las visiones locales de cada hebra con la memoria global.



Estas construcciones deben existir dentro de **parallel**

# Construcción **for**

```
#pragma omp for [clause[[,] clause]..]]
 for-loop
```

- Las iteraciones del `for-loop` serán ejecutadas en paralelo
- Las iteraciones son distribuidas entre los miembros del equipo de una región paralela especificadas anteriormente con la construcción **parallel**
- Las cláusulas posibles son:
  - **private**(*lista-variables*)
  - **firstprivate**(*lista-variables*)
  - **lastprivate**(*lista-variables*)
  - **reduction**(*operador: lista-variables*)
  - **ordered**
  - **schedule**(*kind, size*)
  - **nowait**

## Ejemplo con `for`

En este ejemplo, el procesamiento de un arreglo es paralelizado en forma automática

```
#define N 100
main() {
 int imagen[N];
 int j;
 ...
 #pragma omp parallel private(j)
 {
 #pragma omp for
 for (j=0; j < N; j++) //Se divide automaticament el trabajo
 imagen[j] = 2*imagen[j];
 }
}
```

¿Cómo saber que índices (posiciones del arreglo) le toca a cada hebra?

# Construcción **for** y **schedule**

**schedule(kind, size)** especifica cómo serán repartidas las iteraciones entre las hebras

- El correctitud del programa no debiera ser afectado por esta construcción
- El tipo de schedule, **kind**, puede ser:
  - ① **static**: las iteraciones se dividen en porciones de tamaño **size** y asignadas estáticamente en round-robin a las hebras. Si no se especifica **size**, la división se hace en porciones aproximadamente iguales en tamaño
  - ② **dynamic**: se asigna a cada hebra una porción **size** en forma dinámica. Cuando una hebra termina su porción, solicita una nueva porción de tamaño **size**
  - ③ **guided**: Opera como **dynamic**, pero las porciones son cada vez más pequeñas. Se reducen aproximadamente en forma exponencial hasta llegar a **size**
  - ④ **runtime**: La decisión se posterga hasta tiempo de ejecución, la cual puede estar definida por la variable de medio ambiente **OMP\_SCHEDULE**

```
int i; tid 0, i = 0
#pragma omp parallel num_threads(2) tid 0, i = 1
{ tid 0, i = 4
 #pragma omp for schedule(static, 2) tid 0, i = 5
 for (i=0; i < 9; i++) tid 1, i = 2
 printf("tid %d, i = %d\n", omp_get_thread_num(), i); tid 1, i = 3
} tid 1, i = 6
 tid 1, i = 7
 tid 0, i = 8
```

**static** implica que las iteraciones del loop pueden ser (y a lo mejor son) asignadas a las hebras antes que ellas comiencen a ejecutar el cuerpo del loop.

```
int i;
#pragma omp parallel num_threads(2)
{
 #pragma omp for schedule(static, 2)
 for (i=0; i < 9; i++) {
 printf("tid %d, i = %d\n", omp_get_thread_num(), i);
 if (omp_get_thread_num() == 0) sleep(5);
 }
}
```

# schedule(dynamic,) al rescate!

```
#pragma omp parallel num_threads(2)
{
 #pragma omp for schedule(dynamic, 2)
 for (i=0; i < 9; i++)
 printf("%d, i = %d\n",
 omp_get_thread_num(), i);
}

#pragma omp parallel num_threads(2)
{
 #pragma omp for schedule(dynamic, 2)
 for (i=0; i < 9; i++) {
 printf("%d, i = %d\n",
 omp_get_thread_num(), i);
 if (omp_get_thread_num() == 0) sleep(5);
 }
}
```

## Dos ejecuciones distintas

|          |          |
|----------|----------|
| 0, i = 0 | 1, i = 2 |
| 0, i = 1 | 1, i = 3 |
| 0, i = 4 | 1, i = 4 |
| 0, i = 5 | 1, i = 5 |
| 0, i = 6 | 1, i = 6 |
| 0, i = 7 | 1, i = 7 |
| 0, i = 8 | 1, i = 8 |
| 1, i = 2 | 0, i = 0 |
| 1, i = 3 | 0, i = 1 |
|          |          |
| 1, i = 2 |          |
| 1, i = 3 |          |
| 1, i = 4 |          |
| 1, i = 5 |          |
| 1, i = 6 |          |
| 1, i = 7 |          |
| 1, i = 8 |          |
| 0, i = 0 |          |
| 0, i = 1 |          |

# Barrera implícita y `nowait`

Al término de un bloque paralelo existe una **barrera implícita**, es decir las hebras deben esperar a que todas terminen el bloque para continuar

`nowait` elimina la barrera implícita.

```
#pragma omp parallel num_threads(2)
{
 #pragma omp for schedule(dynamic, 2) nowait
 for (i=0; i < 9; i++) {
 printf("%d, i = %d\n", omp_get_thread_num(), i);
 if (omp_get_thread_num() == 0)
 sleep(5);
 }
 printf("%d sali\n", omp_get_thread_num());
}

1, i = 2
0, i = 0
1, i = 3
1, i = 4
1, i = 5
1, i = 6
1, i = 7
1, i = 8
1 sali
0, i = 1
0 sali
```

```

 }
 #pragma omp parallel
 {
 #pragma omp for nowait
 for (i=0; i<N; i++) {
 // Parallel execution of the loop body
 }
 }
}
```

Independientemente de la asignación de los índices del **for**, la directiva **ordered** dentro de un **for** con cláusula **ordered**, secuencializa la ejecución del bloque

```
0, i = 0
0, i = 1
1, i = 2
1, i = 3
0, i = 4
0, i = 5
1, i = 6
1, i = 7
0, i = 8
```



## reduction en un for

- Las variables en una cláusula **reduction** deben ser **shared**
- Las variables declaradas como **private** no pueden aparecer en la lista de **reduction**
- Sin embargo, durante la ejecución de la región paralela, se crea una copia privada de las variables, como si hubieran sido declaradas **private**
- El operador de reducción puede ser uno de los siguientes:  
+, \*, -, &, ^, |, &&, ||

```
int sum = 0; //shared variable
#pragma omp parallel num_threads(2)
{
 int tid = omp_get_thread_num();
 #pragma omp for reduction(+:sum)
 for (i=0; i < 7; i++)
 sum += i*i;
 printf("%d con sum = %d\n", tid, sum);
}
printf("total sum = %d\n", sum);
```

```
0 con sum = 91
1 con sum = 91
total sum = 91
```

# reduction en parallel

La cláusula **reduction** puede aparacer en la directiva **parallel** antes de un **for** y tener (casi) el mismo efecto

```
int sum = 0; //shared variable
#pragma omp parallel num_threads(2) reduction(+:sum)
{
 int tid = omp_get_thread_num();
 #pragma omp for
 for (i=0; i < 7; i++)
 sum += i*i;
 printf("%d con sum = %d\n", tid, sum);
}
printf("total sum = %d\n", sum);
```

```
0 con sum = 14
1 con sum = 77
total sum = 91
```

Se recomienda entender el concepto de alcance de una cláusula para evitar introducir errores



## Construcción `sections`, con `section`

- Usada para paralelizar un conjunto de bloques entre las hebras de la región paralela (es decir, debe aparecer en una region **parallel**)

- Cada bloque es ejecutado por sólo una hebra

```
#pragma omp sections [clause[,] clause]...
{
 #pragma omp section
 bloque estructurado
 #pragma omp section
 bloque estructurado
}
```

- Las cláusulas son

- **private**(lista-variables)
- **firstprivate**(lista-variables)
- **lastprivate**(lista-variables)
- **reduction**(operador: lista-variables)
- **nowait**

```
#pragma omp parallel
{
 #pragma omp sections
 {
 #pragma omp section
 #pragma omp section
 #pragma omp section
 }
}
```

## Ejemplo con **sections**

```

int sum = 0;
#pragma omp parallel num_threads(3) reduction(+:sum)
{
 #pragma omp sections
 {
 #pragma omp section
 {
 sum = 1;
 printf("%d, en section 0\n", omp_get_thread_num());
 }
 #pragma omp section
 {
 sum = 2;
 printf("%d, en section 1\n", omp_get_thread_num());
 }
 #pragma omp section
 {
 sum = 3;
 printf("%d, en section 2\n", omp_get_thread_num());
 }
 }
}
printf("total sum = %d\n", sum);

```

0, en section 1  
2, en section 2  
1, en section 0  
total sum = 6

## Ejemplo con sections

Si el número de hebras participantes es mayor que el número de **sections**, simplemente algunas no hacen nada.

```
int sum = 0;
#pragma omp parallel num_threads(8) reduction(+:sum)
{
 #pragma omp sections
 {
 #pragma omp section
 {
 sum = 1;
 printf("%d, en section 0\n", omp_get_thread_num());
 }
 #pragma omp section
 {
 sum = 2;
 printf("%d, en section 1\n", omp_get_thread_num());
 }
 }
}

printf("total sum = %d\n", sum);
```

## Condición de carrera en **sections**

Si el número de hebras es menor que el número de **section**, algunas hebras realizarán más de un **section** produciendo resultados erróneos

```
int sum = 0;
#pragma omp parallel num_threads(2) reduction(+:sum)
{
 #pragma omp sections
 {
 #pragma omp section
 {
 sum = 1;
 printf("%d, en section 0\n", omp_get_thread_num());
 }
 #pragma omp section
 {
 sum = 2;
 printf("%d, en section 1\n", omp_get_thread_num());
 }
 #pragma omp section
 {
 sum = 3;
 printf("%d, en section 2\n", omp_get_thread_num());
 }
 }
}
printf("total sum = %d\n", sum);
```

0, en section 0  
0, en section 2  
1, en section 1  
total sum = 5  
...  
1, en section 1  
1, en section 2  
0, en section 0  
total sum = 4

# Construcción **single**

```
#pragma omp single [clause[[,] clause]...]
 bloque estructurado
```

- Especifica que el bloque asociado se ejecuta por sólo una hebra del equipo
- No necesariamente la hebra maestra
- Las cláusulas son
  - **private**(*lista-variables*)
  - **firstprivate**(*lista-variables*)
  - **copyprivate**(*lista-variables*)
  - **nowait**
- Hay una barrera implícita al final de la construcción, a menos que se use la cláusula **nowait**



## Ejemplo single

**single** es útil también para escribir mensajes de depuración

```
void work1() { ... }
void work2() { ... }
main() {
 #pragma omp parallel num_threads(5)
 {
 #pragma omp single
 printf("Invocando work1()\n");
 work1();
 #pragma omp single
 printf("Fin work1()\n");
 #pragma omp single
 printf("Invocando work2()\n");
 work2();
 #pragma omp single
 printf("Fin work2()\n");
 }
}
```



## copyprivate y single

Los valores de las variables listadas en la cláusula **copyprivate** de un **single** son copiadas a las variables correspondientes de las otras hebras. Esto actúa como un **broadcast**

```

void getValue(int *s) { scanf("%d\n", s); } 6745
void useValue(int s) { printf("%d\n", s); } 6745

int main (void) 6745
{ 6745
 int x = 10;
 omp_set_num_threads(3);
 #pragma omp parallel firstprivate(x)
 {
 #pragma omp single copyprivate(x)
 {
 getValue(&x);
 useValue(x);
 }
 }
}

```

Las variables en **copyprivate** deben tener un operador asignación no ambiguo.

```
#pragma omp threadprivate(var1, var2,...)
```

**threadprivate** crea una copia privada de cada variable listada en la directiva, a cada hebra, que bajo ciertas circunstancias es global o estática a la hebra

- 1 `threadprivate` es una directiva , no una cláusula
- 2 Las variables serán privadas a sus hebras, pero globales en cada una de ellas.
- 3 Para que una variable `threadprivate` persista sobre varias regiones paralelas, debe usarse `schedule(static, *)`, y el mismo número de hebras en cada región
- 4 El alcance de `threadprivate` es global al archivo

# Ejemplo threadprivate

```
int x;
#pragma omp threadprivate(x)
main() {
 int tid;
 omp_set_num_threads(3);

 #pragma omp parallel private(tid)
 {
 tid = omp_get_thread_num();
 x = 10*tid + 1;
 printf("Dentro de parallel 1. Hebra = %d con x = %d\n", tid, x);
 }

 printf("Fuera de parallel. Hebra = %d con x = %d\n", tid, x);
 #pragma omp parallel private(tid)
 {
 tid = omp_get_thread_num();
 printf("Dentro de parallel 2. Hebra = %d con x = %d\n", tid, x);
 }
}
```

```
Dentro de parallel 1. Hebra = 0 con x = 1
Dentro de parallel 1. Hebra = 2 con x = 21
Dentro de parallel 1. Hebra = 1 con x = 11
Fuera de parallel. Hebra = 0 con x = 1
Dentro de parallel 2. Hebra = 1 con x = 11
Dentro de parallel 2. Hebra = 2 con x = 21
Dentro de parallel 2. Hebra = 0 con x = 1
```

# Uso incorrecto de **threadprivate**

```
int a[1000];
#pragma omp threadprivate(a)

int main(int argc, char **argv)
{
 int i;
 int sum = 0;

 #pragma omp parallel for schedule(static, 4)
 for (i = 0; i < 1000; i++) {
 a[i] = i;
 }

 #pragma omp parallel for reduction (+:sum) schedule(static, 8)
 for (i = 0; i < 1000; i++) {
 sum = sum + a[i];
 }

 printf("%d\n", sum);
}
```

126858

---

# Construcciones combinadas

- Estas construcciones son “abreviaciones” para una región paralela que contiene sólo una construcción que divide el trabajo:

- parallel for**

```
#pragma omp parallel for [clause[,] clause]...
 for-loop
```

- parallel sections**

```
#pragma omp parallel sections [clause[,] clause]...
{
 #pragma omp section
 bloque estructurado
 #pragma omp section
 bloque estructurado
 ...etc
}
```



Especifica que el bloque estructurado asociado es ejecutado sólo por la hebra maestra del equipo. No existe barrera implícita al comienzo ni al final del bloque estructurado

Restringe la ejecución del bloque estructurado asociado, a una hebra a la vez. Es decir, funciona parecido a una sección crítica

# Construcciones para sincronización

- **barrier**  
Sincroniza las hebras de un equipo. Cuando una hebra llega a esta construcción, espera a que todas las otras hebras del equipo lleguen.  
`#pragma omp barrier`

Asegura que una localización específica de memoria es actualizada atómicamente

```
#pragma omp atomic
 expression
```

**expression** debe ser alguna de las siguientes:

- $x \text{ binop} = \text{expr}$ , donde *binop* es un operador binario  
+, \*, -, /, &, ^, |, <<, >>
- $x++$ ,  $++x$ ,  $x--$ ,  $--x$

La implementación podría reemplazar `atomic` por `critical`. Sin embargo, `atomic` podría explotar funciones de hardware.

Especifica un punto de sincronización en el cual se asegura que todas las hebras tengan una visión consistente de ciertos objetos (variables) en memoria

Esto significa que todas las evaluaciones anteriores a este punto de sincronización que actualicen alguna de las variables especificadas deben completarse, y que ninguna de las expresiones subsiguientes deben ejecutarse hasta que todas las variables de la lista estén en consistencia.

Esto puede implicar que los compiladores deban restaurar los valores de objetos en registros a memoria; que variables en buffers de hardware sean escritos a memoria, etc

---

- barrier

- Al comienzo y término de **critical**
- Al comienzo y término de **ordered**
- Al término de **parallel**
- Al término de **for**
- Al término de **sections**
- Al término de **single**

a menos que existe una directiva `nowait`

- **ordered**  
Especifica que el bloque estructurado se ejecuta en el orden en que se ejecutaría en un loop secuencial

```
#pragma omp ordered
 bloque-estructurado
```

## Construcciones para sincronización: locks

- Un lock es una variable que solo puede ser poseído por a lo más una hebra.
- Un lock puede estar en uno de los siguientes estados
  - ➊ No inicializado
  - ➋ Unlocked (desbloqueado, no poseído, abierto)
  - ➌ Locked (bloqueado, poseído, cerrado, tomado)
- Un lock puede ser poseído por solo una hebra (o tarea)

- `omp_init_lock(omp_lock_t *)`: inicializa el lock
- `omp_set_lock(omp_lock_t *)`: toma el lock si está libre, y lo cierra. Si no está libre, la hebra se bloquea.
- `omp_unset_lock(omp_lock_t *)`: libera el lock
- Un lock simple solo puede ser bloqueado una vez
- Generalmente se usa para proveer exclusión mutua a una sección crítica

Note que es más conveniente, en este caso, usar `critical`



- `omp_nest_lock_t mylock;`
- `omp_init_nest_lock(omp_lock_t *)`: inicializa el lock
- `omp_set_nest_lock(omp_lock_t *)`: toma el lock si está libre o bloqueado por la misma hebra, y lo cierra. Si no está libre, la hebra se bloquea.
- `omp_unset_nest_lock(omp_lock_t *)`: libera (una vez) el lock
- El lock no se libera completamente, hasta que haya sido desbloqueado tantas veces como hay sido bloqueado

- `int omp_get_num_procs(void);`  
Retorna el número máximo de procesadores que pueden ser asignados al proceso
- `int omp_in_parallel(void);`  
Retorna un valor distinto de cero cuando es invocada de una región paralela. Si no retorna cero.
- `void omp_set_dynamic(int dynamic_threads)`  
Si `dynamic_threads != 0`, habilita `schedule` dinámico, con número máximo de `dynamic_threads+1` hebras
- `int omp_get_dynamic(void);`  
Retorna un valor distinto de cero si `schedule` dinámico de hebras está habilitado. Si no, retorna cero.

---



## Tasks en OpenMP ( $\geq 3.0$ )

- 1 Las principales formas de división del trabajo vistas hasta ahora, son apropiadas principalmente a arreglos y matrices, es decir dato estructurado
- 2 **task** permite asignar y dividir trabajo en una forma concurrente no estructurada o irregular
- 3 Ejemplo: Navegación por estructuras de datos como listas o árboles

```
#pragma omp task [clause[,] clause]...
 bloque-estructurado
```

donde las cláusulas pueden ser:

- `if (exp)`
- `untied`
- `shared(list)`
- `private(list)`
- `firstprivate(list)`
- `default(shared | none)`

# Motivación: paralelismo irregular

Supongamos que deseamos paralelizar el procesamiento de los elementos de una lista enlazada:

## Solución sin **task**

```
p = List;
elem = 0;
while (p) {
 list_item[elem++] = p;
 p = next(p);
}
#pragma omp parallel for
for (int i=0; i<elem; i++)
 process(list_item[i]);
```

## Solución con **task**

```
#pragma omp parallel
{
 #pragma omp single private(p)
 {
 p = List;
 while (p) {
 #pragma omp task
 process(p);
 p = next(p);
 }
 }
}
```

---

```
#pragma omp parallel private(p)
{
 #pragma omp for
 for (i=0; i<n_lists; i++) {
 p = listheads[i];
 while (p) {
 #pragma omp task
 process(p)
 p = next(p);
 }
 }
} // barrera implic.
```

```
void traverse(node *p, bool post) {
 if (p->left)
 #pragma omp task
 traverse(p->left, post);
 if (p->right)
 #pragma omp task
 traverse(p->right, post);
 if (post)
 #pragma omp taskwait //barrera explic.
 process(p);
}
```

# ¿Qué es un task?

- 1 Una tarea es:
  - Un trozo de código a ejecutar
  - Un conjunto de datos o medio ambiente
  - Una hebra de ejecución, que ejecuta el código y usa los datos
- 2 Creación y ejecución
  - Cuando una hebra encuentra un **task**, crea la nueva tarea y la planifica
  - Alguna hebra del grupo ejecuta la tarea, posiblemente en un tiempo posterior



task y single

Es posible que sólo una hebra del equipo construya las tareas

```
#pragma omp parallel
{
 #pragma omp single private(p)
 {
 p = List;
 while (p) {
 #pragma omp task
 process(p);
 p = next(p);
 }
 }
}
```

- ❶ La creación de tareas es realizada por sólo una hebra
- ❷ La creación de tareas se ejecuta concurrentemente con la ejecución de las tareas
- ❸ Todas las hebras del equipo participan de la ejecución de tareas

## Ejecución de tareas y cambio de contexto

Por defecto, cuando una hebra comienza la ejecución de una tarea, ambos, tarea y hebra, quedan *unidos* (**tied**), es decir la misma hebra ejecutará la tarea de principio a fin

- 1 La ejecución de una tarea no es necesariamente continua (atómica)
- 2 Una hebra puede suspender la ejecución de su tarea en *puntos de sincronización*, y resumirla en un tiempo posterior
- 3 Cuando una hebra suspende la ejecución de la tarea actual, puede realizar un *task switch* a una tarea suspendida anteriormente
- 4 Si la hebra está **tied** a la tarea, sólo puede resumir tareas que ella misma creó

## Cláusula untied

Una tarea que es creada con la cláusula **untied**, no queda unida a ninguna hebra, y por lo tanto puede ser ejecutada (y resumida) por cualquier hebra del equipo

```
#pragma omp parallel
{
 ...
 #pragma omp single
 {
 for (i=0; < MANYTASKS; i++)
 #pragma omp task untied
 process(item[i]);
 }
}
```

- Se crean muchas tareas en el loop
- Sólo una hebra genera todas las tareas
- La tarea generadora puede ser suspendida, mientras otras comienzan a ejecutar tareas
- Cualquier otra tarea puede resumir la tarea de generación de tareas

## Cláusula **if**

- 1 Cuando el argumento de la cláusula **if** es falso
  - La tarea actual es suspendida
  - La tarea nueva es ejecutada inmediatamente por la hebra generadora
  - La tarea padre resume cuando la tarea termina
- 2 Usado como método manual de optimización
  - Cuando el costo de esperar es más grande que el costo de ejecución inmediato
  - Mejora uso de localidad (cache) y afinidad de memoria

# Sort con **task**

```

void sort(int *low, int *tmp, int size) {
 if (size < quick_size) {
 quicksort(low, low+size-1);
 return;
 }
 quarter = size/4; // asuma división OK
 A = low; tmpA = tmp;
 B = A+quarter; tmpB = tmpA+quarter;
 C = B+quarter; tmpC = tmpB+quarter;
 D = C+quarter; tmpD = tmpC+quarter;
 #pragma omp task
 sort(A, tmpA, quarter);
 #pragma omp task
 sort(B, tmpB, quarter);
 #pragma omp task
 sort(C, tmpC, quarter);
 #pragma omp task
 sort(D, tmpD, quarter);
 #pragma omp taskwait
 #pragma omp task
 merge(A,A+quarter-1, B, B+quarter-1, tmpA);
 #pragma omp task
 merge(C,C+quarter-1, D, D+quarter-1, tmpC);
 #pragma omp taskwait
 merge(tmpA, tmpC-1, tmpC, tmpA+size-1, A);
}

```

