High Performance Computing

SIMD con hebras. OpenMP

Fernando R. Rannou
Departamento de Ingeniería Informática
Universidad de Santiago de Chile

April 10, 2024

¿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!

Beneficios de OpenMP

- OpenMP permite desarrollar "fácilmente" aplicaciones paralelas (de memoria compartida) eficientes y de tamaño medio
- Es la forma más fácil de escribir aplicaciones multihebreadas
- OpenMP es más útil en aplicaciones que manipulen arreglos y matrices grandes.
- Compiladores explotan arquitecturas propias
- Portabilidad
- Experimentos muestran escalabilidad casi lineal hasta 128 procesadores
- Para aplicaciones grandes, es posible usar OpenMP y MPI y combinar modelos de memoria compartida junto con paso de mensajes

Hello world! en OpenMP/C

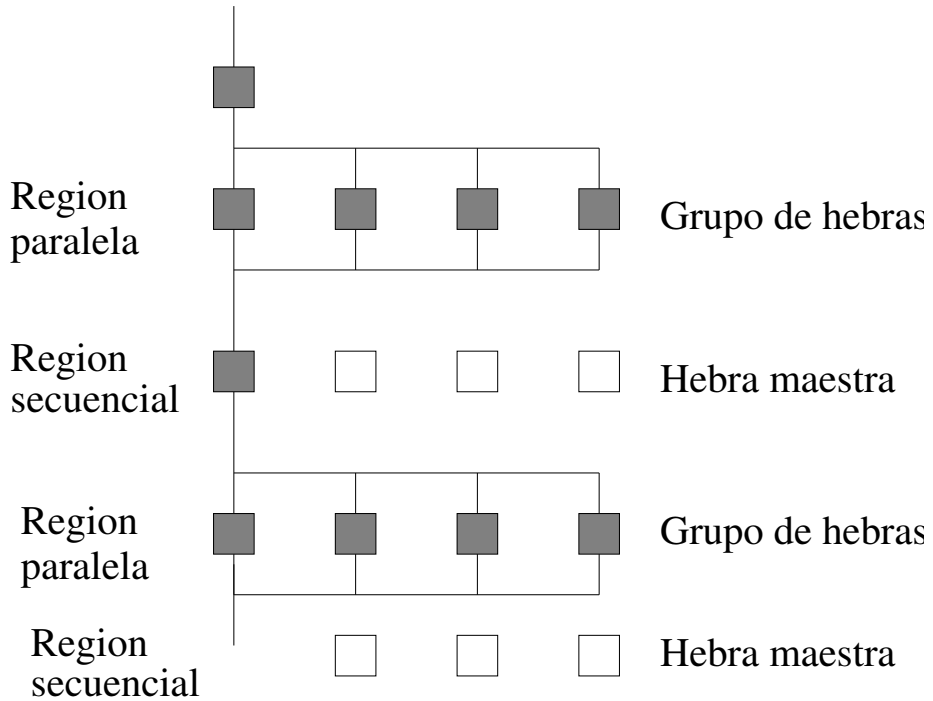
```
1  #include <stdio.h>
2  #ifdef _OPENMP
3  #include <omp.h>
4  #endif


5  void main() {
6       int ntasks = 3;

7  #pragma omp parallel num_threads(ntasks)
8  {
9          printf("Hello World!, numero de hebra = %d\n", omp_get_thread_num());
10 }
11 }

12 Hello World!, numero de hebra = 0
13 Hello World!, numero de hebra = 1
14 Hello World!, numero de hebra = 2
```

Modelo OpenMP de programación

- OpenMP es un modelo de programación de memoria compartida
- Se crean múltiples hebras (pool) para dividir el trabajo
- Las hebras se comunican a través de variables compartidas
- Las variables del programa pueden ser
  - Compartidas por todas las hebras del pool
  - Duplicadas en cada hebra, es decir locales
- OpenMP no resuelve problemas asociados a race conditions
- El programador debe usar mecanismos de sincronización para resolver estos problemas

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



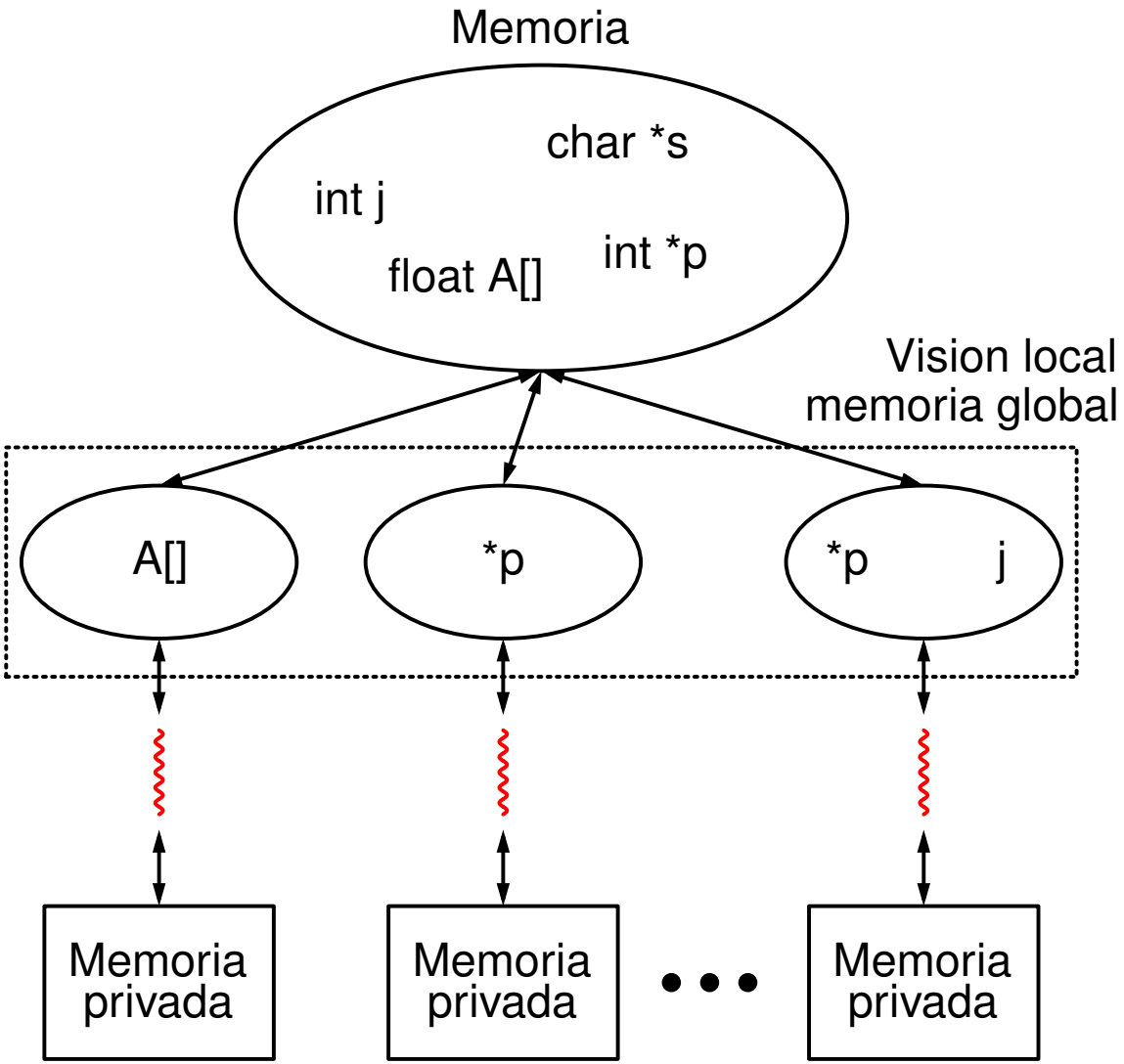
Ejemplo modelo de ejecución

```
1   int val1;

2   void main() {
3        int ntasks = 3;
4        int val3 = 2.0;
5   #pragma omp parallel num_threads(ntasks)
6   {
7           val1 = function1(omp_get_thread_num());
8   }
9           printf("Aqui codigo secuencial\n");
10  #pragma omp parallel num_threads(ntasks)
11  {
12          int val2;

13          val2 = function2(omp_get_thread_num());
14          printf("%d\", val1+val3);
15  }
16          printf("Aqui codigo secuencial\n");
17  }
```

Modelo OpenMP de memoria

- 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.

Model OpenMP de memoria (cont)



Modelo OpenMP de memoria (cont)

- OpenMP permite que la visión temporal de la memoria que tiene una hebra pueda ser distinto al estado actual de la memoria
  - Un valor escrito sobre una variable puede permanecer en la visión temporal de la hebra hasta que es forzada a actualizarse en la memoria
  - Una lectura de una variable puede retornar el valor en la visión temporal de la hebra, a menos que se fuerce la lectura de la memoria
- **Modelo de consistencia relajada** de memoria compartida
- La operación `flush` aplicada sobre un conjunto de variables, obliga consistencia entre la visión temporal de la memoria y la memoria, para esas variables.
- La operación `flush` elimina la visión temporal de la memoria de la hebra (para dichas variables)

Consistencia de memoria

- 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

Construcción región paralela

- Región de código que es ejecutada por varias hebras en paralelo
- Cada hebra ejecuta el mismo código redundantemente (ojo!)
- Sintáxis en C/++

```
#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

¿Qué es una directiva?

- `#pragma omp` *nombre_directiva* `[clause[[,] clause]..]]`
- En el caso anterior, `parallel` es el nombre de una directiva
- Algunas directivas múltiples (o combinadas) no están permitidas. Por ejemplo, la siguiente directiva produciría un error de compilación:

```
#pragma omp parallel barrier
```

Cláusulas de alcance (scope)

- `private` (*list*)
  Define que las variables declaradas en *list* serán privadas a cada hebra. Es decir, se tratan como variables locales.
- `shared` (*list*)
  Define que las variables declaradas en *list* serán variables compartidas por todas las hebras del grupo de trabajo
- Por defecto, las variables son compartidas. Sin embargo
  - Las variables locales de funciones invocadas por las hebras son privadas
  - Las variables automáticas dentro de un bloque son privadas
  - Ejemplo: la variable usada para un for-loop es privada

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

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**)

Funciones runtime en OpenMP

- Además de las directivas, OpenMP provee funciones de librería para
  1. Controlar y consultar el medio ambiente de la ejecución paralela
  2. Sincronizar acceso a datos compartidos
- Estas funciones son externas al lenguaje, y disponibles en forma de bibliotecas, que se linkean en forma automática.
- `int omp_get_num_threads(void)`
- `int omp_get_thread_num(void)`

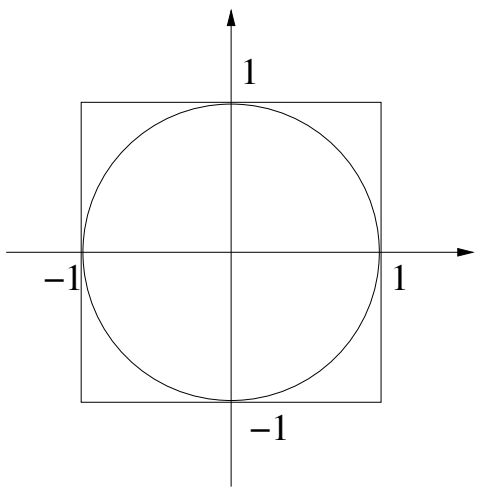
Método Monte Carlo para cálculo de π

- A partir de la figura, encontramos que

$$\frac{\text{Area circulo}}{\text{Area cuadrado}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

- También tenemos que

$$\int_0^1 \sqrt{1 - x^2}\, dx = \frac{\pi}{4}$$



- Una forma de calcular la integral, es mediante método Monte Carlo
- Generamos un secuencia de valores x_r entre 0 y 1; evaluamos la integral en x_r y acumulamos el resultado.

$$\pi = 4 \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} \sqrt{1 - x_r^2}$$

Cálculo secuencial de π

```
#define <stdio.h>
#define <math.h>
#define f(A) (sqrtf(1.0-(A*A)))
#define N 10000000

main() {
  float sum=0.0;
  int i;
  for (i=1;i<=N;i++) {
    x   = 1.0*i/N;
    sum = sum+f(x);
  }
  pi = 4.0*sum/N;
  printf("pi = %f\n", pi);
}
```

Cálculo secuencial Monte Carlo de π

El método Monte Carlo muestrea aleatoriamente el intervalo $[0, 1]$

```
#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);
}
```

Solución OpenMP Monte Carlo de π

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

6   #ifdef _OPENMP
7   #include <omp.h>
8   #endif

9   main() {
10      double sum=0.0, x, pi;
11      int i;
12      double sum0 = 0.0;
13  #pragma omp parallel private(x, sum0) shared(sum) num_threads(2)
14    {
15  #pragma omp for
16      for (i=1;i<=N;i++) {
17          x = drand48();
18         sum0 = sum0+f(x);
19      }
20  #pragma omp critical
21         sum += sum0;
22    }
23      pi = 4.0*sum/N;
24      printf("pi = %f\n", pi);
25  }
```

Resumen de lo visto hasta ahora

- OpenMP ofrece directivas para programar en paralelo con memoria compartida
- Se definen bloques paralelos que son ejecutados por varias hebras
- La ejecución es redundante, a menos que se usen directivas especiales
- Se puede declarar variables locales o privadas a cada hebra
- Se puede usar compilación condicional para mantener el código secuencial válido

Directiva `parallel`

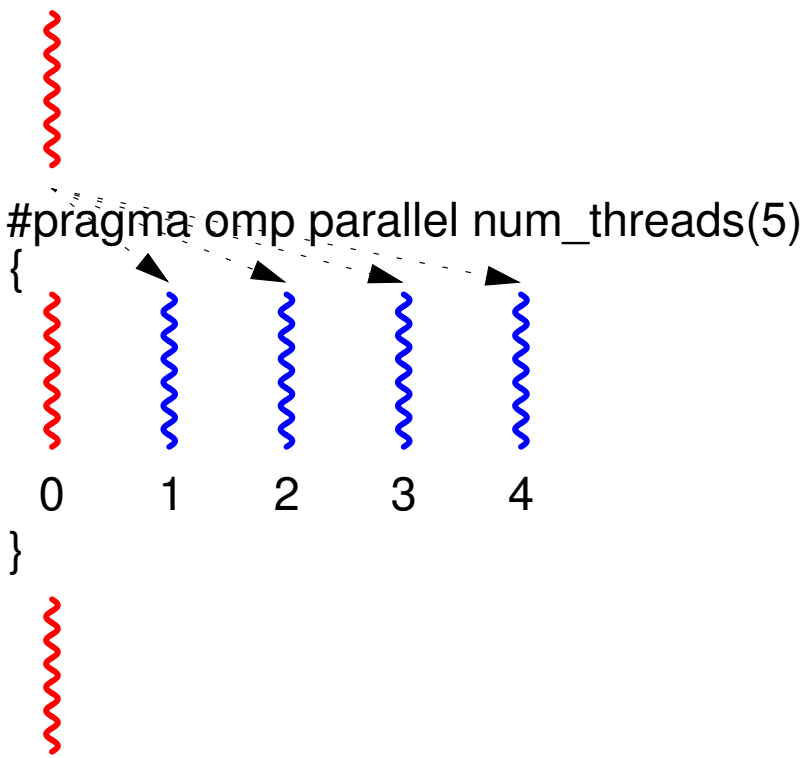
- La estructura sintáctica fundamental para crear bloques paralelos es

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

- donde las claúsulas 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 clausula `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)
```

Continuamos con `parallel`

- Una vez que el equipo de hebras es creado, cada una de ellas puede seguir un camino distinto de ejecución, por ejemplo, invocando a funciones distintas
- Al finalizar el bloque paralelo, la hebra maestra es la única que continúa su ejecución
- Si una hebra de un equipo en una región paralela encuentra otra construcción `parallel`, ella crea un nuevo equipo y pasa a ser la hebra maestra de dicho equipo

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 deactivado 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**:

Atributo de compartición predeterminado

Variables que poseen atributos predeterminados

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 de compartición explícito

Una variable referenciada dentro del alcance de una construcción, posee un atributo de compartición explícitamente determinado cuando aparece en la lista de una de las cláusulas de compartición de la construcción

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 aparace 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

default

`default` determina el atributo de compartición implícitamente determinado, de la siguiente manera

- `default (shared)`
  Todas las variables con atributos implícitos son `shared`
- `default (none)`
  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

`shared`

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

Consistencia relajada

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_thread_num(), x);
   }
}
```

```
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.

Construcciones para "repartir" el trabajo

- Usados para dividir el trabajo entre los miembros del equipo
- No se crean nuevas hebras, sino que se usan las ya existentes
- Existe una barrera implícita al final de la construcción, a menos que se use la cláusula `nowait`
- Las directivas son
  1. `for`
  2. `sections`
  3. `single`

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 clásulas 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`

Entendiendo schedule(static,)

```
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);
}
```

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

static

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);
   }
}
```

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

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);
}
```

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
```

```
#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);
   }
}
```

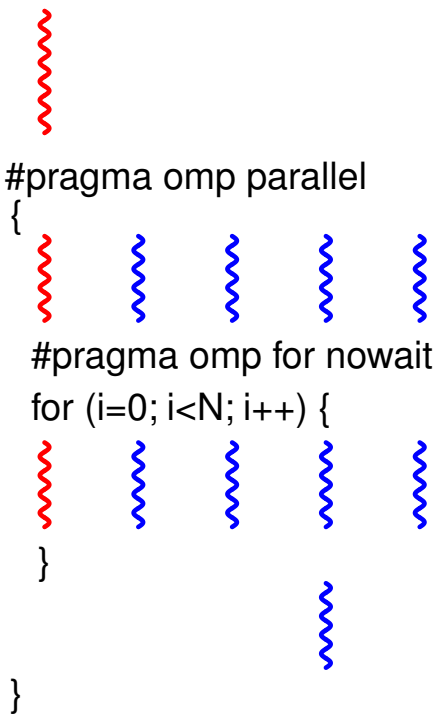
```
1, i = 2
1, i = 3
1, i = 4
1, i = 5
1, i = 6
1, i = 7
1, i = 8
0, i = 0
0, i = 1
```

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
```



ordered con ordered

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

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

```
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

Restricciones sobre el loop

La construcción `for` impone restricciones sobre la estructura del for-loop

```
for (init-expr; var relational-op b; incr-expr} ...
```

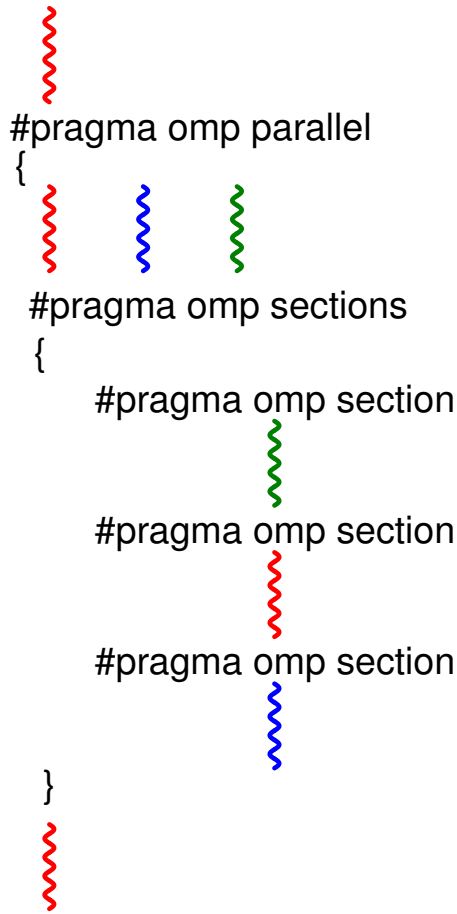
1. `init-expr` : expresión de asignación simple sobre variable entera.
2. `relational-op`: expresión de comparación simple sobre variables y expresiones enteras simples
3. `incr-expr`: expresión simple de incremento, decremento

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`



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 en 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);
```

```
1, en section 0
5, en section 1
total sum = 3
```

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");
    }
 }
```

lastprivate

1. Esta cláusula sólo se aplica a directivas de compartición de trabajo.
2. El último valor que toman las variables de esta lista son asignadas a los objetos originales.
3. En el caso de un `for`, el último valor se refiere al valor después de la última iteración; en el caso de `sections`, al valor después de la ejecución de la última sección.

copyprivate y single

Los valores de las variables listadas en la clásula 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);  }
void useValue(int s)  { printf("%d\n", s); }

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

```
6745

6745
6745
6745
```

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

threadprivate

```
#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

Cláusula inicialización de variables `threadprivate`

- `copyin` (lista-variables)
  Causa que las variables en la lista `threadprivate` tomen los valores que la hebra maestra tiene, justo antes de entrar a la región paralela
- `copyin` también puede usarse en construcciones `parallel for` y `parallel sections`

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
}
```

Construcciones para sincronización

- `master`

  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

```
#pragma omp master
     bloque estructurado
```

- `critical`

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

```
#pragma omp critical [(name])
   bloque estructurado
```

  Una hebra espera al comienzo del bloque hasta que ninguna otra hebra esté ejecutándose en un bloque con el mismo nombre (en cualquier parte del programa)

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
  ```

Construcciones para sincronización

- `atomic`
  Asegura que una localización específica de memoria es actualizada atómicamente

  ```
  #pragma omp atomic
      expresion
  ```

  `expresion` debe ser alguna de las siguientes:

  - *x binop= 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.

Construcciones para sincronización

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

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

  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 sunsecuentes deben ejecutarse hasta que todas la 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

Construcciones para sincronización

Una directiva `flush` está implícita en las siguientes directivas

- `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`

Construcciones para sincronización

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

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

  Una directiva `ordered` debe estar en el alcance dinámico de directivas `parallel` o `parallel for` que poseen una cláusula `ordered`

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
  1. No inicializado
  2. Unlocked (desbloqueado, no poseído, abierto)
  3. Locked (bloqueado, poseído, cerrado, tomado)
- Un lock puede ser poseído por solo una hebra (o tarea)

Locks simples

`omp_lock_t mylock;`

- `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 exclución mutua a una sección crítica

```
main() {
   omp_lock_t l = omp_init_lock();
   #pragma omp parallel num_threads(5)
   {
        omp_set_lock(&l);
        SC();
        omp_unset_lock(&l);
   }
 }
```

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

Locks anidados

- Un lock anidado puede ser bloqueado múltiples veces por la misma hebra
  `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

Funciones de librería

Además de las ya vistas, tenemos:

- `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+ hebras`
- `int omp_get_dynamic(void);`
  Retorna un valor distinto de cero si schedule dinámico de hebras está habilitado. Si no, retorna cero.

Funciones de librería

- `void omp_set_nested(int nested);`
  Si `nested == 0`, deshabilita paralelismo anidado (defecto), es decir regiones paralelas anidadas son serializadas y ejecutadas por la hebra actual
- `int omp_get_nested(void);`
  Retorna un valor distinto de cero si paralelismo anidado está habilitado. Si no, retorna cero.

Funciones de librerías para *timing*

- `double omp_get_wtime(void);`
  Retorna el tiempo de reloj, en segundos, que ha transcurrido desde un instante fijo en el pasado

```
double start;
double end;
start = omp\_get\_wtime();
....
end = omp\_get\_wtime();
printf("Tiempo usado = %f sec.\n", end-start);
```

- `double omp_get_wtick(void);`
  Retorna el tiempo de reloj en segundos entre sucesivos *clicks* de reloj

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);
       }
   }
}
```

Ejemplos task

Procesamiento de varias listas

```
#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.
```

Procesamiento de árbol binario

```
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);
       }
   }
}
```

1. La creación de tareas es realizada por sólo una hebra
2. La creación de tareas se ejecuta concurrentemente con la ejecución de las tareas
3. Todas las hebras del equipo participan de la ejecución de tareas

Ejecución de tareas y cambio de contexto

`tied`

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

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);
}
```

Referencias

El material de `task` proviene de:

The Design of OpenMP Tasks, IEEE Transactions on Parallel and Distributed Systems, vol. 20, No. 3, 2009.