17.8.4 for **Pragma**

El pragma paralelo puede también ser útil cuando se paralelizan los bucles
Considere este bucle doblemente anidado:

```
for ( i = 0; i < m; i++ )  {
    low = a [ i ] ;
    high = b [ i ] ;
    if ( low > high )       {
          printf ( “Exiting during iteration %d\n”, i ) ;
          break;
    }
    for ( j = low; j < high; j++ )
    c [ j ] = ( c [ j ] - a [ i ] / b [ j ] ;
}
```

No podemos ejecutar las iteraciones del bucle externo en paralelo, ya que contiene una sentencia break. Si ponemos un pragma paralelo del bucle indexados por j, habrá un paso fork / join por cada iteración del bucle externo. Nos gustaría evitar esta sobrecarga. Anteriormente demostramos como invirtiendo los bucles podría solucionar este problema, pero esta propuesta no funciona aquí debido a la dependencia de los datos.

Si ponemos el pragma paralelo inmediatamente delante del bucle indexados por i, entonces sólo tendremos un solo fork / join. El comportamiento predeterminado consiste en que cada hilo ejecuta todo el código dentro del bloque. Por supuesto, necesitamos los hilos para dividir las iteraciones del bucle interno.

```
#pragma omp for

/*
 *  OpenMP implementation of Monte Cristo pi-finding algoritm
 */

#include <stdio.h>
int main ( int argc, char *argv [] ) {
      int         count ;             /*  Points inside unit circle  */
      int         i ;
      int         local_count ;       /*  This thread’s subtotal  */
      int         samples ;           /*  Points to generate  */
      insignet  short xi ( 3 ) ;      /*  Random number seed  */
      int         t ;                 /*  Number of threads  */
      int         tid ;               /*  Thread id  */
      double    x, y ;                /*  Coordinates of point  */

      /*  Number of points and numbers of threads are command-line arguments  */

      samples = atoi ( argv [ 1 ] ) ;
      omp_sat_num_threads ( atoi (argv [ 2 ]) ) ;

      count = 0

$pragma omp parallel private ( xi, t, i, x, y, local_count )
      {
                local_count = 0 ;
                xi [ 0 ]  =  atoi ( argv [ 3 ] ) ;
```

```
            xi [ 1 ]  =  atoi ( argv [ 4 ] ) ;
            xi [ 2 ]  =  tid  = omp_get_thread_num () ;
            t  =  omp_get_num_threads () ;

            for  ( i = tid; i < samples; i += t ) {
                x = erand48 ( xi ) ;
                y = erand48 ( xi ) ;
                if ( x * x + y * y <= 1.0 )  local_count++ ;
            }
$pragma omp critical
            count  +=  local_count ;
    }
    printf ( “Estimate of pi : %7, %f\n” , 4.0 * count / samples ) ;
}
```

Figure 17.9 Este programa C/OpenMP usa el metodo de Monte Carlo para calcular π.

Con este programa añadió, nuestro segmento de código se parece a esto:

```
#pragma omp parallel private ( i, j )
for  ( i = 0, i < m; i++ )  {
    low  =  a [ i ] ;
    high  =  b [ i ] ;
    if  ( low > high ) ;
        printf  ( “Exiting during iteration %d\n”, i  ) ;
        break ;
    }
#pragma omp for
    for ( j = low; j < high; j++ )
        c [ j ]  =  ( c [ j ]  -  a [ i ] / b [i] ) ;
}
```

Sin embargo, nuestro trabajo no está complete aún.

17.8.5 single **Pragma**

Hemos paralelizado la ejecución del bucle indexadas por j. ¿Y el otro código dentro del bucle externo? Desde luego, no queremos ver de nuevo el mensaje de error.

El pragma solo indica al compilador que sólo un único hilo debe ejecutar el bloque de código que precede al pragma. Su sintaxis es:

```
#pragma omp single
```

Añadiendo el pragma solo al bloque de código, ahora tenemos esto:

```
#pragma omp parallel private ( i, j )
for  ( i = 0, i < m; i++ ) {
    low  =  a [ i ] ;
    high  =  b [ i ] ;
    if  ( low > high ) {
#pragma omp single
        printf ( “Exiting during iteration %d\n”, i ) ;
        break ;
    }
#pragma omp for
```

```
        for  ( j = low, j < high; j++ ) ;
            c [ j ] = ( c [ j ] – a [ i ] / b [ i ] ;
}
```

El bloque del codigo ahora se ejecuta correctamente, pero podemos mejorar su rendimiento

17.8.6 nowait **Clause**

El compilador coloca una barrera de sincronización al final de cada paralelo para declaración. En el ejemplo la hemos considerado, esta barrera es necesaria, porque necesitamos estar seguros que cada hilo ha completado una iteración del bucle indexado por i antes de que cualquier hilo comience la siguiente iteración. Por otro lado, un hilo puede cambiar el valor por algo o bajo, alterando el número de iteraciones del bucle j realizada por otro hilo.

Por el otro lado, si hacemos variables privadas altas y bajas, no hay necesidad de la barrera al final del bucle indexado por j. La cláusula nowait, añadida a una paralela para pragma, le indica al compilador que omita la sincronización de barrera al final de la paralela del bucle

Después de crear altas y bajas privadas y añadir la cláusula nowait, nuestra versión final de nuestro segmento de código es:

```
#pragma omp parallel private ( i, j, low, high )
for  ( i = 0, i < m ; i ++ ) {
      low  =  a [ i ] ;
      high  =  b [ i ] ;
      if  ( low > high ) {
#pragma omp single
          printf  ( “Exiting during iteration %d\n”, i ) ;
          break ;
      }
#pragma omp for nowait
      for  ( j = low; j < high; j++ )
          c [ j ]  =  ( c [ j ] – a [ j ] ) /b [ i ] ;
}
```

17.9 PARALLELISMO FUNSIONAL

Para este punto nos hemos centrado exclusivamente en la exploración del paralelismo de datos. Otra fuente de concurrencia es el paralelismo funcional. OpenMP nos permite asignar diferentes hilos a diferentes partes del código.

Considere el siguiente ejemplo de segmento de código

```
v  =  alpha () ;
w  =  beta () ;
x  =  gamma () ;
y  =  delta () ;
printf ( “%6.2f\n”, epsilon (x, y) ) ;
```

Si todas las funciones son libre de efectos secundarios, podemos representar las dependencias de datos, como se muestra en la Figura 17.10. Es evidente que las funciones de alfa, beta y delta se pueden ejecutar en paralelo. Si se ejecuta estas funciones al mismo tiempo, no hay más paralelismo funcional para explotar, porque la función gamma debe ser llamada después de las funciones alfa, beta y épsilon y antes de la función.

17.9.1 parallel section **Pragma**

Las secciones de pragma paralelo preceden a un bloque de bloques k de código que pueden ser ejecutadas al mismo tiempo por hilos k. Tiene la siguiente sintaxis:

#pragma omp parallel sections

17.9.2 section **Pragma**

La sección pragma procede a cada bloque de código que abarca el bloque que precede por las secciones paralelas del pragma. (La sección del pragma puede ser omitida por la primera sección paralela después de la sección paralela del pragma)

En el ejemplo consideramos que pueden ser evaluadas al mismo tiempo las llamadas a la función alpha, beta, y delta. En nuestra paralelizacion de este segmento de código usamos llaves para crear un bloque de código conteniendo estos 3 estados asignados. (Recuerda que un estado asignado es un ejemplo trivial de un código de bloque. Por lo tanto un bloque que contiene tres instrucciones de asignación es un bloque de tres bloques de código.)

```
#pragma omp parallel sections
    {
#pragma omp section             /*  This pragma optional  */
        v = alphra () ;
#pragma omp section
        w = beta () ;
#pragma omp section
        y = delta () ;
    }
    X = gamma ( v, w ) ;
    printf ( “%6.2f\n”, epsilon ( x, y ) ) ;
```

Tenga en cuenta que reordenamos de nuevo las declaraciones de asignación para reunir a los tres y poder ejecutarlos en paralelo.

17.9.3 sections **Pragma**

Vamos a darle otro vistazo al diagrama de dependencia de los datos de la figura 17.10. Hay una segunda forma de explotar el paralelismo funcional en este segmento de código. Como hemos señalado antes, si ejecutamos funciones alfa, beta y delta en paralelo, no hay mayores oportunidades para el paralelismo funcional. Sin embargo, si ejecutamos funciones alfa y beta en paralelo, entonces después de su regreso es posible ejecutar funciones gamma y delta en paralelo.

En este diseño tenemos dos secciones paralelas diferentes, una seguida de la otra. Podemos reducir fork / join a coste poniendo todos los cuatro estados de asignación en un solo bloque precedido por el pragma paralelo, entonces las secciones pragma para identificar los primero y segundo pares de funciones que pueden ejecutar en paralelo.

Las secciones del pragma con la sintaxis

#pragma omp sections

Aparece dentro de un bloque paralelo de código. Se tiene exactamente el mismo significado que las secciones paralelas pragma que ya hemos descrito.

Aquí es otra manera de expresar el paralelismo funcional en el segmento de código que hemos estado considerando, mediante el pragma de secciones:

```
#pragma omp parallel
    {
    #pragma omp sections
        {
        #pragma omp sections          /*  This pragma optional  */
            v = alpha () ;
        #pragma omp sections
            w = beta () ;
        }
    #pragma omp sections
        {
        #pragma omp section           /*  This pragma optional  */
            x = gamma ( v, w ) ;
        #pragma omp section
            y = delta () ;
        }
    }
    printf  ( “%6.2f\n”, epsilon ( x, y ) ) ;
```

En un aspecto, esta solución es mejor que la primera que presentamos, ya que cuenta con las dos secciones paralelas de código, cada uno requiere dos hilos. Nuestra primera solución tiene sólo una única sección paralela de código que requiere tres hilos. Si sólo dos procesadores están disponibles, la segunda sección de código podría resultar con mayor eficiencia. Sea o no el caso depende de los tiempos de ejecución de las funciones individuales.