Tema 3. Paralelismo en bucles

Programación Paralela

José María Cecilia

Grado en Informática

1. Paralelismo en bucles

El paralelismo de bucles es un tipo de paralelismo muy común en los códigos científicos. OpenMP tiene un mecanismo para paralelizar bucles de manera sencilla. Los bucles paralelos de OpenMP son un primer ejemplo de las construcciones de "trabajo compartido" de OpenMP. La ejecución paralela de un bucle se puede manejar de diferentes formas. Por ejemplo, se puede crear una región paralela alrededor del bucle y ajustar los límites del bucle:

```
#pragma omp parallel
{
  int threadnum = omp_get_thread_num (),
  numthreads = omp_get_num_threads ();
  int bajo = N * threadnum / numthreads,
  alto = N * (threadnum + 1) / numthreads;
  for (i = bajo; i <alto; i ++)
    // hacer algo con i
}
```

Una opción más natural es usar el pragma parallel for:

```
#pragma omp parallel

#pragma omp for

for (i = 0; i <N; i ++) {
```

Esto tiene varias ventajas. Por un lado, no se tienen que calcular los límites del bucle para los diferentes hilos, y además también se puede decir a OpenMP que asigne las iteraciones del bucle a los diferentes hilos según diferentes planificaciones.

La Figura 1 muestra la ejecución de cuatro subprocesos del siguiente código

```
#pragma omp parallel
{
 code1 ();
#pragma omp for
  for (i = 1; i <= 4 * N; i ++)
    code2 ();
  code3 ();
}
```


José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

El código antes y después del bucle se ejecuta de forma idéntica en cada hilo; las iteraciones del bucle se reparten entre los cuatro hilos.

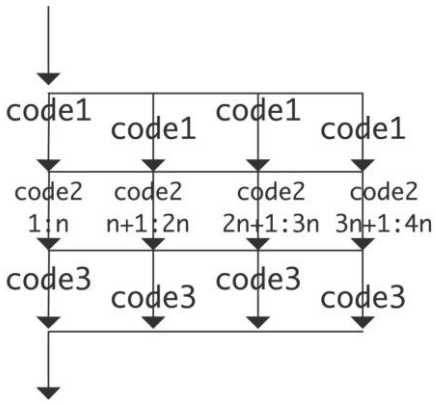

Figura 1 Ejecución de un código con 4 hilos dentro y fuera del bucle. Efectos del pragma parallel for.

La directiva *omp for* no crea un conjunto de *threads*. Esta directiva utiliza el conjunto de *threads* que está activo y divide las iteraciones del bucle entre los subprocesos. Esto significa que la directiva *omp for* debe estar dentro de una región paralela. También es posible tener una directiva combinada #pragma omp *parallel for*:

```
#pragma omp parallel for
 for (i = 0; .....)
```

Ejercicio

Calcule el número π por integración numérica. Este algoritmo usa el hecho de que π es el área del círculo unitario, y se aproxima calculando el área de un cuarto de círculo usando la serie de Leibniz.

$$\frac{\pi}{4} \cong \sum_{i=0}^{N-1} \frac{-1^i}{2i+1} = 1 - \frac{1}{3} + \frac{1}{5} - \cdots = \frac{pi}{4}$$

Escribe un programa en OpenMP usando las siguientes estrategias:

- Pon una directiva *parallel* alrededor de tu bucle. ¿Calcula el resultado de manera correcta? ¿Se reduce el tiempo de ejecución con el incremento del número de hilo? (Respuesta debería ser no y no)

José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

- Cambia la directiva *parallel* por *parallel for*. ¿El resultado es correcto? ¿se consigue un aceleración del código? (Respuesta debería ser no y si)
- Pon un directiva *critical* delante de la actualización. ¿El resultado es correcto? ¿se consigue un aceleración del código? (Respuesta si y no demasiado)
- Elimina la directiva *critical* y añade la cláusula *reduction (+:cuartopi)* en la directiva for. Ahora debería ser correcto y eficiente. Usa diferente número de hilos y calcula el speed-up sobre el código sequencial. Hay alguna diferencia en rendimiento entre OpenMP con 1 hilo y el código secuencial. Haz una gráfica para mostrar los resultados de sequencial, openmp 1, 2, 4, 8, 16 hilos.

Nota: Es posible que en este ejercicio hayas percibido subidas en el tiempo de ejecución del código paralelo. Esto es debido fundamental al *false sharing.*

Existen algunas restricciones en la paralelización de bucles en OpenMP. Básicamente, OpenMP necesita conocer con antelación cuántas iteraciones realizará el bucle.

- El bucle no puede contener sentencias break, return, exit, goto.
- Se permite la declaración de continue.
- La actualización de índices tiene que ser un incremento (o disminución) en una cantidad fija.
- La variable de índice de bucle es privada por defecto, y no se permiten cambios dentro del bucle.

2. Planificadores de bucle

En la paralelización de bucles es normal encontrar más iteraciones en un bucle que hilos se han establecido. Estas iteraciones hay que repartirlas entre los hilos en ejecución. Existen varias maneras de hacerlo mediante la cláusula *schedule*

```
#pragma omp for schedule(....)
```

La primera distinción que realizamos es entre planificadores ***estáticos*** y ***dinámicos***. En el caso de la planificación estática, las iteraciones se asignan simplemente en función del número de iteraciones y del número de hilos (y del parámetro *chunk*; ver más adelante). En los programas dinámicos, por otra parte, las iteraciones se asignan a hilos que están desocupados. Los planificaciones dinámicas son una buena idea si las iteraciones toman una cantidad de tiempo impredecible, por lo que es necesario equilibrar la carga.

José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

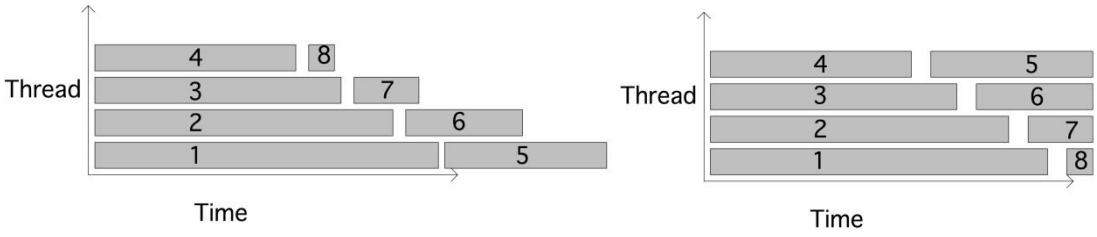

Figura 2. Estrategias de planificación de iteraciones en bucles

La estrategia por defecto es ***estática***, asignado un bloque consecutivo de iteraciones a cada hilo. Si desea bloques de diferentes tamaños, puede definir un tamaño *chunk*

```
#pragma omp for schedule(static[,chunk])
```

En la documentación OpenMP los corchetes indican un argumento opcional. Con la programación estática, el compilador dividirá las iteraciones de bucle en tiempo de compilación, así que, siempre y cuando las iteraciones sean aproximadamente igual de pesadas, esta estrategia es la más eficiente. La elección de un tamaño de chunk es una situación de compromiso entre tener sólo unos pocos trozos, frente al balanceo de carga por tener trozos más pequeños.

Nota: El tamaño de chunk es una de las posibles variables para evitar el *false sharing*.

Ejercicio

¿Por qué un chunk de 1 es una mala idea? (Consejo: piense en las líneas de caché y false sharing)

La estrategia ***dinámica*** de OpenMP colocará bloques de iteraciones (el tamaño predeterminado es 1) en una cola de tareas, y los hilos tomarán una de estas tareas cada vez que terminen con la anterior.

```
#pragma omp for schedule(dynamic[,chunk])
```

Aunque la planificación dinámica puede ofrecer un buen balanceo de carga, si las iteraciones tardan tiempos muy diferentes para ejecutarse, se incluye un sobrecoste de tiempo de ejecución para gestionar la cola de las tareas de iteración.

Por último, está la opción de planificación guiada (***scheduled guided)***, disminuye gradualmente el tamaño de los trozos. La idea es que los trozos grandes son los que llevan menos sobrecarga, pero los trozos más pequeños son mejores para equilibrar la carga. En la Figura 3 se ilustran los distintos modos de planificación.

José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

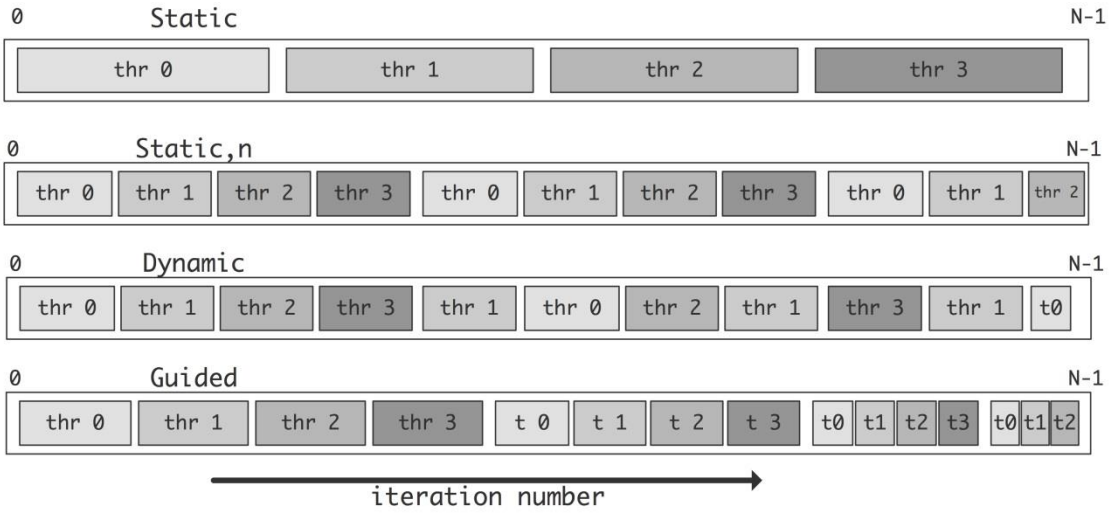

Figura 3. Diferentes modos de planificación de iteraciones en OpenMP

Ejemplo. Planificación de hilos en paralelización de bucles. Dado el siguiente código, varíe la variable de entorno OMP_SCHEDULE para ver las diferentes técnicas de reparto de iteraciones en el bucle

```c
/******************************************************************************
 sched.c
 Ejemplo para analizar la planificacion o scheduling
 la planificacion se indica en una variable de entorno (Runtime)
 ejemplo: export OMP_SCHEDULE="static,4"
******************************************************************************/
#include <omp.h>
#include <stdio.h>
#include <unistd.h>
#define N 40
main ()
{
 int tid;
 int A[N];
 int i;
    for(i=0; i<N; i++) A[i]=-1;
    #pragma omp parallel for schedule(runtime) private(tid)
    for (i=0; i<N; i++)
    {
        tid = omp_get_thread_num();
        A[i] = tid;
        usleep(1);
    }
    for (i=0; i<N/2; i++) printf (" %2d", i);
    printf ("\n");
    for (i=0; i<N/2; i++) printf (" %2d", A[i]);
    printf ("\n\n\n");
    for (i=N/2; i<N; i++) printf (" %2d", i);
    printf ("\n");
    for (i=N/2; i<N; i++) printf (" %2d", A[i]);
    printf ("\n\n");
    return 0;
}
```


José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

3. Reducciones

Hasta ahora nos hemos centrado en bucles con iteraciones independientes. Las reducciones son un tipo común de bucle con dependencias. OpenMP incorpora la cláusula reduction

```
#pragma omp parallel for reduction(operator:list)
 for (i = 0; .....)
```

Esta cláusula realiza una operación de reducción sobre las variables que aparecen en su lista. Se crea una copia privada para cada variable de lista y se inicializa para cada hilo. Al final de la reducción, la variable de reducción se aplica a todas las copias privadas de la variable compartida, y el resultado final se escribe en la variable compartida global.

Ejemplo:

```
1  #include <omp.h>
2
3  int main(int argc, char *argv[])  {
4
5   int   i, n, chunk;
6   float a[100], b[100], result;
7
8   /* Some initializations */
9   n = 100;
10  chunk = 10;
11  result = 0.0;
12  for (i=0; i < n; i++) {
13    a[i] = i * 1.0;
14    b[i] = i * 2.0;
15  }
16
17  #pragma omp parallel for      \
18    default(shared) private(i)  \
19    schedule(static,chunk)     \
20    reduction(+:result)
21
22   for (i=0; i < n; i++)
23      result = result + (a[i] * b[i]);
24
25  printf("Final result= %f\n",result);
26
27 }
28
```

La Tabla 1 muestra las operaciones de reducción que se pueden realizar con esta cláusla y su sintaxis en C.

José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

Operation	C/C++	Initialization
Addition	+	**0**
Multiplication	*	**1**
Subtraction	-	**0**
Logical AND	&&	**0**
Logical OR	||	**.false. / 0**
AND bitwise	&	**Todos los bits on / 1**
OR bitwise	|	**0**
Exclusive OR bitwise	^	**0**
Equivalent		**.true.**
Not Equivalent		**.false.**
Maximum	**max**	**Most negative #**
Minimum	**min**	**Largest positive #**

Tabla 1 Operaciones de reducción de la cláusla reduction

4. Colapsar bucles anidados.

En general, mientras más trabajo haya para dividir entre un número de hilos, más eficiente será la paralelización. En el contexto de los bucles paralelos, es posible aumentar la cantidad de trabajo paralelizando todos los niveles de bucles en lugar de sólo el exterior.

Ejemplo:

```
for ( i=0; i<N; i++ )
 for ( j=0; j<N; j++ )
 A[i][j] = B[i][j] + C[i][j]
```


José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

Las N^2 iteraciones son independientes. Una directiva openmp for sólo paralelizaría 1 nivel. La directiva collapse paralelizaría más de un nivel

```
#pragma omp for collapse(2)
for ( i=0; i<N; i++ )
 for ( j=0; j<N; j++ )
 A[i][j] = B[i][j] + C[i][j]
```

Sólo es posible colapsar bucles perfectamente anidados, es decir, el cuerpo del bucle exterior puede sólo puede consistir en el bucle interior; no puede haber sentencias antes o después del bucle interior en el cuerpo de bucle del bucle exterior. Es decir, los dos bucles en el siguiente ejemplo no se podrían colapsar.

```
for (i=0; ilt;N; i++) {
 y[i] = 0.;
 for (j=0; jlt;N; j++)
 y[i] + A[i][j] * x[j]
 }
```

5. Iteraciones ordenadas

Las iteraciones en un bucle paralelo que se ejecutan en paralelo no se tienen que ejecutar en lock-step (en orden secuencial). Eso significa que en el siguiente código

```
#pragma omp parallel for
 for ( ... i ... ) {
 ... f(i) ...
 printf("something with %d\n",i);
 }
```

no es cierto que todas las evaluaciones de funciones se efectúen más o menos al mismo tiempo, seguidas de todas las sentencias de impresión. Las declaraciones printf pueden realmente suceder en cualquier orden. La cláusula de "ordered" puede forzar la ejecución en el orden correcto:

```
#pragma omp parallel for shared(y) ordered
 for ( ... i ... ) {
 int x = f(i)
 #pragma omp ordered
 y[i] += f(x)
 z[i] = g(y[i])
 }
```

Hay una limitación: cada iteración sólo puede encontrar una directiva ordered. La directiva ORDERED especifica que las iteraciones del bucle cerrado se ejecutarán en el mismo orden que si se ejecutaran en un procesador serie. Los hilos tendrán que esperar antes de ejecutar su porción de iteraciones si las iteraciones anteriores no se han completado todavía. La directiva ORDERED proporciona una manera de "afinar" cuando se aplica dentro de un bucle. De lo contrario, no es necesario.

José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

Tiene alguna restricción

- Sólo se permite un hilo en una sección ordenada en cualquier momento
- Es ilegal ramificarse dentro o fuera de un bloque ORDERED.
- Una iteración de un bucle no debe ejecutar la misma directiva ORDERED más de una vez, y no debe ejecutar más de una directiva ORDERED.
- Un bucle que contenga una directiva ORDERED, debe ser un bucle con una cláusula ORDENADA.

6. Nowait

La barrera implícita al final de una región paralela puede ser cancelada con una cláusula nowait. Esto tiene el efecto de que los hilos que terminen pueden continuar con el siguiente código en la región paralela:

```
#pragma omp parallel
 {
 #pragma omp for nowait
 for (i=0; i<N; i++) { ... }
 // more parallel code
 }
```

En el siguiente ejemplo, los hilos que se terminan con el primer bucle pueden comenzar en el segundo. Tenga en cuenta que esto requiere que ambos bucles tengan la misma planificación.

```
#pragma omp parallel
 {
 x = local_computation()
 #pragma omp for nowait
 for (i=0; ilt;N; i++) {
 x[i] = ...
 }
 #pragma omp for
 for (i=0; ilt;N; i++) {
 y[i] = ... x[i] ...
 }
 }
```

7. Bucles while

OpenMP sólo puede manejar bucles `for': los bucles while no se paralelizan. Los bucles while se deben evitar cuando se pretende paralelizar con OpenMP.

```
while ( a[i]!=0 && i<imax ) {
         i++;
}
// Aqui I es el primer  indice donde I no es cero
```

```
result = -1;
 #pragma omp parallel for
```


José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

```
for (i=0; i<imax; i++) {
        if (a[i]!=0 && result<0) result = i;
}
```

Ejercicio: Muestra que este código tiene una condición de carrera. ¿Dónde está esa condición de carrera?

Se puede arreglar la condición de carrera haciendo que la condición esté una sección crítica. En este ejemplo en particular, con una cantidad muy pequeña de trabajo por iteración, es probable que sea ineficiente en este caso (¿por qué?).Solución más eficiente usa el pragma {lastprivate}

```
result = -1;
 #pragma omp parallel for lastprivate(result)
 for (i=0; ilt;imax; i++) {
 if (a[i]!=0) result = i;
 }
```

Ahora se ha resuelto un problema ligeramente diferente: la variable result contiene la última ubicación donde a[i] es cero.

8. Qué es el False Sharing?

La mayoría de los procesadores de alto rendimiento, como los procesadores UltraSPARC, insertan un búfer de caché entre la memoria principal (más lenta) y los registros de alta velocidad de la CPU. El acceso a una dirección de memoria provoca que una porción de la memoria real (una línea de caché) contenga la ubicación de memoria que se desea copiar en la caché. Las referencias posteriores a la misma ubicación de memoria o a las que le rodean probablemente pueden ser satisfechas fuera de la caché hasta que el sistema determine que es necesario mantener la coherencia entre la caché y la memoria.

Sin embargo, las actualizaciones simultáneas de datos individuales en la misma línea de caché procedentes de diferentes procesadores invalidan líneas de caché completas, aunque estas actualizaciones son lógicamente independientes entre sí. Cada actualización de un dato individual de una línea de caché marca la línea como no válida. Otros procesadores que acceden a un dato diferente en la misma línea ven la línea marcada como no válida. En este caso, se ven obligados a recoger una copia válida del bloque de memoria de otro lugar, aunque el elemento al que se accede no haya sido modificado. Esto se debe a que la coherencia de caché se mantiene en base a un bloque (línea) de caché, y no en datos individuales. Como resultado, se producirá un ***overhead*** debido al aumento del tráfico de interconexión, gestión del SSOO, entre otros. Además, mientras se está produciendo la actualización de la línea de caché, el acceso a los elementos de este bloque están parados.

Esta situación se llama ***false sharing.*** Si esto ocurre con frecuencia, el rendimiento y la escalabilidad de una aplicación OpenMP se verán afectados significativamente. El ***false sharing,*** degrada el rendimiento cuando se dan todas las condiciones siguientes:

José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu

- Los datos compartidos son modificados por múltiples procesadores.
- Varios procesadores actualizan los datos dentro de la misma línea de caché.
- Las actualizaciones se producen con mucha frecuencia (por ejemplo, en un bucle muy condensado).
- Los datos compartidos que son de sólo lectura en no tiene problemas de ***false sharing***.

Reducir el false sharing

El análisis cuidadoso de los bucles paralelos que juegan un papel importante en la ejecución de una aplicación puede indicar problemas de escalabilidad del rendimiento causados por *false sharing*. En general, el *false sharing* se puede reducir mediante:

- Haciendo uso de los datos privados tanto como sea posible.
- Utilizando las características de optimización del compilador para eliminar las lecturas y escrituras de/en memoria.
- En casos específicos, el impacto de la falsa distribución puede ser menos visible cuando se trata de problemas de mayor tamaño, ya que puede haber menos compartición.
- Las técnicas para hacer frente a la falsa distribución dependen en gran medida de la aplicación concreta. En algunos casos, un cambio en la forma en que se asignan los datos puede reducir la falsa distribución. En otros casos, cambiar el mapeo de las iteraciones a hilos, dando a cada hilo más trabajo por trozo (cambiando el valor de tamaño de trozo) también puede llevar a una reducción en la falsa compartición.

José M. Cecilia - Tlf: (+34) 968 278587 - jmcecilia@ucam.edu