Práctica 1: Computación de tipo stencil: Solución con OpenMP

Daniel Ballesteros Álvarez
Hernán Maximiliano González Calderón
Universidad de Valladolid
Escuela Técnica Superior de Ingeniería Informática

15 de noviembre de 2013

1. Decisiones de diseño

En el código a paralelizar se han encotrado cuatro puntos paralelizables, que son los siguientes:

- El bucle encargado de copiar la matriz de escritura a la de lectura en cada vuelta del bucle principal
- El bucle para crear la frontera periódica de la matriz
- El bucle principal encargado de calcular la nueva matriz
- El bucle encargado de calcular la suma de comprobación

1.1. Bucle de copia de matriz

Para este bucle el primer acercamiento consistió en utilizar la directiva `#pragma omp parallel for` estableciendo como variables compartidas `rows`, `cols`, `matrix1` y `matrix2` y como variables privadas `i` y `j`. A continuación puede verse el fragmento de código modificado.

```
do {

                resid = 0.0;

                // Iteración del algoritmo
                update(matrix1,matrix2,rows,cols);

                // Actualizar la copia
                #pragma omp parallel default(none)\
                shared(rows,cols,matrix1,matrix2)\
                private(j)
                {
                #pragma omp for
                for (i=1; i<rows+1; i++) {
                                for (j=1; j<cols+1; j++) {
                                        m1(i,j) = m2 (i,j);
                        }
                }
                }

...
        } while (resid > MAX_RESID);
```

Sin embargo, esta solución no es la óptima, dado que la copia de la matriz se hace elemento a elemento, lo que supone una carga innecesaria. Una alternativa más eficiente es intercambiar los punteros de las matrices de lectura y escritura, tal y como puede verse en el siguiente fragmento de código.

```
do {
...
                // Actualizar la copia
                m_type * aux = matrix1;
                matrix1 = matrix2;
                matrix2 = aux;
...
} while (resid > MAX_RESID);
```

1.2. Bucle de generación de frontera periódica

A la hora de paralelizar este fragmento surgieron dos propuestas:

La primera era usar directivas `#pragma omp for` para cada uno de los bucles y la segunda era usar una directiva `#pragma omp for nowait` y una `#pragma omp for` para los laterales de la matriz y usar para copiar los elementos de las esquinas dentro de una directiva `#pragma omp sections`.

Diversas pruebas demostraron un funcionamiento ligeramente mejor con la primera alternativa. A continuación puede verse el fragmento de código de la primera alternativa.

Para esta región paralela se ha definido `rows`, `cols`, `matrix_new`, `matrix_old` y `resid` como variables compartidas; e `i`, `j` y `tmpResid` como variables privadas.

```
...
        // 2. Copia de los bordes
        int i, j;
        m_type tmpResid;
        #pragma omp parallel default(none)\
        shared(rows,cols,matrix_new,matrix_old,resid)\
        private(j,tmpResid)
        {
        #pragma omp for //nowait
        for(i=1; i<rows+1; i++){
                m_old(i,0) = m_old(i,cols);
                m_old(i,cols+1) = m_old(i,1);
        }
        #pragma omp for //nowait
        for(j=0; j<cols+2; j++){
                m_old(0,j) = m_old(rows,j);
                m_old(rows+1,j) = m_old(1,j);
        }
...
```

El segundo caso se ve reflejado en el siguiente fragmento.

```
...
        // 2. Copia de los bordes
        int i, j;
        m_type tmpResid;
        #pragma omp parallel default(none)\
        shared(rows,cols,matrix_new,matrix_old,resid)\
        private(j,tmpResid)
        {
        #pragma omp for nowait
        for(i=1; i<rows+1; i++){
                m_old(i,0) = m_old(i,cols);
                m_old(i,cols+1) = m_old(i,1);
        }
        #pragma omp for
        for(j=1; j<cols+1; j++){
                m_old(0,j) = m_old(rows,j);
                m_old(rows+1,j) = m_old(1,j);
        }

        #pragma omp sections
        {
        #pragma omp section
        m_old(0,0) = m_old(rows,0);
        #pragma omp section
        m_old(rows+1,0) = m_old(1,0);
        #pragma omp section
        m_old(0,cols+1) = m_old(rows,cols+1);
        #pragma omp section
        m_old(rows+1,cols+1) = m_old(1,cols+1);
        }
...
```

1.3. Generación de nueva matriz

La paralelización de esta zona fue inmediata con la utilización de la directiva `#pragma omp for`. Dado que este bucle y los anteriores están recogidos en la misma región paralela no ha sido necesario preocuparse por la definición de las variables compartidas y privadas.

```
        // 3. Bucle principal
        #pragma omp for
        for(i=1; i<rows+1; i++){
           for(j=1; j<cols+1; j++){
              // 3.1 El nuevo valor es el anterior más la acción de los vecinos.
                        m_new (i,j) = ( m_old (i, j) * WEIGHT_CENTER) +
...
```

```
...
                        // 3.2 Guardamos la mayor diferencia entre el valor
                        //antiguo y el actualizado
                        {
                        //m_type tmpResid = fabs (m_old(i, j) - m_new(i, j));
                        tmpResid = fabs (m_old(i, j) - m_new(i, j));
                        if ( tmpResid > resid ) resid = tmpResid;
                        }

                }
        }
        }
```

1.4. Suma de comprobación

Este último fragmento puede resultar algo más problemático dado que esta sujeto a pérdida de precisión si se paraleliza por el tipo de datos que usa. Se han contemplado dos alternativas para su paralelización.

La primera utilzar una directiva `#pragma omp for` y una variable privada temporal `tmpSum` que contiene la suma parcial para cada hilo, sumas parciales que luego son añadidas a la total, `sum`, dentro de una sección crítica. A continuación puede verse un ejemplo de esa solución.

```
m_type sum = 0;
  m_type tmpSum = 0;
  int i,j;
  #pragma omp parallel default(none)\
  shared(rows,cols,sum,matrix)\
  private(tmpSum,j)
  {
  tmpSum = 0;
  #pragma omp for
  for(i=1; i<rows+1; i++)
          for(j=1; j<cols+1; j++)
                  tmpSum += m(i,j);

  #pragma omp critical
  sum+=tmpSum;
  }
```

La segunda es utilizar la directiva `#pragma omp for` con la cláusula `reduction(sum:+)`, lo cual en realidad es equivalente.

Aunque con ésto se consigue cierto grado de mejora en la eficiciencia, se ha decidido que la pérdida de precisión no compensa la ganancia de eficiencia.

2. Tiempos de respuesta

Los tiempos de ejecución para el código secuencial para cada una de las matrices de entrada son los que se especifican en la siguiente tabla.

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	0,005s	0,356667s	1,106333s	29,473333s
Bucle de cómputo	0,001410s	0,317187s	1,081001s	29,027571s

Cuadro 1: Tiempos de ejecución del código secuencial

Los tiempos de respuesta del código paralelo con un sólo hilo para los mismos casos de entrada se resumen en la siguiente tabla junto a sus *speedups*.

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	0,006s	0,526s	1,606333s	42,645333s
Bucle de cómputo	0,001958s	0,487306s	1,578907s	42,200928s
Speedup	0,833333	0,678074	0,688732	0,6911268

Cuadro 2: Tiempos de ejecución del código paralelo para un hilo

Lo que se puede observar es que el *speedup* es menor que cero, es decir, que en las ejecuciones del código paralelo con un sólo hilo son más lentas que las de de ejecuciones del código secuencial. La razón está en que hay una ligera carga de debido al uso de OpenMP.

A continuación se muestra el resultado del mismo código para dos hilos de ejecución.

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	0,006333s	0,294667s	0,855667s	21,846333s
Bucle de cómputo	0,002280s	0,256489s	0,827608s	21,394403s
Speedup	0,789515	1,210407	1,292948	1,3491204

Cuadro 3: Tiempos de ejecución del código paralelo para dos hilos

Como puede verse se ha conseguido una mejora de rendimiento con la añadidura de un hilo más. Con dos hilos los costes de planificación aun no son muy grandes y ya se puede apreciar mejora a la hora de hacer cálculos en paralelo, como demuestran las tablas.

Para el caso de la matriz 15x20 puede apreciarse una bajada de rendimiento, lo que demuestra que para ese número de datos e hilos los costes de planificación no compensan. Las sobrecargas de planificación sobrepasan la mejora conseguida con la ejecución paralela.

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	0,006s	0,237s	0.657s	17,22s
Bucle de cómputo	0,001901s	0,198484s	0,630934s	16,772716s
Speedup	0,833333	1,504924	1,683916	1,711576

Cuadro 4: Tiempos de ejecución del código paralelo para cautro hilos

En la siguiente tablas se muestra el resultado de añadir un máximo de cuatro hilos.

Como puede verse, salvo para el caso de la matriz más pequeña, todas las demás han conseguido un *speedup* mayor que 1, luego la paralelización merece la pena para el resto de casos de entrada. En el caso de la matriz 15x20, el coste de planificación sobrepasa la ganancia de la paralelización. Como puede apreciarse, mientras más grandes sean las matrices, mejor se comporta la paralelización.

3. Optimización de la copia de matrices

Como pudo verse en los primeros apartados la copia de matrices que se hacía en el código original implicaba copiar elemento a elemento la matriz de escritura en la matriz de lectura. Ésto se puede solucionar inmediatamente y, de hecho, se ha hecho, intercambiando sencillamente los punteros de las matrices.

4. Conclusiones y pruebas descartadas

Por último, merece la pena hacer mención a algunas opciones que se han tenido en cuenta, pero finalmente no se han podido incluir y explicar por qué no se llegaron a implementar.

Inicialmente al estudiar el problema se encontró un problema en la implementación que se presentaba: se está trabajando usando extensivamente la memoria RAM. Como las matrices de ejemplo que se presentan son pequeñas comparadas con la RAM que se tiene en las maquinas actuales este problema podría pasarse por alto.

Pero si se quiere hacer uso de este programa en una aplicacion real, donde puedan ser necesario manejar varios gigas de datos, tantos que la memoria RAM no fuera capaz de conterlos, el código que tenemos sería muy ineficiente.

Ésto es algo que pasa incluso en programas ampliamente utilizados en la comunidad cientifíca e incluso en el ámbito de empresas. Baste de ejemplo el campo de la estadística donde se trabajan con enormes cantidades de datos.

Este problema aparece en el excelente programa Gnu R para cálculos estadísticos.El programa es sumamente rápido con los cálculos, ya que carga todo en la RAM pero por contra tiene este problema cuando los datos a usar son de gran magnitud. Lo han subsanado con paquetes externos que hacen uso de *streaming* desde el disco duro.

Como simple anécdota cabe mencionar la diferencia que tiene otro gran ejemplo de este campo: SAS. Programa que hace un excelente uso de recursos trabajando con ingentes cantidades de datos de un orden muy superior al tamaño de la RAM, haciendo uso del disco duro.

Es algo más lento en ciertos cálculos, pero es ampliamente usado en el mundo empresarial para mover datos de enormes bases de datos.

Una solución que se usa en la práctica, por costes de licencias, además de eficiencia, es usar lo mejor de ambos.

Cerrando este inciso y para continuar con el problema propuesto cabe comentar que se hizo una implementación en el propio lenguaje R, ya que se trabaja de la misma manera que con el algoritmo problema y se trata de un lenguaje vectorial, donde se comprobó la velocidad de este lenguaje con matrices de este tamaño. Pese a ser implementado para ver si se podía encontrar en sus funciones especiales, tappy en concreto, alguna idea para nuestro problema no se consiguió ningún resultado.

Para subsanar ésto se consideró la idea de particionar la matriz en submatrices más pequeñas, para unirlas al final. Ésto además añade el extra de una paralelización mucho mayor. Pero en el caso de usar memoria compartida, con la tecnología actual donde el número de núcleos es muy limitado sólo se añadiría una sobrecarga al crear y gestionar los numerosos hilos.

Ésto será muy útil para los otros dos modelos de programación paralela que se atacarán más adelante, sobre todo en el que haga uso de una GP-GPI donde existe una enorme cantidad de unidades de proceso.

Surgen así nuevos problemas como la sincronización, ver el número de particiones, etc; pero ya nos enfrentaremos a ello cuando llegue el momento.

Cabe mencionar al respecto que también se trató de ver cuál era el número óptimo de hilos según los nućleos que tuviera la máquina y el temaño del conjuto de datos. Éste es un problema complejo del ámbito de la investigación operativa denominado problema de asignación. Una perspectiva que se usó para enfocarlo fue la programación dinámica, ya que se cumplen las características que necesita como el principo de optimidad de Belman para ver hallar la solución óptima del número óptimo para el tamaño de la matriz a usar.

Sería una función recursiva para minimizar usando como restricciones el tiempo de crear/gestionar los hilos.

Como en la práctica se pide el cálculo para cuatro matrices de dimensiones fijas y no es un problema de fácil solución, a parte de que OpenMP no permite elegir a que hilo mandar la tarea, se comprobó empíricamente en vez de analíticamente.

También es algo que podría ser útil en los dos métodos de programación o una mezcla entre ellos. Pero al ser máquinas iguales lo más probable es que el reparto sea proporcional.

Para finalizar, otra opción que se ha tenido en cuenta, pero que a la que finalmente no se ha encontrado una forma óptima de implementarla fue el incluir la suma de comprobación en el bucle principal, para ahorrarnos el tener que recorrer de nuevo el bucle. Los dos problemas que se nos presentaron fueron que

al añadirlo dentro de la función *update*, se calculaba la suma todas las veces que entrara en el bucle, lo cual añadía una carga innecesaria.

La solución que se encontró es intentar que lo hiciera sólo en la última iteración. Como la suma es asociativa, el orden en que se sume es indiferente, por lo que con definir la variable suma como pública y protegiendo su escritura bastaría.

El problema a esta solución es encontrar cuál es la última iteración, entre otras cosas por la pérdida de precisión que vamos arrastrando al paralelizar. Por lo tanto no se ha incluido finalmente en el codigo.