Scan Sky: OpenMP

García Prado, Sergio Calvo Rojo, Adrián

11 de marzo de 2017

Resumen

En este documento se relatan el conjunto de mejoras aplicadas a un código base para obtener mejoras a nivel de rendimiento utilizando como medida el tiempo de ejecución. El modelo de paralelización utilizado para dicha tarea ha sido memoria compartida*, para lo cual se ha utilizado el framework* OpenMp*[1] en su versión para el lenguaje de programación* C*. El código fuente obtenido tras dichas optimizaciones se podrá consultar a través de* `https://github.com/garciparedes/parallel-scan-sky/blob/master/openmp/src/ScanSky_openmp.c` *[3]*

I. Introducción

En este documento se exponen el conjunto de mejoras realizadas sobre "un código secuencial para contar el número de objetos diferentes que se ven en una imagen o fotografía celeste, en general de espacio profundo, obtenida por un radiotelescopio"[2]

"Las imágenes ya han sido procesadas, discretizando la luminosidad observada en cada punto en 16 diferentes niveles de grises o colores. Los pixels del fondo del espacio, una vez eliminado el ruido, tienen el índice de color 0. Los pixels de la imagen con una luminosidad o color suficientemente parecidos se representan con un mismo valor entre 1 y 15."[2]

"La imagen se carga en una matriz desde un fichero de texto plano. El fichero contiene un número entero en cada línea. Las dos primeras líneas contienen el número de filas y columnas de la imagen. El resto son números entre 0 y 15 con los valores de cada pixel, ordenados por filas."[2]

"Los pixels del mismo índice de color que están juntos, en horizontal o vertical (no en diagonal), se considera que son del mismo objeto. El programa etiqueta cada objeto de la imagen con una etiqueta diferente. Todos los pixels del mismo objeto tendrán la misma etiqueta. Para determinar el número de objetos, al final se cuentan el número de etiquetas diferentes. Los píxeles de índice 0 no se etiquetan."[2]

II. Optimización

A continuación se realiza una breve descripción acerca del conjunto de mejoras realizadas sobre el código secuencial base utilizado para la práctica. El *framework* utilizado para llevar a cabo dichas técnicas de optimización paralela es **OpenMP**[1]. Este modelo se basa en la idea de **memoria compartida**, es decir, el espacio de direcciones es común para todos los hilos que se ejecuten de manera concurrente. Sin embargo, también se permite la reserva de zonas de memoria privada para cada elemento de proceso, de una forma similar a y como sucede en el caso de llamadas a funciones en las cuales se definen variables.

Lo primer a destacar en cuanto a las modificaciones realizadas en la implementación original es la definición de la región paralela, la cual se puede apreciar de manera resumida en la figura 1. Esta región del código abarca de comienzo a fin el conjunto de lineas de código cuyo tiempo de ejecución es registrado. En esta parte se puede apreciar que se ha decidido deshabilitar la

compartición por defecto de variables compartidas, lo cual ofrece un mayor control a la vez que una casi inapreciable mejora de eficiencia en el proceso de inicialización de la región paralela.

```
#pragma omp parallel \
default(none), \
shared(k_indexer, k_max, matrixData, matrixResult, \
    matrixResultCopy,columns,rows, flagCambio,numBlocks)
{
    // ...
}
```

Figura 1: *Inicialización de la región paralela*

Nótese que además de las variables existente anteriormente, se ha añadido una nueva estructura de datos llamada *k_ indexer*, junto con un contador *k_ max* que determina el número de elementos almacenados en la estructura. La utilidad de la misma es almacenar todas aquellas posiciones no nulas de la imagen de entrada, lo cual aporta grandes ventajas reduciendo en gran medida el número de comprobaciones innecesarias sobre dicha matriz.

En la figura 2 se muestra el código modificado para la labor de inicializar los valores de la matriz de resultados (*matrixResult*), la cual será utilizada durante el resto del programa. Puesto que dicha tarea presenta una naturaleza muy paralela, se ha decidido que su ejecución sea realizada de dicha manera. La carga no está balanceada debido tanto a los diferentes casos que pueden sucederse según el contenido de cada casilla, como a la necesidad de mantener el número de elementos almacenados en la estructura auxiliar *k_ indexer*, que requiere que su contador de elementos se incremente en una zona de exclusión mutua. Por estas razones se cree que lo más conveniente es una planificación **dinámica** que a largo plazo consiga una estabilización en cuanto a la carga de trabajo asignada a cada hilo. Nótese que el bucle de rellenado de los bordes de las columnas (*j's*) se ha extraido fuera del principal, lo que evita escrituras innecesarias. Debido a que no existen problemas de dependencias entre estos bucles, se ha eliminado la barrera de salida de los mismos a través de la directiva *nowait*.

Lo siguiente es un bucle **do-while** representado a través de un **for** en el código base, pero reescrito de dicha manera para mejorar la comprensión. Para conservar la semántica de ejecución de dicha directiva debido al paralelismo ha sido necesario añadir una barrera (*barrier*) explicita al comienzo de la misma. Esto se ilustra de manera resumida en la figura 3.

```
/* 4. Computacion */
do {
    #pragma omp barrier

    #pragma omp single
    flagCambio=0;

    //...

} while(flagCambio !=0);
```

Figura 3: *Bucle Principal de Cómputo*

El contenido del bucle de la figura 3 se compone de dos partes bien diferenciadas, la primera de ellas destinada a actualizar una matriz de copia *matrixResultCopy* sobre la matriz de resultados *matrixResult* anteriormente citada. La segunda parte se corresponde con el algoritmo utilizado para detectar figuras distintas en la matriz.

En la figura 4 se muestra la tarea de copia de la matriz de resultados, la cual se reduce a un simple cambio de referencias entre la matriz original y la de copia. Utilizando esta alternativa

```
/* 3. Etiquetado inicial */
#pragma omp for \
nowait,\
schedule(dynamic, ((rows-1)*(columns-1))/omp_get_num_threads()), \
private(i,j,k)
for(i = 1; i < rows-1; i++){
    for(j = 1; j < columns-1; j++){
        // Si es 0 se trata del fondo y no lo computamos
        if(matrixData[i*(columns)+j]!=0){
            matrixResult[i*(columns)+j]=i*(columns)+j;
            #pragma omp atomic capture
            {
                k = k_max; k_max++;
            }
            k_indexer[k] = i*(columns)+j;
        } else {
            matrixResult[i*(columns)+j]=-1;
        }
    }
    matrixResult[i*(columns)]=-1;
    matrixResult[i*(columns)+columns-1]=-1;
}

#pragma omp for \
nowait,\
schedule(static), \
private(j)
for(j=1;j< columns-1; j++){
    matrixResult[j]=-1;
    matrixResult[(rows-1)*(columns)+j]=-1;
}
```

Figura 2: *Rellenado de la matriz de resultados, sobre la cual se desarrollará el cómputo.*

frente a la iterativa basada en asignar el mismo valor de cada posición de una matriz en otra se consigue una gran mejora de eficiencia.

```
/* 4.2.1 Actualizacion copia */
#pragma omp single
{
    flagCambio=0;
    temp = matrixResultCopy;
    matrixResultCopy = matrixResult;
    matrixResult = temp;
}
```

Figura 4: *Copia de la Matriz de Resultados*

La región de código de la figura 5 muestra el resultado final tras varias optimizaciones sobre la misma. Nótese que se ha aplicado la técnica de *inlining* eliminando la llamada a la función *computacion()*, lo cual ha permitido una gran redución en cuanto al número de operaciones aritméticas para el cálculo de las posiciones en las que se debe comprobar casillas contiguas. En la figura 5 tan solo se muestra uno de los casos por la semejanza con el resto. También se ha restringido el conjunto de valores que tomaba *flagCambio* a una variable de tipo binario, lo cual elimina el coste de incrementar su valor. Por último, se ha comprobado que en esta sección de código se obtienen mejores resultados cuando la planificación del mismo es **estática** a pesar del gran número de condicionales y por consiguiente de desbalanceos de carga que se esperaría que sucedieran. La conclusión a la cual se ha llegado ha sido que debido al reducido coste de cada condición, otras estrategias más sofisticadas no han llegado a mostrar sus ventajas.

```
/* 4.2.2 Computo y detecto si ha habido cambios */
#pragma omp for \
schedule(static), \
reduction(||:flagCambio),\
private(k)
for(k=0;k<k_max;k++){
    if((matrixData[k_indexer[k]-columns] == matrixData[k_indexer[k]]) &&
        (matrixResult[k_indexer[k]] > matrixResultCopy[k_indexer[k]-columns]))
    {
        matrixResult[k_indexer[k]] = matrixResultCopy[k_indexer[k]-columns];
        flagCambio = 1;
    }
    // ...
}
```

Figura 5: *Cómputo necesario para reconocer figuras diferentes en la matriz*

La última sección del código, una vez que la matriz de resultados se encuentra estabilizada, no habiéndose producido modificaciones durante la anterior fase, es la de realizar el conteo del número de figuras distintas que contiene la matriz. Esto se ilustra en la figura 6 Para ello se ha utilizado un bucle **for**, tal y como en el código base. Debido a la semejanza operacional de todas sus iteraciones, este se ha paralelizado teniendo una estrategia **estática**, la cual se cree que funciona mejor que otras por la misma razón que en el caso de la figura 5.

```
/* 4.3 Inicio cuenta del numero de bloques */
#pragma omp for \
schedule(dynamic, k_max/omp_get_num_threads()), \
private(k),\
reduction(+:numBlocks)
for(k=0;k<k_max;k++){
    if(matrixResult[k_indexer[k]] == k_indexer[k]) numBlocks++;
}
```

Figura 6: *Conteo del número de figuras diferentes*

III. Conclusiones

La adaptación de código secuencial a paralelo puede ser una tarea compleja en un gran número de situaciones debido a la dificultad para conocer de manera clara y concisa cuáles son las zonas del código con mayor carga computacional, así como las dependencias que en la versión secuencial no surgen.

El modelo que propone *OpenMP*[1] para el desarrollo de aplicaciones que utilicen paralelismo es una manera sencilla de obtener grandes ventajas a nivel de rendimiento, ya que en la mayoría de casos ofrece un elevado nivel de transparencia respecto de tareas de sincronización o interconexión de hilos.

Referencias

[1] OpenMP. `http://www.openmp.org`.

[2] Computación Paralela, 2016/17.

[3] García Prado, S., and Calvo Rojo, A. Parallel Scan Sky. `https://github.com/garciparedes/parallel-scan-sky`.