Scan Sky: MPI

García Prado, Sergio
sergio@garciparedes.me

Calvo Rojo, Adrián
adrian.calvo.rojo@alumnos.uva.es

31 de marzo de 2017

Resumen

En este documento se relatan el conjunto de mejoras aplicadas a un código secuencial base. El modelo de paralelización utilizado para dicha tarea ha sido memoria distribuida*, para lo cual se ha utilizado el framework* MPICH*[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/mpi/src/ScanSky_mpi.c*** *[2]*

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"[3]

"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."[3]

"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."[3]

"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."[3]

II. Optimización

Puesto que las optimizaciones a nivel de paralelización se han llevado a cabo a partir de la especificación *MPI*, la primera tarea es realizar las tareas de inicialización tal y como se indica en dicha especificación. Esto se muestra en la figura 1.

```
MPI_Request *request = (MPI_Request *)malloc( 5 * sizeof(MPI_Request) );
MPI_Datatype row_type;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size (MPI_COMM_WORLD, &world_size);
int world_right = (world_rank +1 ) % world_size;
int world_left = (world_rank -1 + world_size) % world_size;
```

Figura 1: *Inicialización del entorno MPI*

La alternativa escogida como modelo de paralelización ha sido la división de la matriz en bloques equiespaciados dependientes a nivel de filas. Por lo tanto (y debido a la restricción impuesta a priori que restringe la lectura de datos a un único proceso), el primer paso es transferir el número de filas y columnas que contiene el fichero de entrada al resto de procesos, para después calcular el desplazamiento correspondiente para dicho proceso (mismo para todos excepto para el último en los casos en que corresponda). Además se define el *tipo de datos contiguo* que será utilizado posteriormente para la comunicación entre procesos. Esto se muestra en la figura 2.

```
MPI_Ibcast(rows_columns, 2, MPI_INT, 0, MPI_COMM_WORLD, &request[0]);
if (world_rank != 0){
    MPI_Wait(&request[0], MPI_STATUS_IGNORE);
}
MPI_Type_contiguous( rows_columns[1]-2, MPI_INT, &row_type );
MPI_Type_commit(&row_type);
row_shift = (rows_columns[0])/world_size;
row_init = 1;
row_end = row_shift +1;
if(world_rank == world_size-1){
    row_end = (rows_columns[0]-1)- (row_shift*(world_size-1));
}
```

Figura 2: *Transferencia del número de filas y columnas, cálculo del desplazamiento correspondiente e incialización de tipo de datos contiguo*

El siguiente paso es la transferencia de la matriz de entrada a cada proceso según corresponda. Se ha preferido una implementación manual de dicha tarea por las siguientes razones: *a*) la ineficiencia derivada de transmitir la matriz completa a cada uno de los procesos y *b*) la necesidad de solapamiento entre las particiones. Por lo tanto, no es posible utilizar la función *scatter()* e ineficiente utilizar la función *broadcast()*. Por contra, no se aprovechan las ventajas de transmisión jerárquica implementadas por *MPICH*, pero debido al relativamente reducido tamaño del cluster en que se ejecutará dicha penalización es asumible. Esto se muestra en la figura 3.

```
if (world_rank == 0){
    for(i = 1; i < world_size-1; i++ ){
        MPI_Isend(&matrixData[(row_shift*i)*(rows_columns[1])],
            (row_shift + 2)*rows_columns[1],
            MPI_INT, i, 0, MPI_COMM_WORLD, &request[0]);
    }
    MPI_Isend(&matrixData[(row_shift*(world_size-1))*(rows_columns[1])],
        ((rows_columns[0]) - (row_shift*(world_size-1)))*rows_columns[1],
        MPI_INT, world_size-1, 0, MPI_COMM_WORLD, &request[0]);
} else {
    MPI_Irecv(&matrixData[(row_init-1)*(rows_columns[1])], (2 + row_end - row_init)*rows_columns[1],
        MPI_INT, 0, 0, MPI_COMM_WORLD, &request[4]);
    MPI_Recv_init(&matrixResultCopy[(row_init-1)*rows_columns[1]+1], 1,
        row_type, world_left, 0, MPI_COMM_WORLD, &request[1]);
    MPI_Send_init(&matrixResultCopy[(row_init)*rows_columns[1]+1], 1,
        row_type, world_left, 0, MPI_COMM_WORLD, &request[3]);
    MPI_Start(&request[1]);
    MPI_Wait(&request[4], MPI_STATUS_IGNORE);
}
if (world_rank != world_size - 1) {
    MPI_Recv_init(&matrixResultCopy[(row_end)*rows_columns[1]+1], 1,
        row_type, world_right, 0, MPI_COMM_WORLD, &request[0]);
    MPI_Send_init(&matrixResultCopy[(row_end-1)*rows_columns[1]+1], 1,
        row_type, world_right, 0, MPI_COMM_WORLD, &request[2]);
    MPI_Start(&request[0]);
}
```

Figura 3: *Transmisión de la matriz de datos e inicialización de conexiones permanentes*

En el código de la figura 3, se muestra también la inicialización de conexiones permanentes (realizadas de manera no bloqueante para tratar de aprovechar al máximo posible las capacidades

de cómputo de cada proceso), que serán utilizadas en partes posteriores para transmitir la primera y última filas al proceso anterior y siguiente en el rango. También se deja abierta la recepción para no sufrir retrasos por dichas razones.

En la figura 4 se muestra el código utilizado para la recepción de filas de los procesos contiguos. No hay problemas de pérdida de mensajes debido a nuevas recepciones por las razones que se expondrán en parrafos subsiguientes. Además, se ha añadido un bucle de comprobación de cambios, utilizado para obviar el cómputo cuando no se hayan producido cambios en la matriz de resultado durante la fase anterior, ni en los bordes recibidos (esto se puede comprender fácilmente mediante la lectura del código completo).

```
if (world_size != 1 && t != 0){
    if (world_rank == world_size -1 ) {
        MPI_Wait(&request[1], MPI_STATUS_IGNORE);
        MPI_Start(&request[1]);
        if (!local_flagCambio){
            // ...
        }
    } else if (world_rank == 0 ) {
        MPI_Wait(&request[0], MPI_STATUS_IGNORE);
        MPI_Start(&request[0]);
        if (!local_flagCambio){
            //...
        }
    } else {
        MPI_Waitall(2, request, MPI_STATUS_IGNORE);
        MPI_Startall(2,request);
        if (!local_flagCambio){
            for(j = 1; j < rows_columns[1]-1;j++){
                if (matrixResult[(row_init-1)*rows_columns[1]+j]!=
                    matrixResultCopy[(row_init-1)*rows_columns[1]+j]) {
                    local_flagCambio = 1;
                    break;
                }
                if (matrixResult[(row_end)*rows_columns[1]+j]!=
                    matrixResultCopy[(row_end)*rows_columns[1]+j]) {
                    local_flagCambio = 1;
                    break;
                }
            }
        }
    }
}
```

Figura 4: *Recepción de las filas límite de la zona de computo dedicada a cada proceso*

Una vez realizadas las operaciones pertinente, es necesario transferir los resultados de los bordes a los procesos contiguos. Esto se muestra en la figura 5. Además se realiza la compartición del *flagCambio* entre todos los procesos mediante la función *allReduce()* de manera no bloqueante. Se espera a los resultados de dicha operación solo en el caso en que la durante la iteración correspondiente no se hayan hecho modificaciones en la matriz de resultados. La razón por la cual no surgen conflictos derivados de la recepción de filas viene propiciada por el punto de envio de los mismos, que se hace una vez que ha finalizado la compartición de *flagCambio*. Por tanto, dicha instrucción actua también como barrera de sincronización entre procesos.

Una vez completado el bucle de propagación de celdas con la misma figura, el siguiente paso es realizar el conteo de las figuras que contienen cada una de las partes de la matriz, el último paso es transferir los resultados parciales a través de la función *reduce()* al proceso inicial encargado de devolver los resultados una vez finalizado el programa. Dicha labor se muestra en la figura 6.

```
if (world_size != 1) {
    if(t !=  0){
        MPI_Wait(&request[4], MPI_STATUS_IGNORE);
    }
    if (world_rank != world_size - 1) {
        MPI_Wait(&request[2], MPI_STATUS_IGNORE);
        MPI_Start(&request[2]);
    }
    if (world_rank != 0) {
        MPI_Wait(&request[3], MPI_STATUS_IGNORE);
        MPI_Start(&request[3]);
    }
    flagCambio=0;
    MPI_Iallreduce(&local_flagCambio, &flagCambio, 1, MPI_CHAR, MPI_LOR,
        MPI_COMM_WORLD, &request[4]);
    if (!local_flagCambio){
        MPI_Wait(&request[4], MPI_STATUS_IGNORE);
    }
}
```

Figura 5: *Envío de las filas límite de la zona de cómputo dedicada a cada proceso y sincronización de* flagCambio

```
local_numBlocks = 0;
for(i=row_init;i<row_end;i++){
 const int t1 = (i+row_shift*world_rank)*rows_columns[1];
 for(j=1;j<rows_columns[1]-1;j++){
  if(matrixResult[i*rows_columns[1]+j] == t1+j) local_numBlocks++;
 }
}
MPI_Reduce(&local_numBlocks, &numBlocks, 1, MPI_INT, MPI_SUM, 0,
   MPI_COMM_WORLD);
```

Figura 6: *[TODO]*

En esta descripción se han obviado las modificaciones no referidas a optimizaciones a nivel de paralelización por razones de simplicidad. Nótese que se ha tratado de reducir al máximo el espacio necesario para las tareas de cómputo, reservando en cada caso el mínimo tamaño de memoria posible.

III. Conclusiones y Propuestas Futuras

La adaptación de código secuencial a código paralelo a nivel de memoria distribuida no es una tarea trivial, ya que además del nivel de dificultad derivado con tareas de sincronización, se añade el derivado de la necesidad de comunicación explicita entre procesos, algo que en el modelo de memoria compartida con hilos no sucede.

Debido a las restricciones temporales para realizar el trabajo que se ha descrito en este documento, no ha sido posible implementar técnicas de reequilibrado de carga. A pesar de ello se cree que podría ser interesante llevarlo a cabo para entradas de datos de gran tamaño mediante un patron *job-stealing* entre procesos contiguos, lo cual compensaría los costes de comunicación en gran medida sobre clusters heterogéneos.

Referencias

[1] MPICH High-Performance Portable MPI. `https://www.mpich.org`.

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

[3] González Escribano, A., Moretón Fernández, A., and Rodríguez Gutierrez, E. Computación paralela, 2016/17.