Práctica 2: Computación de tipo stencil: Solución con MPI

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

28 de noviembre de 2013

1. Decisiones de diseño

Antes de empezar se han fijado unas hipótesis iniciales para poder tomar la decisión de cómo abordar la paralelización del código mediante paso de mensajes.

Dada la naturaleza del modelo de programación a usar es fundamental minimizar el número de conexiones entre procesos, evitando sobretodo recargarlos con información redundante pudiendo provocar cuellos de botella. Ésto depende de diversas variables como puedan ser la velocidad de conexión entre procesos, el tamaño de los mensajes enviados, el número de mensajes, etc.

Para evitarlo se ha establecido una serie de restricciones específicas para un tipo concreto de sistema, Piukeman, el portátil de uno de los miembros del equipo (Acer Aspire con con 4 procesadores y 4G RAM).

Para otros casos habría que modificar el código para adaptarlo a las nuevas y diversas configuraciones. Se abordará este aspecto al final de este informe explicando los problemas que se han intentado evitar y proponiendo planes de acción adecuados.

Las mencionadas hipótesis son las que se listan a continuación:

- Se trabaja en un entorno totalmente homogéneo.
- El reparto se hace de forma uniforme.
- El programa se va a ejecutar en una máquina, de recursos conocidos y suficientes para trabajar holgadamente.

Ésto último implica que no habrá problemas como escasez de RAM o de cuellos de botella debido a la velocidad de comunicación entre los procesos siempre y cuando se respeten las restricciones de recursos.

Teniendo en cuenta lo anteriormente expuesto la solución adoptada ha pasado por encargar al proceso raíz, `rank == 0`, la lectura de la matriz desde el fichero, el reparto equitativo de las submatrices y el envío al resto de procesos para que cada uno trabaje independiente. Con esta base los aspectos que cabe destacar dentro de la solución son la forma en la que se lee la matriz, la forma en la que se reparte la matriz, la forma en la que se calculan las sucesivas iteraciones, la forma en la que los procesos comunican la información interdependiente y la forma en que al final ponen en conjunto el trabajo realizado.

1.1. Lectura de la matriz

El código de lectura de la matriz es secuencial y es una tarea de la que se encarga exclusivamente el proceso raíz. Dicha lectura se realiza a una matriz sin bordes, dado que no son necesarios en principio y su creación implicaría envíos más largos y costosos más adelante.

1.2. Reparto de matrices

Puesto que la matriz se define mediante un array unidimensional reinterpretado en una matriz bidimensional mediante el uso de macros, la forma idónea de dividirlo es por filas. De esa forma a cada proceso se enviarán a cada proceso zonas sucesivas de memoria.

Dicha repartición se realiza mediante el uso de el tipo derivado *subarray* para facilitar la semántica del envío de una matriz sin halo y la recepción en una matriz que tan sólo tiene halo superior e inferior. Se ha tomado la decisión de no usar halos laterales, dado que pueden ser referenciados fácilmente mediante aritmética modular. Se puede ver a continuación un ejemplo de ello en un fragmento del código de la función de actualización.

```
 ...

m_new(i,j) = (m_old(i,j) * WEIGHT_CENTER) +
((m_old(i-1,j) +
m_old(i+1,j) +
m_old(i,(j-1+cols)%cols) +
m_old(i,(j+1)%cols)) * WEIGHT_ADJACENT) +
((m_old(i-1,(j-1+cols)%cols) +
m_old(i-1,(j+1)%cols) +
m_old(i+1,(j-1+cols)%cols) +
m_old(i+1,(j+1)%cols)) * WEIGHT_CORNER);

 ...
```

1.3. Comunicación de halos

Una vez que cada proceso ha recibido su submatriz se encargará de comunicarse con los procesos contiguos, el rank anterior y rank posterior, para enviarse los halos superiores e inferiores, respectivamente.

Los envíos de los halos están formados por tres envíos no bloqueantes y

uno bloqueante para evitar esperas innecesarias, tal y como puede verse en el siguiente fragmento.

```
...

MPI_Isend(halos+(cols*SUP_SND),cols,MPI_INT,\
(rank-1+size)%size,tag,MPI_COMM_WORLD,reqs+SUP_SND);
MPI_Isend(halos+(cols*INF_SND),cols,MPI_INT,\
(rank+1)%size,tag,MPI_COMM_WORLD,reqs+INF_SND);
MPI_Irecv(halos+(cols*INF_RCV),cols,MPI_INT,\
(rank+1)%size,tag,MPI_COMM_WORLD,reqs+INF_RCV);
MPI_Recv(halos+(cols*SUP_RCV),cols,MPI_INT,\
(rank-1+size)%size,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

MPI_Wait(reqs+SUP_SND,MPI_STATUS_IGNORE);
MPI_Wait(reqs+INF_SND,MPI_STATUS_IGNORE);
MPI_Wait(reqs+INF_RCV,MPI_STATUS_IGNORE);

...
```

1.4. Cómputo de iteraciones sucesivas

A partir de que todos los procesos tienen sus fragmentos y halos correspondientes realizarán las mismas tareas. Cada proceso hará el cálculo de la siguiente iteración de forma independiente comunicando con el resto de procesos los halos actualizados, de la forma ya vista, y sus residuos parciales al fin de cada iteración.

La salida del bucle de cómputo viene dada por la condición de que el residuo máximo de cada proceso sea menor que el establecido por el problema. Para hallar ese residuo máximo se utiliza la función `MPI_Allreduce()` con la operación `MPI_MAX` que equivale a una función `MPI_Reduce` y a una función `MPI_Bcast`, que se supone optimizada para la implentación de MPI.

1.5. Cómputo de suma de comprobación

Una vez terminado el bucle principal de cómputo cada proceso hará la suma de comprobación de su parte y se sumará a la del resto de procesos con una función `MPI_Allreduce()` con la operación `MPI_SUM`.

1.6. Cómputo de tiempo de ejecución

El tiempo se ha medido tomando las mismas referencias que en el código secuencial estableciendo una reducción de máximo de la misma manera que en el cálculo del residuo.

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	0,358s	1,116s	29,689s
Bucle de cómputo	0,001422s	0,316709s	1,091294s	29,507141s

Cuadro 1: Tiempos de ejecución del código secuencial

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

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	0,016s	0,637s	1,912s	51,199s
Bucle de cómputo	0,002627s	0,573626s	1,866324s	50,566205s
Speedup	0,3125	0,562009	0,583682	0,579875

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

Como puede verse la sobrecarga de repartición y copias de matrices propia del programa en paralelo penaliza gravemente los tiempos ejecución si sólo se usa un procesador. Como con OpenMP el caso que peor resultado da es del la matriz más pequeña, lo que demuestra que soluciones de paralelización no son óptimas para conjuntos de datos muy pequeños.

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

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	0,019s	0,350s	1,004s	27,170s
Bucle de cómputo	0,002407s	0,295530s	0,959196s	25,952534s
Speedup	0,263158	1,022857	1,111554	1,092713

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

Se puede apreciar una mejora en todos los conjuntos de entradas salvo en el primero, que incluso ha disminuido su rendimiento. La mejora es cercana al doble de velocidad con respecto al mismo código y el doble de procesos y ligeramente más rápida que la ejecución del código secuencial.

Sin embargo el rendimiento de MPI es, hasta ahora, menor que el de de OpenMP si los comparamos tomando mismo número de hilos con mismo número de procesos. Ésto se puede explicar dado que el manejar multiprocesamiento en vez de multihenebrado es necesario hacer cambios de contexto y manejar la memoria de forma distribuida.

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

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	0,026s	0,303s	0,887s	23,597s
Bucle de cómputo	0,002929s	0,212857s	0,826619s	22,031397s
Speedup	0,192308	1,181518	1,258174	1,131034

Cuadro 4: Tiempos de ejecución del código paralelo para cuatro procesos

Se aprecia una creciente mejora de rendimiento en todos los conjuntos de datos salvo en el primero y más pequeño, que sigue degradándose. Ésto ocurría con OpenMP y permanece constante, con la salvedad de que la velocidad de degradación es mayor esta vez.

Los tiempos de respuesta del código paralelo con cinco proceso para los mismos casos de entrada se resumen en la siguiente tabla junto con sus *speedups*. Este caso es especial dado que se usa un proceso más de los que procesadores se dispone, eso implicará carga en planificación.

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	2,708s	4,403s	2,235s	40,654s
Bucle de cómputo	2,615897s	4,255337s	2,024210s	39,505586s
Speedup	0,001846	0,081308	0,499329	0,730285

Cuadro 5: Tiempos de ejecución del código paralelo para cuatro procesos

Como puede verse el rendimiento ha caído en picado añadiendo un proceso más de los que procesadores cuenta la máquina. A pesar de que ésto era algo de esperar sorprende el caso de las matriz $1000x1000$ que tiene un tiempo de respuesta de casi la mitad del que tiene la matrix $300x400$. El incidente merece un estudio en mayor profundidad, pero se cree que puede deberse a que la matriz de mayor tamaño es cuadrada y ofrece un mejor comportamiento para el algoritmo elegido.

Para cinco procesos la división de las matrices queda la siguiente forma.

- $15x20$**:** Cinco matrices $3x20$ que realmente son de $5x20$ al incluir los halos superiores e inferiores. Lo que supone una gran duplicación de datos.
- $300x400$**:** Cinco matrices $60x400$ que realmente son de $62x400$ al incluir los halos superiores e inferiores.
- $1000x1000$**:** Cinco matrices $200x1000$ que realmente son de $202x1000$ al incluir los halos superiores e inferiores.
- $5000x5000$**:** Cinco matrices $1000x5000$ que realmente son de $1002x400$ al incluir los halos superiores e inferiores.

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

Matrices	**15x20**	**300x400**	**1000x1000**	**5000x5000**
Programa completo	5,873s	9,032s	4,207s	29,553s
Bucle de cómputo	5,723984s	8,759895s	4,701912s	27,839923s
Speedup	0,001846	0,081308	0,499329	0,730285

Cuadro 6: Tiempos de ejecución del código paralelo para cuatro procesos

En este caso se puede comentar lo mismo que en el caso anterior, pero destacando el caso especial de la matriz más grande que ha demostrado un mejor resultado que el caso anterior. Se supone que puede deberse a que es más fácil planificar con un número de procesos que sea múltiplo de el número de procesadores.

3. Optimizaciónes propuestas y conclusiones

3.1. Intercambio de matrices

Igual que en en la práctica anterior se propone y, de hecho se ha implementado, realizar el itercambio de las matrices de lectura y escritura haciendo un intercambio de punteros y no una asignación elemento a elemento. Sin embargo esta optimización ya se considera trivial.

3.2. Problema de asignación

Este ejercicio problema es muy dependiente de la máquina o sistema de máquinas a utilizar, por lo que el aprovechamiento adecuado de los recursos disponibles puede volverse un problema extremamente complejo.

Los principales variables para tener en cuenta son la potencia de cómputo de cada procesador o sistema, el ancho de banda entre la comunicación de los procesos y el tamaño de los datos.

Al haber fijado la restricción de que la máquina sea homogénea hemos simplificado al máximo el problema de asignación de tareas, éstas se reparten equitativamente.

Con entornos heterogéneos no sería tan simple, volviéndose la asignación de tareas un problema fundamental que crecería segun aumente la heterogeneidad. Para afrontar un caso así habría que analizar previamente el sistema y tener en cuenta sus limitaciones. Incluso pudiera darse el caso de cosas a priori contradictorias, como por ejemplo la necesidad de descartar un sistema de cómputo más lento para ganar eficiencia.

3.3. Comunicación

La segunda parte de la problemática es la que surge de las comunicaciones entre los diferentes procesos y su ancho de banda. Para evitar este problema se ha de intentar minimizar el número y tamaño de los mensajes, mandando los mensajes mas grandes inicialmente y luego se reduciendo al mínimo su tamaño y cantidad.

Para explicar esta situación, se podría pensar en un caso límte, sistemas de gran granularidad como un *grid* donde la carga de comunicación debe ser mínima.

Ya que el escenario elegido consta de una sola maquina con varios nucleos este problema no debiera, en principio, ser un problema importante. Aún así

podrían haber aparececido problemas si se hubiera excedido en el número y tamaño de los mensajes.

Cabe añadir que se podría haber optimizado más la repartición de submatrices usando una correcta lógica de punteros en vez de un bucle que se encargue de mcrear una nueva submatriz para el rank 0.

3.4. Tamaño de los datos

Para el final se ha dejado el tema del tamaño de los datos. Aquí la problemática es similar a la presente en la práctica de OpenMP. Estamos trabajando con matrices relativamente pequeñas.

Si usara el código para procesar matrices de dimensiones realmente grandes, podrían surgir varios problemas, de los cuales se comentarán algunos los más importantes y una posible solución para ellos.

Si la RAM es capaz de albergar todo el tamaño de la matriz multiplicado por dos, pero no la sobrecarga de la que hace uso MPI, se podríamos optar por híbrido e incluir OpenMP en la solución. Realmente este es un problema no muy común, pero interesante de comentar, dado que el código paralelizado con MPI usa considerable más cantidad de memoria que el mismo código paralelizado con OpenMP y esta crece a medida que crece el tamaño de la matriz.

Para solventar ésto hay que considerar varios factores, uno de ellos es si los datos a procesar están en todas las máquinas que van a usarse o sólo en una.

Como la matriz va a dividire, en vez de hacer que un proceso la cargue entera y envíe a los diferentes procesadores las submatrices que les corresponden, puede cargarse en cada máquina sólo la parte que se necesite. Evitando de esta manera mandar grandes cantidades de datos y comuinicando sólo los halos cuando sea necesario.

Otro factor a tener en cuenta es la capacidad de MPI de realizar funciones IO de forma paralela, con MPI_IO. No sólo reduciría el tiempo de lectura, sino que evitaría las cargas de comunicación dado que ya no sería necesario repartir los trozos de matriz entre todos los proceso.