Guión del grupo T20-Clase 18-05-2020

Consistencia y Problema 9

Repasamos un poco el concepto de consistencia y vimos el problema 9 del barrier que no era correcto y la forma en que podíamos solucionarlo.

Slide 54 - Consistencia

Bueno, en este tema hemos visto hasta el momento cómo se garantiza que todos vean los datos guardados en una posición de memoria de forma coherente. También hemos visto como garantizar el orden, a nivel de programa, de las actualizaciones sobre posiciones de memoria. Pero... qué orden ven todos los threads, ejecutándose en diferentes procesadores, de las escrituras realizadas a diferentes posiciones de memoria dentro de los diferentes programas??? Ese orden puede variar dependiendo de cómo el hardware reorganiza el orden de las escrituras o el propio compilador cómo las haya cambiado en el programa para, por ejemplo, optimizar el código.

Estamos hablando de un concepto nuevo: consistencia.

En las slides 55-56 se muestran ejemplos donde el programador esperaría que todo fuera bien pero que dependerá de la consistencia garantizada por el sistema : compilador, hardware, etc. De hecho, en el código de la Slide 56, el compilador seguramente «optimizaría» el código de while(next<=mynext); por un bucle infinito de un salto sobre el mismo ya que no ve actualizaciones de las dos variables... Este caso último se puede arreglar forzando algun tipo de consistencia para alguna variable concreta (omp flush (next)).

Tema 5: Data Decomposition

¿Por qué Data Decomposition?

Una buena razón es lo que vimos con el ejemplo del quiz 3 del tema pasado, donde vimos que la forma de inicializar o trabajar con taskloop o omp for puede afectar al rendimiento de nuestra aplicación. Omp for nos daba un mayor control de acceder a los datos más cercanos si teníamos un sheduling estático.

Hasta el momento hemos visto estrategias de paralelización donde se pensaba más en el código u operaciones a realizar (iteraciones, funciones, etc.) y que se querían repartir para poder ejecutar de forma concurrente/parallela.

Pero en ocaciones lo que tiene más peso, es:

- el volumen de datos es muy grande, o bien
- como vimos con el ejercicio del Quiz, es la forma de colocar los datos ya que puede afectar al rendimiento de las aplicaciones debido a los accesos y/o el protocolo de coherencia necesario

En este caso, determinar qué thread/proceso opera con los datos puede ser importante. De hecho, existe una regla que se llama Owner Compute Rule (**Slide 11**), que indica exactamente eso... que

quien tiene los datos debería ser quien realizara los cálculos con ellos, ya sea de lectura o de escritura.

Cuando hablamos de particionar esto **tiene mucho sentido cuando trabajamos con máquinas con memoria distribuida (Slide 5)**, donde se debería pensar bien donde se hace el alocatamiento de memoria y luego el procesamiento. Esto implicará o no comunicaciones entre los diferentes nodos de la arquitectura con memoria distribuida. En el caso de NUMA no tenemos comunicaciones explícitas, ya que todo se hacen los accesos con loads y stores pero sí que se podría observar accesos innecesarios a otros NUMA nodes.

Por lo tanto, tiene sentido hablar de particionar/distribuir/data decomposition con memoria compartida también? Y tanto! Tiene sentido, y sobretodo con las máquinas NUMAs donde la memoria compartida está físicamente distribuida y se tienen diferentes tiempos de acceso **(Slide 5)**. En este caso no es que tengamos los datos particionados, pero seria bueno saber quien los inicializa y procesa para evitar excesivos overheads y cuellos de botella debido a mantener la coherencia de memoria.

También puede pasar que haya datos que no queramos particionar y en este caso, en una arquitectura con memoria distribuida estaríamos seguramente replicando los datos o cálculos, y en una máquina con memoria compartida significaría que todos pueden acceder.

¿Qué particionamos/distribuimos?

Puede ser que nos interese distribuir/particionar los datos de entrada (de los que leemos datos), los datos de salida (en los que escribimos o actualizamos) o bien ambas, datos de entrada y salida a la vez.

En las transparencias 7-10 se realizan ejemplos considerando una operación que se deber realizar sobre una base de datos. En ocasiones, debido a la replica o el particionamiento, al final se tiene que realizar una reducción de los resultados. La replica de los datos, desde un punto de vista de memoria compartida viene a indicar que los threads pueden y acceden a cualquier posición de la estructura de datos.

Nosotros hemos explicado estos conceptos usando el problema 2 del Tema 5 como base, donde se planteo qué se podría particionar (distributed) o no (en este caso consideramos que todo estaba en un single node). Eso nos ayudará a entender por un lado qué significa «particionar» los datos de entrada o de salida cuando usamos memoria compartida.

¿Cómo distribuimos los datos?

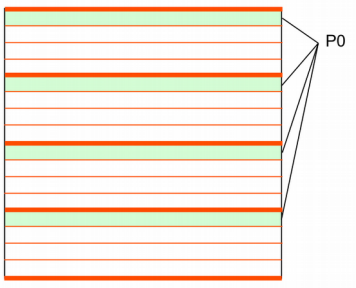
Dependiendo de cómo sea el problema (iterativo o recursivo) se suele hablar de:

- Recursive Data Decomposition: Para aquellas estructuras estilo grafos, árboles, etc. que normalmente se tiene que tratar con programas recursivos. Estas estructuras de datos normalmente son difíciles de gestionar y requieren realizar un buen particionamiento del grafo/árbol, etc. En las slides 18 y 19 se muestran dos estrategias, como ejemplo, de particionamiento de grafos cuyos nodos tienen información espacial de los elementos/cuerpos del problema N-body (un problema muy famoso que no hemos explicado

todavía :-) y no sabemos si podremos explicar, aunque no os preocupéis porque no es crítico). La distribución/particionamiento en el espacio se ha usado para dividir el grafo en partes y así poder distribuir el trabajo. Estas estrategias han seguido la estrategia Quadtree y Orthogonal distribution. La primera siempre divide el espacio en 4 partes iguales hasta que ya sólo queda un cuerpo/elemento. La segunda es más consciente de los elementos en una partición para intentar dividir en 2 partes que a cada parte haya la mitad de cuerpos/elementos.

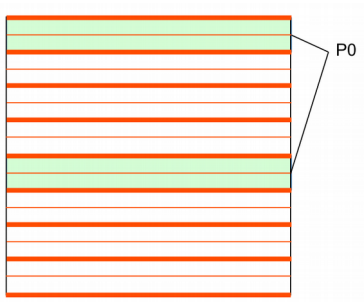
- Geometric Data Decomposition: Para aquellas estructuras que se puedan dilimitar sus rangos en las dimensiones. Prácticamente estamos hablando de estructuras de datos con una o más dimensiones, y el tipo de división/particionamiento se hace respecto a una o más dimensiones. En Paralelismo nos centramos en vectors (1D) y matrices (2D) y como mucho hacemos particionamientos en las dos dimensiones.
    - Block Geometric Data Decomposition: Tomando como referencia una dimensión, este tipo de particionamiento indica que cada thread/proceso se lleva N/nthreads posiciones consecutivas en esa dimensión. En la Slide 12 , figura de la izquierda, se observa que se ha hecho un Block Geometric Data Decomposition, por filas, de la matriz (es decir lo que llamaremos en PAR: Block Geometric Data Decomposition by rows). Bloques de filas se le asignan a cada proceso/thread.

    - Cyclic Geometric Data Decomposition: Tomando como referencia una dimensión, este particionamiento hace que a cada thread le toque una posición (fila), y si no se han acabado las posiciones se vuelve a repetir le patron de 1 posición por thread, hasta acabarse todas las posiciones. En la Slide 12, figura de la derecha, se observa como al thread 0 (proceso 0 – P0) se le asigna una posición (fila) cada cierto número de posiciones (Cyclic Geometric Data Decomposition by rows). Este número de posiciones corresponde con el total the threads que trabajan en este programa.



    - Block-cyclic Geomtric : Es un híbrido de los anteriores. Se de distribuyen bloques de posiciones (no tantos como N/nthreads ya que seria block) ni una sóla posición (que

sería un cyclic puro). El objectivo de esta estrategia es intentar reducir el overhead que puede significar repartir una posición por thread, pero evitando problemas de desbalanceo de carga de trabajo (ejemplo en la slide 13). En la transparencia número 14 (a la derecha) podemos un ejemplo de este tipo de distribución por filas de la matriz (Block-cyclic Geometric Data Decomposition by rows using chunks of 2 rows).

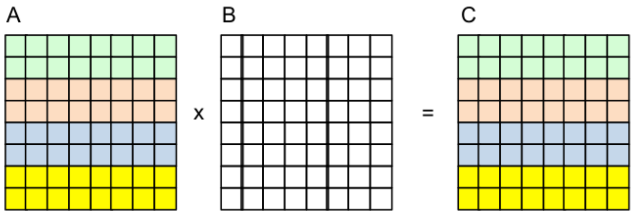


En esta estrategia de Data Decomposition normalmente estamos hablando de un scheduling estático de las tareas y con un omp for, en el caso de distribuir una estructura de datos que se recorre con un bucle es muy fácil implementarlo con un omp for y el scheduling adecuado. Ahora bien, para poder acercarnos más como se debería gestionar en una arquitectura con memoria distribuida y para abarcar muchos más casos de data partitioning (un acceso a una estructura donde **NO hay bucle para poder hacer omp for, como por ejemplo el caso del hash_table en el caso del problema 2 del Tema 5)**, no usaremos, en la mayoría de los casos, la directiva omp for.

Así, una vez hemos determinado qué estrategia de Data Partition vamos a seguir, podemos preguntar qué thread somos y, en función de este identificador lógico trabajar con el conjunto de datos de entrada, salida o entrada/salida que le corresponda a ese thread/proceso.

Ejemplo práctico con el Matrix Multiply

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE])
{
   for (int i=0; i<MATSIZE; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
}
```



A and C partitioned by rows on 4 processors (logically in shared memory architectures, physically in distributed memory architectures). B is replicated.

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE]) {

      for (int i=0; i<MATSIZE; i++)
            for (int j=0; j<MATSIZE; j++)
```

```
                    for (int k=0; k<MATSIZE; k++)
                          C[i][j] += A[i][k]*B[k][j];
}
```

El ejemplo realiza un Block Geometric Data Decomposition por filas de la Matriz A y C. Es decir, aquí se está haciendo del input A y del output C. En cambio, la matriz B, se considera replicada ya que todos los threads acceden a todos los elementos.

Nosotros en clase nos centramos en otro ejemplo que tambíen os hemos colgado en el Racó. Vimos cómo realizar el block, cyclic y block-cyclic, por filas, de un ejemplo donde se procesaban todos los elementos de una matriz, aplicándole una función y haciendo la acumulación del resultado sobre una variable escalar sum. La matrix, se distribuye de las diferentes formas mientras que la variable sum se supone replicada – el detalle de lo que vimos se puede ver en tema5_blocking.pdf. Pensar cómo se podría haber por columnas :-)

También hemos visto el problema 2 del Tema 5 de forma extendida. Hemos visto como las diferentes distribuciones de la entrada y salida puede implicar acceso remoto o local, con el pertinente coste de acceso en una máquina NUMA. En este ejemplo vimos también ejemplos de aplicar block, block-cyclic, y cyclic sobre un vector de salida, hash_table.

EI! MIRAROS ESTE PROBLEMA! Allá vimos ejemplos de BLOCK geometric data decomposition del input, siendo este un vector 1D, y sobre todo la parte donde se distribuye el output (hash_table), también 1D, però donde no hay bucle para hacer el particionamiento. Vimos cómo hacerlo para BLOCK Geometric Data decomposition. CUIDADO! En estos ejemplos no hay una buena distribución de la carga entre particiones ya que se le dá todo al último. Podríais pensar como distribuirlo de forma adecuada :-) . Sólo mirando el ejemplo del tema5_blocking os debería ser fácil mejorarlo. Finalmente, en las dos últimas transparencias se muestra como hacer un CYCLIC y un BLOCK-CYCLIC Geometric Data decomposition del output hash_table. Sería muy bueno que lo mirárais para el próximo dia.

Finalmente, el próximo dia veremos un ejemplo de geometric data distribution para bloques con el problema 5 del Tema 5. Cuidado, en el apartado de la repartición a bloques cuadrados del trabajo entre K^2 threads, pensad en esta distribución (y dejar de banda la que pide el problema):

0 1 2

3 4 5

6 7 8

Finalmente, un resumen de donde encontrar ejemplos:

Documentos y ejemplos de tipos de geometric data decomposition.

Tema5_blocking.pdf:

BLOCK, CYCLIC, BLOCK_CYCLIC Geometric Data Decomposition by rows (Matriz input 2D) . Con buen balanceo de carga. Variable de salida replicada, de la que se hace reducción.

He añadido un Geometric Data Decomposition by BLOCKS de 2 filas x 4 columnas, considerando que tenemos 2x4 threads.

Problema 2 :

BLOCK Geometric Data Decomposition (Input data 1D) - Sin balanceo de carga, con sincronización de acceso al hash_table que está distribuida.

BLOCK Geometric Data Decomposition de OUTPUT (hash table 1D) – Sin balanceo de carga, pero SIN SINCRONIZACIÓN.

CYCLIC Geometric Data Decomposition de OUTPUT (hash table 1D) - FALTA VERLO EN CLASE

BLOCK-CYCLIC Geometric Data Decomposition de OUTPUT (hash table 1D) – FALTA VERLO EN CLASE

Problema 5 (FALTA VERLO EN CLASE)

A partir de la transparencia 10: Geomtric Data Decomposition de INPUT BY BLOCKS (BsxBS) - FALTA VERLO EN CLASE