***Análisis de Cache en la Mutiplicación de Matrices en C con Paralelismo de Memoria Compartida y OpenMP***

**Israel Nazareth Chaparro Cruz (*Autor*)**

Maestrante en Ingeniería en Informática y Sistemas

Universidad Nacional Jorge Basadre Grohmann

Tacna, Perú

ichaparroc@unjbg.edu.pe

**Fernando Edgard Oviedo Liendo (*Autor*)**

Maestrante de Ingeniería de Sistemas e Informática

Universidad Nacional Jorge Basadre Grohmann

Tacna, Perú

foviedo.liendo@gmail.com

1. **RESUMEN**

En el trabajo se implementó un programa multiplicación de matrices con paralelismo de memoria compartida y OpenMP evaluar distintas medidas de desempeño en el procesamiento de grandes cantidades de datos, obteniendo como resultado que el manejo del almacenamiento de datos prima por sobre la cantidad de operaciones a realizarse respecto al tiempo de ejecución de los mismos.

1. **INTRODUCCIÓN**

Durante varios años, los procesadores han sido capaces de procesar información más rápido que acceder a datos en la memoria principal, eso quiere dice, si un procesador debe leer datos de la memoria principal por cada operación que realice, pasará la mayor parte del tiempo esperando que esos datos lleguen de la memoria. Para abordar este problema, los diseñadores de chip han agregado bloques de memoria a los procesadores relativamente más rápidos, llamados memoria caché.

El diseño de la memoria caché tiene en cuenta los principios de *localidad espacial* cuando un programa accede a una instrucción o a un dato, existe una elevada probabilidad de que instrucciones o datos cercanos se accedan pronto, y *localidad temporal* cuando un programa accede a una instrucción o dato, existe una elevada probabilidad de que esa misma instrucción o dato vuelva a ser accedido pronto. El espacio que tiene una caché para albergar un bloque de memoria principal se le denomina línea de caché.

El uso de la memoria caché tiene un enorme impacto en la memoria compartida.

1. **MARCO TEÓRICO**
   1. ***COHERENCIA DE CACHE***

Las cache de las cpu son manejados por el hardware, esto impide que al programar se tenga control directo en ellos y trayendo consecuencias importantes en sistemas que utilizan memoria compartida. En el siguiente gráfico se presenta una arquitectura de dos núcleos con memoria compartida, en el que cada núcleo cuenta con su propia memoria cache.

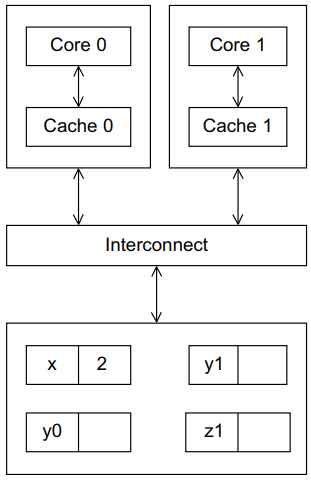


Figura 1: Arquitectura de Memoria Compartida de 2 núcleos.

Supongamos que *x* es una variable compartida inicializada en 2, *y0* es una variable privada del *Core 0*, *y1* y *z1* son variables privadas del Core1. Supongamos que se ejecutan en los tiempos indicados según el cuadro adjunto:

Tabla 1: Tiempos e instrucciones por cada núcleo.

|  |  |  |
| --- | --- | --- |
| **Time** | ***Núcleo 0*** | ***Núcleo 1*** |
| 0 | y0 = x ; | y1 = 3 \* x ; |
| 1 | x = 7 ; | Declaración no relacionada |
| 2 | Declaración no relacionada | z1 = 4 \* x ; |

En el *Time 0* la ubicación de memoria para *y0* se inicializará en 2, y ubicación de memoria para *y1* será 6. Sin embargo, no tenemos claro aún qué valor obtendrá *z1*. Podría parecer que en el *Time 1* del *Core 0*, el valor de x es actualizado a 7, luego de esto el valor de *z1* en el *Core 1* sería de 28 (z1 = 4 \* 7). Sin embargo, en el *Time 0*, *x* se encuentra en la memoria cache del *Core 1*, pero en el *Time 2 el* valor de *x* es *7* almacenado en el cache del *Core 0*, pero no en el cache del Core 1, continuando este con el valor 2, por consiguiente, en el Time 2, z1 obtendrá el valor de 8.

Para asegurar la coherencia de cache, existen dos enfoques principales: la *coherencia de cache snooping* y la *coherencia de cache basada en directorios*

* 1. ***COHERENCIA DE CACHE SNOOPING***

La idea de snooping viene de sistemas basados en buses, cuando los núcleos comparten un bus, cualquier señal transmitida en el bus pueden ser “vistos” por todos los núcleos conectados al mismo bus.

Por lo tanto, cuando el *Core 0* actualiza *x* almacenado en su cache, y transmite esta información a través del bus, y si el *Core 1* está snooping (“fisgoneando”) el bus, éste verá que el valor de x ha cambiado y puede marcar esta copia de *x* como inválida.

* 1. ***COHERENCIA DE CACHE BASADO EN DIRECTORIOS***

Los protocolos de coherencia de cache basado en directorios, pretenden resolver el problema mediante una estructura de datos llamada *directorio*. El directorio almacena el estado de cada línea de cache. Por lo general, esta estructura de datos es distribuida. Cuando se lee una línea de la memoria cache del *Core 0*, la entrada del directorio correspondiente a esa línea se actualizaría indicando que el *Core 0* tiene una copia de esa línea. Cuando una variable es actualizada, el directorio es consultado, y los controladores de cache de los núcleos que tienen la línea de cache de esa variable en sus caches son invalidados.

Se requerirá un almacenamiento adicional para el directorio, pero cuando se actualiza una variable cache, solo los *Core* que tienen almacenada esa variable deberán ser contactados.

* 1. ***FALSO COMPARTIDO***

Las memorias caches están implementadas en el hardware de la CPU, operan en líneas de caches, no en variables individuales. Esto puede traer consecuencias desastrosas en el rendimiento.

Supongamos que queremos llamar a la función *f(i,j)* y añadir los valores calculados en un vector:

*int i, j, m, n;*

*double y[m];*

*/∗ Se asigna y = 0 ∗/*

*. . .*

*for (i = 0; i < m; i++)*

*for (j = 0; j < n; j++)*

*y[i] += f(i,j);*

Podemos paralelizar esto dividiendo en un bucle externo entre los Core. Si tenemos *core\_count* cores podremos asignar las primeras *m/core\_count* iteraciones hasta el primer *Core*, las siguientes *m/core\_count* iteraciones al segundo *Core* y así sucesivamente.

*int i,j,iter\_count,m,n,core\_count*

*double y[m];*

*iter\_count = m/core\_count*

*/∗ Core 0 hace esto ∗/*

*for (i = 0; i < iter count; i++)*

*for (j = 0; j < n; j++)*

*y[i] += f(i,j);*

*/∗ Core 1 hace esto ∗/*

*for (i = iter count+1; i < 2∗iter count; i++)*

*for (j = 0; j < n; j++)*

*y[i] += f(i,j);*

El falso compartido tiene lugar cuando dos procesadores usan variables distintas pero que están en la misma línea de cache (o en la misma página). En este caso, cuando uno de los procesadores accede a una variable de estas para escritura, el bloque debe ser transmitido al otro procesador, pues siendo variables distintas comparten línea de cache. En la siguiente figura se puede observar como dos variables distintas (A y B), están compartiendo la misma línea de cache, lo cual provocará múltiples invalidaciones entre los procesadores cada vez que uno de ellos escribe en una de estas dos variables.

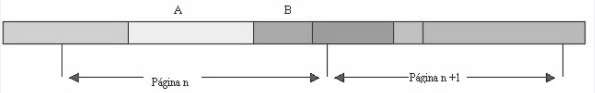


Figura 2: Dos variables almacenadas en una misma línea de cache.

Tengamos en cuenta que el falso compartido no causa resultados incorrectos. Sin embargo, puede perjudicar el rendimiento de un programa al causar muchos más accesos a la memoria de lo necesario. Podemos reducir su efecto mediante usando almacenamiento temporal que es local para el hilo o proceso y luego copiar temporalmente en el almacenamiento compartido.

Recordemos que denominamos *cache-miss* al proceso en el que un núcleo busca un determinado dato en la memoria cache y no lo encuentra, esto está relacionado directamente al falso compartido y coherencia de cache.

1. **DESARROLLO**
   1. ***MULTIPLICACIÓN DE MATRICES***

El uso de la Coherencia de Caché puede tener un dramático efecto en el rendimiento de sistemas de memorias compartidas. Definamos las siguientes matrices:

Multiplicando*:* Matriz *A(a,b)* de dimensiones *a* x *b*

Multiplicador*:* Matriz *B(b,c)* de dimensiones *b* x *c*

Resultado*:* Matriz *C(a,c)* de dimensiones *a* x *c*

En donde por propiedad, para que pueda ser llevada la multiplicación es necesario que el número de columnas de la matriz *A* sea igual al número de filas de la matriz *B* y el resultado será una matriz con el número de filas de la matriz *A* y el número de columnas de la matriz *B*.

Las matriz *C* (resultado) es un array de *l* arrays de tamaño *n*, entonces para el producto: *A\*B = C* el resultado para elemento de la matriz *C* es:

*Ci,j = Ai,k \* Bk,j*

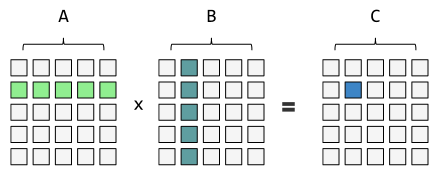
En donde el iterador i recorre las filas de la matriz C, j las columnas y k recorre simultáneamente las columnas de la matriz A y filas de la matriz B:

*for(i=0;i<l;i++)*

*for(j=0;j<c;j++)*

*for(k=0;k<b;k++)*

*C[i][j]+=A[i][k]\*B[k][j];*

 Figura 2: Multiplicación de matrices.

* 1. ***PARALELIZACIÓN DE LA MULTIPLICACIÓN DE MATRICES EN***

Es posible paralelizar el código presentado anteriormente en un entorno de memoria compartida, para ello el objetivo será paralelizar el primer *for* vinculado al *iterador* i que recorre las filas de la matriz *C*, de esta forma podemos asegurar que cada fila de la matriz resultado solo sea manejada por un hilo de ejecución, lo que nos evita en primera instancia un problema de coherencia de cache:

*#pragma omp parallel for num\_threads(thread\_count) default(none) private(i,j,k) shared(A,B,C,a,b,c)*

*for(i=0;i<l;i++)*

*for(j=0;j<c;j++)*

*for(k=0;k<b;k++)*

*C[i][j]+=A[i][k]\*B[k][j];*

* 1. ***CASOS DE ESTUDIO***

A continuación, se presentan los tamaños de la matriz A para cada experimento:

*Caso* *1: 8000000 x 8*

*Caso 2: 8000 x 8000*

*Caso 3: 8 x 8000000*

Teniendo los valores de las dimensiones *a* y *b*, el valor de *c* es delimitado de acuerdo a la cantidad de memoria disponible en el recurso computacional utilizado, considerándose un valor de *c* igual a 8, las cantidades de memoria requeridas para almacenar las matrices de enteros A, B y C y la cantidad de sumas (a\*(b-1)\*c) por caso son:

Tabla 2: Memoria y Cantidad de Sumas por caso.

|  |  |  |  |
| --- | --- | --- | --- |
| **Caso** | **Dimensiones** | **Memoria** | **Cantidad de Sumas** |
| 1 | 8000000 x 8 x 8 | 488.3 MB | 448000000 |
| 2 | 8 x 8000000 x 8 | 244.6 MB | 511999936 |
| 3 | 8000 x 8000 x 8 | 488.3 MB | 511936000 |

Cabe destacar que para caso el número de multiplicaciones es 512,000,000, y es esta la medida priorizada para la estandarización de experimento, debido a que la velocidad del procesador para ejecutar una suma es mucho menor que la de una multiplicación; recalcando que para un número de multiplicaciones “m” y sumas “n” solo existe una tupla de dimensiones “a,b,c”.

* 1. ***MATERIALES Y MÉTODOS***

Para el presente trabajo, se contó con los siguientes recursos computacionales para llevar a cabo la experimentación:

Tabla 3: Recursos computacionales.

|  |  |
| --- | --- |
| **Característica** | ***Valor*** |
| Procesador | Intel Atom® N570 |
| Núcleos físicos | 2 |
| Hilos | 4 |
| Cache | 1MB L2 (262144 enteros) |
| RAM | 2 GB |

Como métodos se hizo uso del binario */usr/bin/time* con parámetro *verbose* para determinadas mediciones, y la herramienta *perf* para la medición de performance y analizador de cache. La programación fue realizada en el lenguaje C – compilador GCC utilizando OpenMP.

1. **RESULTADOS**
   1. ***EXPERIMENTOS***

Para cada caso de estudio, se realizó un total de 10 ejecuciones intercaladas bajo el mismo recurso computacional, las mediciones no difirieron en mucho.

Se utilizaron los siguientes comandos desde la terminal para la medición:

*#/usr/bin/time –v . ./matrix [hilos] [a] [b] [c]*

*#perf stat -e LLC-loads,LLC-load-misses ./matrix [hilos] [a] [b] [c]*

El código fuente como el ejecutable se encuentran disponibles en: <https://github.com/ichaparroc/omp-matrix-mult>

La tabla resumen del promedio de 10 experimentos por caso se presenta a continuación:

Tabla 4: Experimentos.

|  |  |  |  |
| --- | --- | --- | --- |
| **Medida** | **Caso 1**  **8000000x8x8** | **Caso 2**  **8000x8000x8** | **Caso 3**  **8x8000000x8** |
| Mutiplicaciones | 512000000 | 512000000 | 512000000 |
| Sumas | 448000000 | 511936000 | 511999936 |
| Memoria (MB) | 488.3 | 244.6 | 488.3 |
| CPU % | 259 | 336 | 301 |
| Tiempo (s) | 11.93 | 8.43 | 21.87 |
| cache-loads | 16628452 | 305220963 | 468658580 |
| cache-misses % | 10.97 | 1.28 | 19.61 |

Para todos los casos, el número de *hilos* utilizados es 4. La siguiente gráfica de área apilada de 100% ilustra el rendimiento por cada caso:

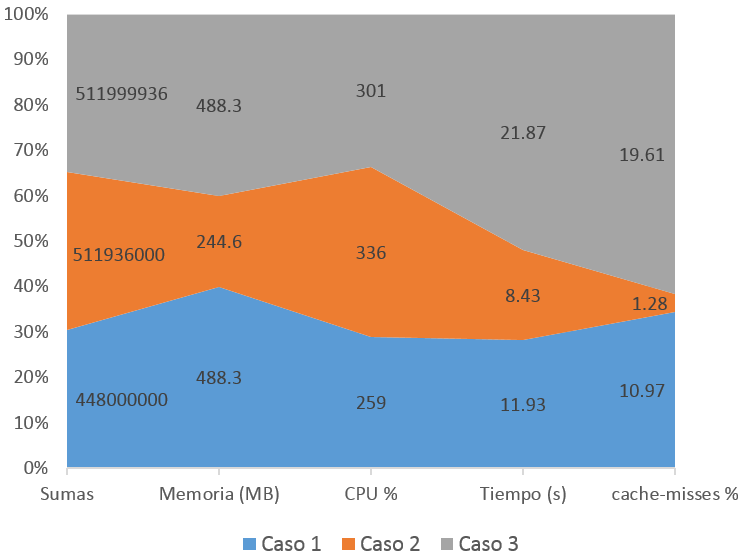


Figura 3: Experimentos.

* 1. ***ANÁLISIS E INTERPRETACIÓn***

Conforme al gráfico, podemos observar que la combinación del número de sumas y cantidad de memoria necesaria influyen en el tiempo de ejecución de cada caso, para el caso 2 el número de sumas es el 2do más grande, pero al ser la cantidad de memoria a manejar la mitad de pequeña respecto a los otros dos casos, es por ello que el tiempo de ejecución es menor que el del caso de menor cantidad de sumas.

Así mismo, el porcentaje de CPU utilizado está relacionado al tiempo de ejecución en la medida que este se encuentre ejecutando instrucciones y no esperando los datos de la cache o memoria principal, sin embargo, encontramos un porcentaje de CPU utilizado mayor en el caso 3 que en el caso 1 debido a que el caso 3 tiene un 13% más de sumas que el caso 1.

Por otro lado, existe una evidente relación ente el tiempo de ejecución y el porcentaje de cache-misses, la relación es casi la misma en los casos 3 y 1, en el caso 2 la relación es totalmente diferente, esto debido al tamaño de la matriz resultante, los tamaños de la matiz resultante C se presentan a continuación:

*Caso 1: 8000000x8 (b=8)*

*Caso 2: 8000x8 (b=8000)*

*Caso 3: 8x8 (b=8000000)*

Es importante tomar en cuenta no solo el tamaño de la matriz resultante, sino también la dimensión *b*, debido a que esta recorrerá las columnas de la matriz *B*, lo que provoca que por localidad espacial cada hilo cargue a la cache innecesariamente cada fila de la matriz *B*.

Es decir, para el caso 1 tenemos como valor de la dimensión b a 8, lo que provocará menos carga innecesaria de filas de la matriz B a la cache, pero el número de cache-misses incrementará debido a que la matriz contendrá 8000000 arrays de tamaño 8, pudiendo estar estos arrays almacenados de manera cotigua provocando cache-misses.

Para el caso 2, tenemos la ventaja de un menor tamaño de memoria requerida, a su vez el parámetro b no es muy elevado a comparación del caso 3, ello resulta en un porcentaje muy pequeño de cache-misses, sin embargo el tiempo de ejecución no llega a ser mucho menor que el caso 1 debido a que no se ha considerado una medición separa de la multiplicación de matrices en sí mismo y la locación de memoria e inicialización de las matrices.

Para el caso 3, tenemos la gran desventaja de un número grande como parámetro *b*, lo que provocará mucha carga innecesaria de filas de la matriz B en la cache, a su vez presenta la mayor cantidad de cache-misses debido al tamaño de la matriz resultante cuyas filas por su pequeño tamaño tienen más probabilidades de compartir una línea de caché.

1. **conclusiones**

Es importante tener en cuenta al momento de implementar programas paralelos la forma en como estos se van a almacenar la información por sobre la cantidad de operaciones que se van a realizar, el caso de estudio con menos sumas fue un 30% más lento que el más rápido.

Es muy importante tomar en cuenta que la velocidad a la que el procesador ejecuta operaciones es mucho mayor que la velocidad de acceso a los datos, es por ello que es importante enfocarse a como estos se almacenan y adecuarlos al mecanismo de manejo de cache que tiene el hardware. Por ejemplo, una implementación de las matrices como un array gigante y no como un array a arrays hubiera afectado significativamente los valores de todo el experimento.

Por otro lado, fue relativamente más dificultoso manejar la experimentación y los casos de estudio en el presente trabajo que respecto a la multiplicación matriz-vector, mostrándose de forma más notoria en este último como el falso compartido incrementa el número de cache-misses.

**referencias**

1. Peter S. Pacheco, An Introduction to Parallel Programming, cap.2, pp.43-46, edic. 2011.
2. Peter S. Pacheco, An Introduction to Parallel Programming, cap 5, pp.251-256., edic. 2011.
3. Vicente Arnau Llombart, Organización de Computadoras, 2012, cap 5, pp.22.