Puebas sobre el comportamiento de la memoria cache

Cáceres Terrones, Sixto Manuel
Ciencia de la Computación
Universidad Católica San Pablo
sixto.caceres@ucsp.edu.pe

I. Introducción

Este informe analiza el rendimiento de algoritmos paralelos, centrándose en cómo la localidad de datos en caché afecta la ejecución. Según el libro de Pacheco (sección 2.2.3 de "An Introduction to Parallel Programming") [1], comparamos diferentes métodos de acceso a matrices. El acceso por filas aprovecha la contigüidad de datos minimizando fallos de caché, mientras el acceso por columnas provoca accesos dispersos aumentando dichos fallos. Los experimentos en C++ y Go confirman que el orden de acceso impacta significativamente en la eficiencia de aplicaciones paralelas.

II. Bucles anidados del libro

Dentro del libro *An Introduction to Parallel Programming* de Peter Pacheco [1], en el capítulo 2 titulado *Parallel Hardware and Parallel Software*, se discute el impacto de la localidad espacial y temporal en la caché del sistema. En la sección 2.2.3 se ejemplifica cómo el orden de acceso a una matriz bidimensional, almacenada en formato row-major, afecta el rendimiento del programa. El texto explica que al recorrer la matriz por filas se aprovecha la contigüidad de los datos en memoria, lo que reduce significativamente los fallos de caché en comparación con el acceso por columnas [1].

A. Bucles anidados en C++

El código en C++ implementa dos variantes para el acceso a una matriz, en línea con lo descrito previamente en la sección sobre "Bucles anidados del libro". Se distinguen dos patrones:

- **Acceso por filas:** Se recorre la matriz de forma secuencial por filas, aprovechando la localidad espacial, ya que los elementos de cada fila están almacenados contiguamente. Esto permite que, una vez cargada una línea de la caché, se reutilice para acceder a los siguientes elementos de la misma fila, reduciendo el número de fallos de caché [1].
- **Acceso por columnas:** Se recorre la matriz columna por columna, lo que implica saltar entre diferentes filas. Dado que los elementos consecutivos en memoria corresponden a elementos de la misma fila y no de la misma columna, este patrón de acceso resulta en un mayor número de fallos de caché.

A continuación se incluye la imagen correspondiente al gráfico generado a partir de este código en C++:

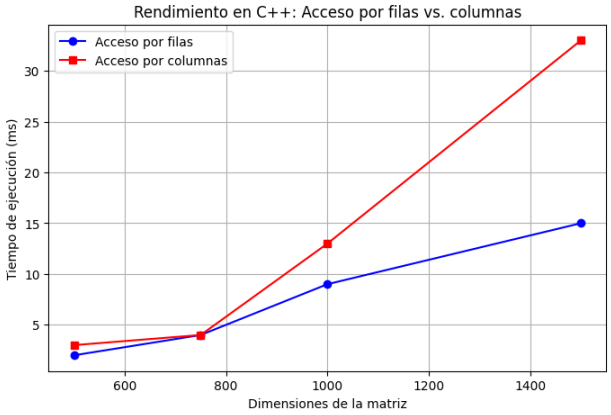

Fig. 1. Valgrind sobre el ejecutable de C++

La Figura 1 demuestra que el acceso por filas es considerablemente más eficiente que el acceso por columnas, lo que confirma lo expuesto en el libro acerca de la ventaja de la localidad espacial para reducir fallos de caché.

B. Bucles anidados en Go

El código equivalente en Go sigue la misma lógica de acceso, permitiendo así establecer una comparación directa con la implementación en C++:

- **Acceso por filas:** Se recorre la matriz secuencialmente por filas, facilitando el aprovechamiento de la caché debido a la contigüidad de los datos, similar a la implementación en C++ [1].
- **Acceso por columnas:** El recorrido por columnas implica un acceso disperso en memoria, lo que resulta en mayor latencia por fallos de caché.

La imagen a continuación muestra el gráfico generado con la implementación en Go:

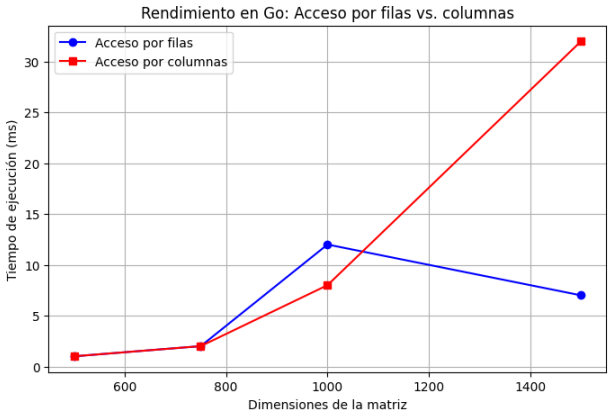

Fig. 2. Valgrind sobre el ejecutable de Go

La Figura 2 confirma que, al igual que en C++, el acceso por filas supera al acceso por columnas en eficiencia. Esto se debe al menor número de fallos de caché al aprovechar la disposición en memoria de los datos, lo cual es coherente con la explicación detallada en el libro [1].

III. Multiplicación Clásica de Matrices

En este análisis se evalúa el rendimiento de la multiplicación clásica de matrices implementada en C++ y Go, utilizando matrices de tamaño 512×512. El objetivo es comparar los *cache misses* (específicamente los D1 misses) entre ambas implementaciones para entender su eficiencia en términos de acceso a memoria.

A. Datos y Algoritmo

Se utilizaron matrices de tamaño 512×512, lo que resulta en un total de $262,144$ elementos por matriz. El algoritmo clásico de multiplicación de matrices sigue un enfoque directo: para calcular cada elemento $C[i][j]$ de la matriz resultante, se realiza el producto punto de la fila i de la primera matriz A y la columna j de la segunda matriz B. Esto implica tres bucles anidados.

```
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$ valgrind --tool=ca
chegrind --cache-sim=yes ./classic_c
==6775== Cachegrind, a high-precision tracing profiler
==6775== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==6775== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==6775== Command: ./classic_c
==6775==
--6775-- Warning: Cannot auto-detect cache config, using defaults.
--6775--          Run with -v to see.
Tiempo clásico: 19019 ms
==6775==
==6775== I refs:        7,264,335,616
==6775== I1  misses:            2,142
==6775== LLi misses:            2,127
==6775== I1  miss rate:          0.00%
==6775== LLi miss rate:          0.00%
==6775==
==6775== D refs:        4,037,160,861  (2,691,925,805 rd   + 1,345,235,056 wr)
==6775== D1  misses:      134,927,396  (  134,564,840 rd   +       362,556 wr)
==6775== LLd misses:      134,916,135  (  134,554,152 rd   +       361,983 wr)
==6775== D1  miss rate:           3.3% (          5.0%     +           0.0%  )
==6775== LLd miss rate:           3.3% (          5.0%     +           0.0%  )
==6775==
==6775== LL refs:         134,929,538  (  134,566,982 rd   +       362,556 wr)
==6775== LL misses:       134,918,262  (  134,556,279 rd   +       361,983 wr)
==6775== LL miss rate:            1.2% (          1.4%     +           0.0%  )
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$
```

Fig. 3. Valgrind sobre el ejecutable de C++

```
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$ valgrind --tool=ca
chegrind --cache-sim=yes ./classicgo
==6797== Cachegrind, a high-precision tracing profiler
==6797== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==6797== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==6797== Command: ./classicgo
==6797==
--6797-- Warning: Cannot auto-detect cache config, using defaults.
--6797--          Run with -v to see.
Tiempo clásico: 5.17320401s
==6797==
==6797== I refs:        2,430,272,202
==6797== I1  misses:            4,587
==6797== LLi misses:            3,443
==6797== I1  miss rate:          0.00%
==6797== LLi miss rate:          0.00%
==6797==
==6797== D refs:          273,348,857  (271,653,607 rd   + 1,695,250 wr)
==6797== D1  misses:      134,885,596  (134,555,228 rd   +   330,368 wr)
==6797== LLd misses:      134,876,689  (134,547,445 rd   +   329,244 wr)
==6797== D1  miss rate:          49.3% (       49.5%     +      19.5%  )
==6797== LLd miss rate:          49.3% (       49.5%     +      19.4%  )
==6797==
==6797== LL refs:         134,890,183  (134,559,815 rd   +   330,368 wr)
==6797== LL misses:       134,880,132  (134,550,888 rd   +   329,244 wr)
==6797== LL miss rate:           5.0% (        5.0%     +      19.4%  )
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$
```

Fig. 4. Valgrind sobre el ejecutable de GO

B. Complejidad Algorítmica

El algoritmo clásico de multiplicación de matrices tiene una complejidad algorítmica de $O(n^3)$, donde n es el tamaño de la matriz (en este caso, $n = 512$). Esto se debe a los tres bucles anidados, cada uno iterando n veces, lo que resulta en un total de n^3 operaciones.

C. Análisis y Comparativa

Los resultados obtenidos con `valgrind --tool=cachegrind`, mostrados en las Figuras 3 y 4, revelan los siguientes *D1 misses*:

- **C++:** $134,927,336$ *D1 misses*, con una tasa de miss del 3.3%.
- **Go:** $134,885,596$ *D1 misses*, con una tasa de miss del 49.3%.

A pesar de que el número total de *D1 misses* es similar entre ambas implementaciones, la tasa de miss en Go es significativamente mayor. Esto se debe a que C++ realiza muchas más referencias de datos $(4,037,160,861)$ en comparación con Go $(273,348,857)$, lo que diluye su tasa de miss. En cambio, Go, con menos referencias, tiene una tasa de miss más alta, indicando un uso menos eficiente del caché.

IV. Multiplicación de Matrices por Bloques

En este análisis se evalúa el rendimiento de la multiplicación de matrices por bloques implementada en C++ y Go, utilizando matrices de tamaño 512×512. El objetivo es comparar los *cache misses* (específicamente los D1 misses) entre ambas implementaciones para entender su eficiencia en términos de acceso a memoria.

A. Datos y Algoritmo

Se utilizaron matrices de tamaño 512×512, lo que resulta en un total de $262,144$ elementos por matriz. En la multiplicación por bloques, las matrices se dividen en submatrices de tamaño 64×64, y el algoritmo opera bloque a bloque. Para cada elemento $C[i][j]$ de la matriz resultante, se calculan los productos de las submatrices correspondientes de A y B, mejorando la localidad de los datos al trabajar con porciones más pequeñas que caben en el caché.

```
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$ valgrind --tool=ca
chegrind --cache-sim=yes ./block_c
==6992== Cachegrind, a high-precision tracing profiler
==6992== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==6992== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==6992== Command: ./block_c
==6992==
--6992-- Warning: Cannot auto-detect cache config, using defaults.
--6992--          Run with -v to see.
Tiempo bloques: 26560 ms
==6992==
==6992== I refs:        10,839,010,492
==6992== I1  misses:             2,154
==6992== LLi misses:             2,136
==6992== I1  miss rate:           0.00%
==6992== LLi miss rate:           0.00%
==6992==
==6992== D refs:         6,039,981,738  (3,979,254,777 rd   + 2,060,726,961 wr)
==6992== D1  misses:       137,357,204  (  135,159,642 rd   +     2,197,562 wr)
==6992== LLd misses:        25,099,343  (   22,902,354 rd   +     2,196,989 wr)
==6992== D1  miss rate:            2.3% (          3.4%     +           0.1%  )
==6992== LLd miss rate:            0.4% (          0.6%     +           0.1%  )
==6992==
==6992== LL refs:          137,359,358  (  135,161,796 rd   +     2,197,562 wr)
==6992== LL misses:         25,101,479  (   22,904,490 rd   +     2,196,989 wr)
==6992== LL miss rate:             0.1% (          0.2%     +           0.1%  )
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$
```

Fig. 5. Valgrind sobre el ejecutable de C++

```
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$ valgrind --tool=ca
chegrind --cache-sim=yes ./blockgo
==6977== Cachegrind, a high-precision tracing profiler
==6977== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==6977== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==6977== Command: ./blockgo
==6977==
--6977-- Warning: Cannot auto-detect cache config, using defaults.
--6977--          Run with -v to see.
Tiempo bloques: 22.125842929s
==6977==
==6977== I refs:        2,731,912,523
==6977== I1  misses:            5,236
==6977== LLi misses:            3,855
==6977== I1  miss rate:          0.00%
==6977== LLi miss rate:          0.00%
==6977==
==6977== D refs:          683,004,078  (546,042,975 rd   + 136,961,103 wr)
==6977== D1  misses:      137,208,516  (135,041,906 rd   +   2,166,610 wr)
==6977== LLd misses:       21,865,634  ( 19,700,487 rd   +   2,165,147 wr)
==6977== D1  miss rate:          20.1% (       24.7%     +         1.6%  )
==6977== LLd miss rate:           3.2% (        3.6%     +         1.6%  )
==6977==
==6977== LL refs:         137,213,752  (135,047,142 rd   +   2,166,610 wr)
==6977== LL misses:        21,869,489  ( 19,704,342 rd   +   2,165,147 wr)
==6977== LL miss rate:            0.6% (        0.6%     +         1.6%  )
latermanuel@latermanuel-VirtualBox:~/Documents/Paralela/CacheMisses$
```

Fig. 6. Valgrind sobre el ejecutable de GO

B. Complejidad Algorítmica

El algoritmo de multiplicación por bloques tiene una complejidad algorítmica de $O(n^3)$, donde n es el tamaño de la matriz (en este caso, $n = 512$). Esto se debe a que, aunque se divide en bloques, el número total de operaciones sigue siendo proporcional a n^3. Sin embargo, la división en bloques mejora la eficiencia del caché al reducir accesos a memoria principal.

C. Análisis y Comparativa

Los resultados obtenidos con `valgrind --tool=cachegrind`, mostrados en las Figuras 5 y 6, revelan los siguientes *D1 misses*:

- **C++:** $137,357,204$ *D1 misses*, con una tasa de miss del 2.3%.
- **Go:** $137,208,516$ *D1 misses*, con una tasa de miss del 20.1%.

A pesar de que el número total de *D1 misses* es similar entre ambas implementaciones, la tasa de miss en Go es significativamente mayor. Esto se debe a que C++ realiza muchas más referencias de datos ($6,039,981,738$) en comparación con Go ($683,004,078$), lo que reduce su tasa de miss. En cambio, Go, con menos referencias, tiene una tasa de miss más alta, indicando un uso menos eficiente del caché.

V. Comparación entre la Multiplicación Clásica y la Multiplicación por Bloques

En este apartado se presenta el análisis comparativo entre dos metodos de multiplicación de matrices: el método *clásico* y el método *por bloques*. Se han realizado pruebas experimentales en dos lenguajes de programación (C++ y Go) para evaluar el comportamiento de cada algoritmo en función del tamaño de la matriz. A continuación se exponen las gráficas comparativas y se discuten las observaciones obtenidas.

A. Comparación en C++: Clásico vs. Por Bloques

A continuación se muestra la gráfica que contrasta los tiempos de ejecución obtenidos en C++ para ambos métodos de multiplicación. En el gráfico se observa cómo, en tamaños pequeños, el método clásico presenta tiempos menores debido al overhead de manejo de bloques; sin embargo, a medida que aumenta el tamaño de la matriz, el algoritmo por bloques mejora su desempeño al aprovechar la localidad de los datos.

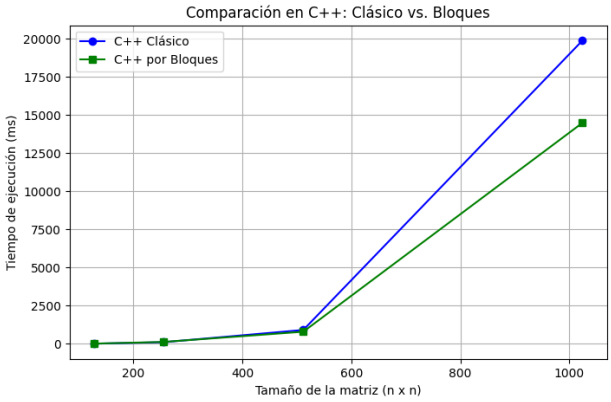

Fig. 7. Valgrind sobre el ejecutable de C++

Como se puede apreciar en la Figura 7, para matrices de dimensiones menores (por ejemplo, 128x128 y 256x256) el algoritmo clásico es competitivo o incluso superior, debido al overhead de dividir en bloques. Sin embargo, para dimensiones mayores (512x512 y 1024x1024) el enfoque por bloques muestra una clara ventaja, reflejando el beneficio de la optimización en la utilización de la memoria caché.

B. Comparación en Go: Clásico vs. Por Bloques

En el caso de Go, la gráfica a continuación ilustra la comparación entre los dos métodos.

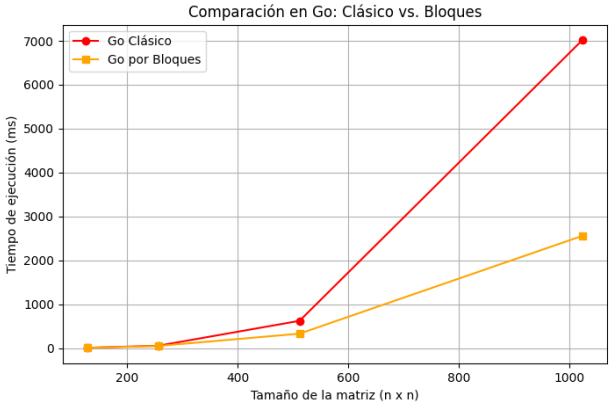

Fig. 8. Valgrind sobre el ejecutable de C++

Como se aprecia en la Figura 8, en Go el método por bloques muestra tiempos de ejecución inferiores en comparación con el método clásico para todos los tamaños de matrices evaluados. Este comportamiento se debe a una implementación que optimiza eficazmente la localidad de datos, como se detalla en [2], y a una eficiente utilización de la concurrencia, característica intrínseca del lenguaje Go [3]. Estas optimizaciones permiten que, incluso en matrices de menor tamaño, el algoritmo por bloques ofrezca un rendimiento superior, evidencia que se intensifica al aumentar la dimensión de las matrices.

VI. Conclusiones

El análisis realizado evidencia la importancia de la localidad de datos para mejorar el rendimiento en algoritmos paralelos. Se comprobó que el acceso secuencial por filas, aprovechando la contigüidad de la memoria, reduce significativamente los fallos de caché en comparación con el acceso por columnas. Además, la comparación entre la multiplicación clásica y la multiplicación por bloques demuestra que, si bien el enfoque clásico es competitivo en dimensiones reducidas, el método por bloques ofrece una optimización notable en el manejo de la caché para matrices de mayor tamaño. En resumen, implementar técnicas que maximicen la eficiencia de la memoria es crucial para el desarrollo de aplicaciones de alto rendimiento en entornos de cómputo actuales.

VII. Disponibilidad del Código

El código utilizado para las pruebas y la generación de los resultados presentados en este informe está disponible en el siguiente repositorio de GitHub:

`https://github.com/LaterManuel/CacheLabTest`

References

[1] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[3] R. Pike, "Concurrency is not parallelism," 2012, https://blog.golang.org/concurrency-is-not-parallelism.