Pruebas sobre el comportamiento de la memoria caché

Roque Sosa Owen Haziel
Universidad Nacional de San Agustín
oroque@unsa.edu.pe
Arequipa

2024

Resumen

Se realizará la implementación, resultados y análisis de ejecución de multiplicación de matrices clásica, por bloques; usando *Valgrind* y *QCacheGrind*

1. Implementación y comparación de los 2-bucles anidados FOR del libro

En este apartado se presentan los resultados obtenidos al comparar el tiempo de ejecución entre dos implementaciones del cálculo de la suma prefijo: una secuencial y otra paralela utilizando un modelo PRAM simulado. Para medir los tiempos de ejecución se empleó la biblioteca `chrono` de C++ en un sistema con un arreglo de un millón de elementos.

1.1. Suma prefijo secuencial

```cpp
// Sequential version of prefix sum
void sequential_prefix_sum(vector<int> &A) {
    int n = A.size();
    for (int i = 1; i < n; i++) {
        A[i] += A[i - 1];
    }
}
```

1.2. Suma prefijo paralela (simulación PRAM)

```cpp
// Parallel version of prefix sum (simulated PRAM approach)
void parallel_prefix_sum(vector<int> &A) {
    int n = A.size();
    int logN = ceil(log2(n));

    // Outer loop simulating the number of iterations
    for (int i = 0; i < logN; i++) {
        // Inner loop, processing elements in parallel
        #pragma omp parallel for
        for (int j = pow(2, i); j < n; j++) {
            A[j] += A[j - pow(2, i)];
        }
    }
}
```

1.3. Resultados

- **Suma prefijo secuencial:** el tiempo de ejecución de la versión secuencial fue de aproximadamente **5 milisegundos**.
- **Suma prefijo paralela (PRAM simulado):** el tiempo de ejecución de la versión paralela fue de aproximadamente **633 milisegundos**.

1.4. Análisis

A partir de los resultados obtenidos, se puede observar que la implementación secuencial es significativamente más rápida que la versión paralela simulada bajo el modelo PRAM. Esto se debe principalmente a varios factores:

- La implementación paralela, aunque simula un entorno paralelo, no aprovecha la verdadera paralelización del hardware. Esto introduce sobrecarga adicional en los cálculos, lo que resulta en un mayor tiempo de ejecución.
- La versión secuencial tiene una complejidad de tiempo lineal $\mathcal{O}(n)$, lo que la hace más eficiente para este tamaño de entrada específico (un millón de elementos) en comparación con la versión paralela, que tiene una complejidad logarítmica en el número de pasos y lineal en cada paso.
- En la simulación paralela, el acceso a memoria y la sincronización entre iteraciones añade una sobrecarga que no es trivial. A pesar de que el algoritmo en teoría tiene menor complejidad en el número de pasos, la constante oculta es grande debido a la sincronización.

2. Ejecución de algoritmos de multiplicación de matrices clásica vs. bloques en Valgrind

2.1. Resultados en Valgrind

```
owen@LAP-OWEN:/mnt/e/UNSA/2024B/PARALELA/LAB-MEMCACHE$ valgrind --leak-check=full ./matrix-classic
==3079== Memcheck, a memory error detector
==3079== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3079== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3079== Command: ./matrix-classic
==3079==
==3079== error calling PR_SET_PTRACER, vgdb might block
Evaluando la multiplicación clásica con matrices de tamaño 64x64
Tiempo para la multiplicación clásica: 0.104206 segundos
Evaluando la multiplicación clásica con matrices de tamaño 128x128
Tiempo para la multiplicación clásica: 0.805898 segundos
Evaluando la multiplicación clásica con matrices de tamaño 256x256
Tiempo para la multiplicación clásica: 8.63191 segundos
Evaluando la multiplicación clásica con matrices de tamaño 512x512
Tiempo para la multiplicación clásica: 74.4756 segundos
==3079==
==3079== HEAP SUMMARY:
==3079==     in use at exit: 0 bytes in 0 blocks
==3079==   total heap usage: 2,907 allocs, 2,907 frees, 4,335,376 bytes allocated
==3079==
==3079== All heap blocks were freed -- no leaks are possible
==3079==
==3079== For lists of detected and suppressed errors, rerun with: -s
==3079== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 1: Resultados de la multiplicación de matrices con 3 bucles.

```
owen@LAP-OWEN:/mnt/e/UNSA/2024B/PARALELA/LAB-MEMCACHE$ valgrind --leak-check=full ./matrix-block
==3080== Memcheck, a memory error detector
==3080== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3080== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3080== Command: ./matrix-block
==3080==
==3080== error calling PR_SET_PTRACER, vgdb might block
Evaluando la multiplicación por bloques con matrices de tamaño 64x64
Tiempo para la multiplicación por bloques: 88429700 segundos
Evaluando la multiplicación por bloques con matrices de tamaño 128x128
Tiempo para la multiplicación por bloques: 694105200 segundos
Evaluando la multiplicación por bloques con matrices de tamaño 256x256
Tiempo para la multiplicación por bloques: 6463309000 segundos
Evaluando la multiplicación por bloques con matrices de tamaño 512x512
Tiempo para la multiplicación por bloques: 60854375100 segundos
==3080==
==3080== HEAP SUMMARY:
==3080==     in use at exit: 0 bytes in 0 blocks
==3080==   total heap usage: 2,907 allocs, 2,907 frees, 4,335,376 bytes allocated
==3080==
==3080== All heap blocks were freed -- no leaks are possible
==3080==
==3080== For lists of detected and suppressed errors, rerun with: -s
==3080== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 2: Resultados de la multiplicación de matrices con 6 bucles (por bloques).

2.2. Análisis del Movimiento de Datos

La multiplicación de matrices clásica presenta una complejidad algorítmica de $O(N^3)$, dado que cada entrada de la matriz resultado C depende de la suma de los productos de N elementos de las matrices A y B. Esta versión no optimizada tiende a generar un mayor número de accesos a la memoria principal, lo que resulta en un uso menos eficiente de la jerarquía de caché.

Por otro lado, la multiplicación de matrices por bloques también presenta una complejidad algorítmica de $O(N^3)$. Sin embargo, al dividir las matrices en bloques de tamaño $b \times b$, el algoritmo optimiza los accesos a caché, ya que los datos de los bloques de las matrices A y B pueden permanecer en caché durante más tiempo, minimizando la cantidad de transferencias de datos entre la memoria principal y la caché. Esto reduce los tiempos de acceso a memoria y mejora la eficiencia global del algoritmo, especialmente en matrices de gran tamaño.

2.3. Resultados en QCallGrind

```
owen@LAP-OWEN:/mnt/e/UNSA/2024B/PARALELA/LAB-MEMCACHE$ valgrind --tool=callgrind --cache-sim=yes ./matrix-classic
==3090== Callgrind, a call-graph generating cache profiler
==3090== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3090== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3090== Command: ./matrix-classic
==3090==
--3090-- warning: L3 cache found, using its data for the LL simulation.
==3090== For interactive control, run 'callgrind_control -h'.
==3090== error calling PR_SET_PTRACER, vgdb might block
Evaluando la multiplicación clásica con matrices de tamaño 64x64
Tiempo para la multiplicación clásica: 0.575778 segundos
Evaluando la multiplicación clásica con matrices de tamaño 128x128
Tiempo para la multiplicación clásica: 6.35901 segundos
Evaluando la multiplicación clásica con matrices de tamaño 256x256
Tiempo para la multiplicación clásica: 51.1539 segundos
Evaluando la multiplicación clásica con matrices de tamaño 512x512
Tiempo para la multiplicación clásica: 546.61 segundos
==3090==
==3090== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==3090== Collected : 19583185436 6776929292 4005523344 3517 194397585 135839 2424 7540 52163
==3090==
==3090== I   refs:      19,583,185,436
==3090== I1  misses:             3,517
==3090== LLi misses:             2,424
==3090== I1  miss rate:           0.00%
==3090== LLi miss rate:           0.00%
==3090==
==3090== D   refs:      10,782,452,636  (6,776,929,292 rd + 4,005,523,344 wr)
==3090== D1  misses:       194,533,424  (  194,397,585 rd +       135,839 wr)
==3090== LLd misses:            59,703  (        7,540 rd +        52,163 wr)
==3090== D1  miss rate:            1.8% (          2.9%   +           0.0%  )
==3090== LLd miss rate:            0.0% (          0.0%   +           0.0%  )
==3090==
==3090== LL refs:          194,536,941  (  194,401,102 rd +       135,839 wr)
==3090== LL misses:             62,127  (        9,964 rd +        52,163 wr)
==3090== LL miss rate:             0.0% (          0.0%   +           0.0%  )
```

Figura 3: callgrind.out de multiplicación clásica

2.4. Enlace a Repositorio

https://github.com/OwenRoque/PARALELA/tree/master/LAB-MEMCACHE

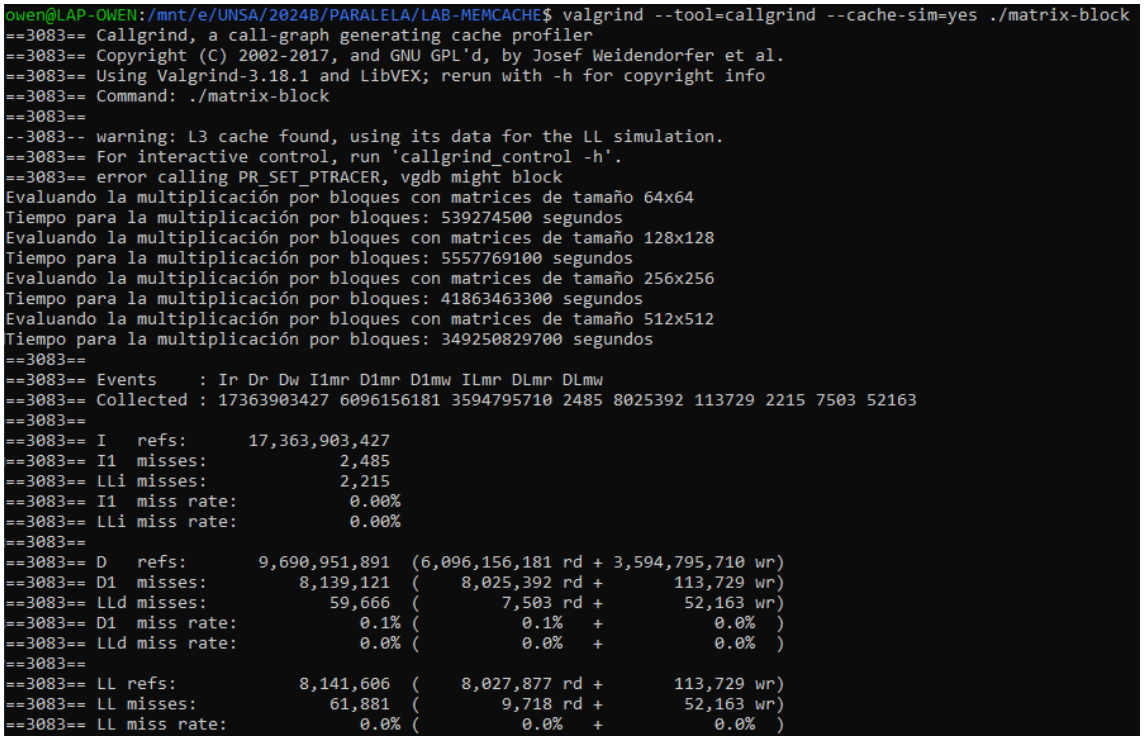

Figura 4: callgrind.out de multiplicación por bloques