PRUEBAS CON LA MEMORIA CACHÉ Computación Paralela y Distribuida

Angel Andres Bejar Merma
Universidad Nacional de San Agustin
abejar@unsa.com
Ciudad de Arequipa

24 de septiembre de 2023

Resumen

El alumno debe realizar un informe en formato artículo (en Latex) donde la implementación, resultados y análisis de la ejecución para los siguientes problemas:

Keywords— Computacion Paralela,Memoria Cache,Valgrind, Matrices

1. Practica 2

1. **Implementar en C/C++ la multiplicación de matrices clásica, la versión de tres bucles anidados y evaluar su desempeño considerando diferentes tamaños de matriz.**
   Las matrices cuadradas N*N [Cormen et al., 2022]se definen con una complejidad de O(n^2), por lo que, la complejidad de multiplicación matrices cuadráticas es de O(n^3) Como se puede ver en la fig 1 mientras mas grande sea la matriz ,mas lento sera el tiempo de ejecución, esto se debe a que una de las matrices se se lee como el primer algoritmo el cual necesita desalojar la memoria de las lineas anteriores y hacer este proceso toma mas tiempo. ya que generara n*n fallos para la segunda matriz y este proceso de alojar y desalojar la memoria demora en tiempo de ejecución.

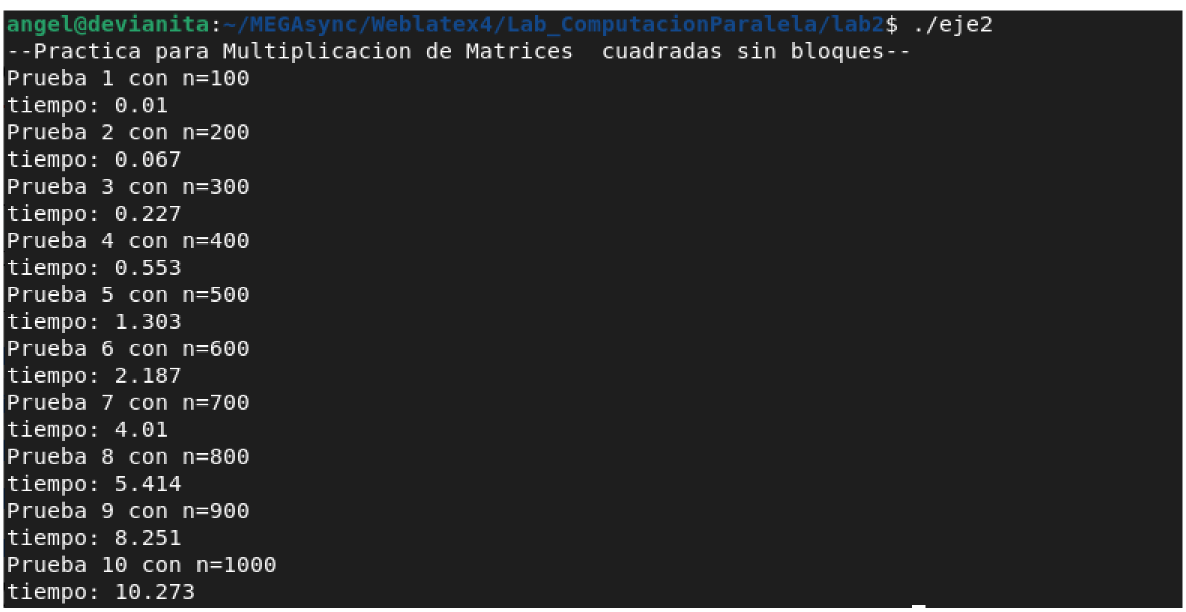


Figura 1: Muestras los tiempos de ejecución para matrices cuadráticas

2. **Implementar la versión por bloques (investigar en internet), seis bucles anidados, evaluar su desempeño y compararlo con la multiplicación de matrices clásica.**

   La matriz por bloques: una matriz por bloques o una matriz particionada es una matriz interpretada, caracterizada por estar dividida en secciones llamadas bloques o submatrices. Una matriz por bloques se puede visualizar como la matriz original con una colección de líneas horizontales y verticales que la dividen, o particionan, en una colección de matrices más pequeñas. Esta técnica se utiliza para reducir los cálculos con matrices; en expansiones de filas y columnas; y en diversas aplicaciones en **ciencias de la computación**, incluido el diseño de chips integrados. Un ejemplo es el algoritmo de Strassen para la multiplicación de matrices rápida, así como la codificación Hamming(7,4) para detección de errores y recuperación de datos en las transmisiones digitales https://es.wikipedia.org/wiki/Matriz_por_bloques.

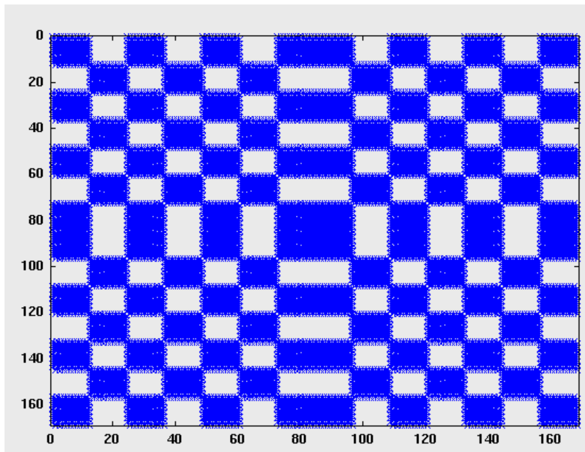


Figura 2: Una matriz por bloques de 168×168 elementos con submatrices de 12x12, 12x24, 24x12 y 24x24. Los elementos distintos de cero están en azul, los elementos cero están en gris

   Las pruebas se realizaron en bloques de tamaño 10, 50, 100 y matrices cuadradas de 100, 200, 300, ..., 2000, esto se puede ver en las figuras 4 6 8 en los cuales se puede observar que sus tiempo de ejecución es mucho mas corto que una multiplicación de matrices normales. Es se debe a que se define el tamaño de bloques, el cual sera cargado en cache, y nos permite hacer que las variables locales se almacenen en registros de tal modo que no se requiere lectura/escritura para acceder a esta variable, que la matriz siempre sera guardada en la RAM y que el bloque siempre se alojara en la cache.Ademas se ve en detalle la comparación 10 .

3. **Ejecutar ambos algoritmos paso a paso, y analizar el movimiento de datos entre la memoria principal y la memoria cache. Hacer una evaluación de acuerdo a la complejidad algorítmica.**

   Matriz A y B N*N con N = 6:

   Para Multiplicación de Matrices normal

   Primero se lee A[0][:] y luego B[:][0] lo cual nos dice que A[0][:] tendra una falla de cache a la hora de leer este dato, mientras que para B[:][0] cometerá N fallas de cache ya que tendrá que desalojar las filas anteriores para alojar una nueva fila, esto generara demoras en tiempo de ejecución. y mientras el nivel de cache sea mayor, esta sera mas lenta.

   Este Proceso ocurre N veces A y para B N*N veces lo cual la localidad de los datos en B es temporal.

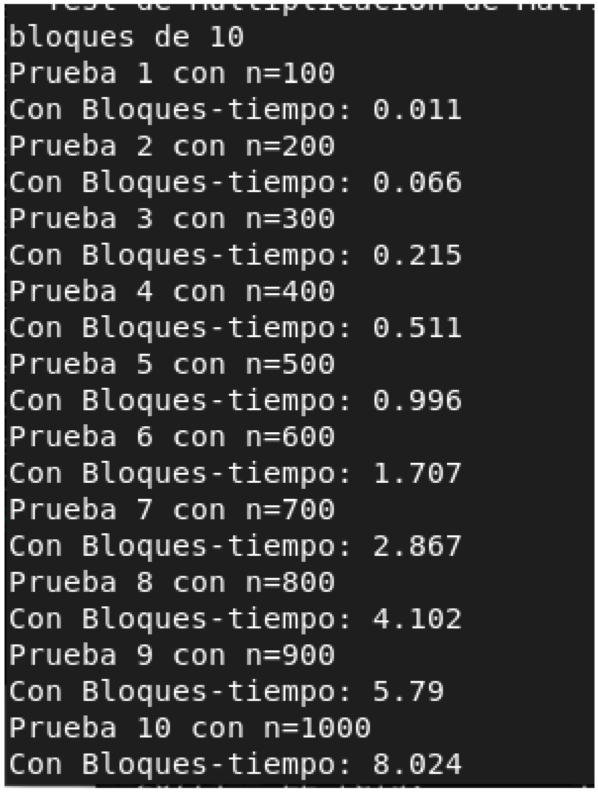
   Para Multiplicación de Matrices por bloques

   El tamaño del bloque se define de manera que solo este bloque se carge en la memoria caché, es decir, de la matriz A y la matriz B solo se carga una parte de bloque ,cada uno de tamaño bsize $*$ bsize, y esto sera multiplicado como un escalar evitando la lectura y escritura para acceder a este bloque.

   Aquí la matriz siempre ira a la RAM mientras que el bloque a la caché, el cual es suficiente para almacenar los bloques, es decir, $|\text{cache}| = \text{bsize}^2 + 2 * \text{bsize}$.

   Por tanto, las referencias a la matriz A tienen una buena ubicación espacial, ya que se accede a cada segmento con un paso de 1. Además, también tienen una buena ubicación temporal ya que todo el segmento está referenciado bsize veces en sucesión. Las referencias a la matriz B, a su vez, tienen una buena localidad temporal porque se accede al bloque de tamaño bsize x bsize n veces sucesivamente.
   Complejidad:

   Tanto la asignación como la liberación de cada matriz ocurre en $O(n^2)$.



```
bloques de 10
Prueba 1 con n=100
Con Bloques-tiempo: 0.011
Prueba 2 con n=200
Con Bloques-tiempo: 0.066
Prueba 3 con n=300
Con Bloques-tiempo: 0.215
Prueba 4 con n=400
Con Bloques-tiempo: 0.511
Prueba 5 con n=500
Con Bloques-tiempo: 0.996
Prueba 6 con n=600
Con Bloques-tiempo: 1.707
Prueba 7 con n=700
Con Bloques-tiempo: 2.867
Prueba 8 con n=800
Con Bloques-tiempo: 4.102
Prueba 9 con n=900
Con Bloques-tiempo: 5.79
Prueba 10 con n=1000
Con Bloques-tiempo: 8.024
```

Figura 3: Tiempos de ejecución para bloques de 10

Multiplicación de Matrices sin Bloques: La multiplicación de las matrices A y B se realiza en O(n^3).
Multiplicación de Matrices por Bloques : Para la multiplicación, hay dos bucles externos que varían los bloques dentro de B de tamaño bsize x bsize. Luego, un bucle i que itera sobre las N filas de A y C, un bucle j para cada columna del bloque B (es decir, bsize veces) y un bucle k para cada elemento del segmento de A y la línea B (bsize veces).

$$\frac{N}{\text{bsize}} * \frac{N}{\text{bsize}} * N * \text{bsize} * \text{bsize} = N^3$$

4. **Ejecutar ambos algoritmos utilizando las herramientas valgrind y kcachegrind para obtener una evaluación mas precisa de su desempeño en términos de cache misses.**

```
Con Bloques-tiempo: 5.79
Prueba 10 con n=1000
Con Bloques-tiempo: 8.024
Prueba 11 con n=1100
Con Bloques-tiempo: 10.925
Prueba 12 con n=1200
Con Bloques-tiempo: 13.543
Prueba 13 con n=1300
Con Bloques-tiempo: 17.29
Prueba 14 con n=1400
Con Bloques-tiempo: 21.432
Prueba 15 con n=1500
Con Bloques-tiempo: 26.308
Prueba 16 con n=1600
Con Bloques-tiempo: 31.995
Prueba 17 con n=1700
Con Bloques-tiempo: 38.426
Prueba 18 con n=1800
Con Bloques-tiempo: 45.501
Prueba 19 con n=1900
Con Bloques-tiempo: 53.228
Prueba 20 con n=2000
Con Bloques-tiempo: 69.761
```

Figura 4: Tiempos de ejecución para bloques de 10

```
bloques de 50
Prueba 1 con n=100
Con Bloques-tiempo: 0.011
Prueba 2 con n=200
Con Bloques-tiempo: 0.063
Prueba 3 con n=300
Con Bloques-tiempo: 0.211
Prueba 4 con n=400
Con Bloques-tiempo: 0.487
Prueba 5 con n=500
Con Bloques-tiempo: 0.961
Prueba 6 con n=600
Con Bloques-tiempo: 1.654
Prueba 7 con n=700
Con Bloques-tiempo: 2.59
Prueba 8 con n=800
Con Bloques-tiempo: 3.949
Prueba 9 con n=900
Con Bloques-tiempo: 5.661
Prueba 10 con n=1000
```

Figura 5: Tiempos de ejecución para bloques de 50

```
Con Bloques-tiempo: 5.661
Prueba 10 con n=1000
Con Bloques-tiempo: 7.563
Prueba 11 con n=1100
Con Bloques-tiempo: 10.063
Prueba 12 con n=1200
Con Bloques-tiempo: 13.069
Prueba 13 con n=1300
Con Bloques-tiempo: 16.713
Prueba 14 con n=1400
Con Bloques-tiempo: 23.436
Prueba 15 con n=1500
Con Bloques-tiempo: 27.063
Prueba 16 con n=1600
Con Bloques-tiempo: 31.152
Prueba 17 con n=1700
Con Bloques-tiempo: 37.688
Prueba 18 con n=1800
Con Bloques-tiempo: 44.322
Prueba 19 con n=1900
Con Bloques-tiempo: 52.391
Prueba 20 con n=2000
Con Bloques-tiempo: 60.972
```

Figura 6: Tiempos de ejecución para bloques de 50

```
bloques de 100
Prueba 1 con n=100
Con Bloques-tiempo: 0.01
Prueba 2 con n=200
Con Bloques-tiempo: 0.064
Prueba 3 con n=300
Con Bloques-tiempo: 0.211
Prueba 4 con n=400
Con Bloques-tiempo: 0.498
Prueba 5 con n=500
Con Bloques-tiempo: 0.974
Prueba 6 con n=600
Con Bloques-tiempo: 1.687
Prueba 7 con n=700
Con Bloques-tiempo: 2.655
Prueba 8 con n=800
Con Bloques-tiempo: 4.027
Prueba 9 con n=900
Con Bloques-tiempo: 5.625
Prueba 10 con n=1000
Con Bloques-tiempo: 7.709
Prueba 11 con n=1100
Con Bloques-tiempo: 10.367
Prueba 12 con n=1200
```

Figura 7: Tiempos de ejecución para bloques de 100

```
-tool=cachegrind ./eje3
==24565== Cachegrind, a cache and branch-prediction profiler
==24565== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==24565== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==24565== Command: ./eje3
==24565==
--24565-- warning: L3 cache found, using its data for the LL simulation.
Test de Multiplicación de Matrices cuadraticas por bloques
bloques de 10
Prueba 1 con n=100
Con Bloques-tiempo: 0.274
Prueba 2 con n=200
Con Bloques-tiempo: 2.178
Prueba 3 con n=300
Con Bloques-tiempo: 7.501
Prueba 4 con n=400
Con Bloques-tiempo: 17.521
```

```
Prueba 10 con n=1000
Con Bloques-tiempo: 7.709
Prueba 11 con n=1100
Con Bloques-tiempo: 10.367
Prueba 12 con n=1200
Con Bloques-tiempo: 13.306
Prueba 13 con n=1300
Con Bloques-tiempo: 17.728
Prueba 14 con n=1400
Con Bloques-tiempo: 24.786
Prueba 15 con n=1500
Con Bloques-tiempo: 29.716
Prueba 16 con n=1600
Con Bloques-tiempo: 33.066
Prueba 17 con n=1700
Con Bloques-tiempo: 38.571
Prueba 18 con n=1800
Con Bloques-tiempo: 46.891
Prueba 19 con n=1900
Con Bloques-tiempo: 52.864
Prueba 20 con n=2000
Con Bloques-tiempo: 61.776
```

Figura 8: Tiempos de ejecución para bloques de 100

```
==24565==
==24565== D   refs:      87,728,958,276  (84,367,121,918 rd   + 3,361,836,358 wr)
==24565== D1  misses:       102,777,518  (   102,752,187 rd   +        25,331 wr)
==24565== LLd misses:        33,775,669  (    33,757,738 rd   +        17,931 wr)
==24565== D1  miss rate:            0.1% (           0.1%     +           0.0%  )
==24565== LLd miss rate:            0.0% (           0.0%     +           0.0%  )
==24565==
==24565== LL refs:          102,779,993  (   102,754,662 rd   +        25,331 wr)
==24565== LL misses:         33,778,029  (    33,760,098 rd   +        17,931 wr)
==24565== LL miss rate:             0.0% (           0.0%     +           0.0%  )
```

Figura 12: Evaluación en términos de cache misses bloque usando kcachegrind

```
--Test Multiplicacion de matrices --
--Matrices sin bloque vs Matrices por bloques
--Bloques de 10--
Test 1 con n=100
Sin Bloques-tiempo: 0.001
Con Bloques-tiempo: 0.001
Test 2 con n=200
Sin Bloques-tiempo: 0.013
Con Bloques-tiempo: 0.012
Test 3 con n=300
Sin Bloques-tiempo: 0.037
Con Bloques-tiempo: 0.041
Test 4 con n=400
Sin Bloques-tiempo: 0.089
Con Bloques-tiempo: 0.093
Test 5 con n=500
Sin Bloques-tiempo: 0.228
Con Bloques-tiempo: 0.191
Test 6 con n=600
Sin Bloques-tiempo: 0.419
Con Bloques-tiempo: 0.321
Test 7 con n=700
Sin Bloques-tiempo: 0.675
Con Bloques-tiempo: 0.526
Test 8 con n=800
Sin Bloques-tiempo: 1.101
Con Bloques-tiempo: 0.786
Test 9 con n=900
Sin Bloques-tiempo: 1.502
Con Bloques-tiempo: 1.084
Test 10 con n=1000
Sin Bloques-tiempo: 2.165
Con Bloques-tiempo: 1.48
Test 11 con n=1100
```

Figura 9: muestra la comparación de los tiempos de ejecución de la multiplicación de matrices

```
-tool=cachegrind ./eje3
==25231== Cachegrind, a cache and branch-prediction profiler
==25231== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==25231== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==25231== Command: ./eje3
==25231==
--25231-- warning: L3 cache found, using its data for the LL simulation.
Test de Multiplicación de Matrices cuadraticas por bloques
bloques de 10
Prueba 1 con n=100
Sin Bloques-tiempo: 0.269
Prueba 2 con n=200
Sin Bloques-tiempo: 2.156
Prueba 3 con n=300
Sin Bloques-tiempo: 7.46
Prueba 4 con n=400
Sin Bloques-tiempo: 17.894
Prueba 5 con n=500
Sin Bloques-tiempo: 37.065
Prueba 6 con n=600
==25231== brk segment overflow in thread #1: can't grow to 0x4840000
==25231== (see section Limitations in user manual)
==25231== NOTE: further instances of this message will not be shown
Sin Bloques-tiempo: 67.7
Prueba 7 con n=700
Sin Bloques-tiempo: 99.709
Prueba 8 con n=800
Sin Bloques-tiempo: 158.074
Prueba 9 con n=900
Sin Bloques-tiempo: 224.192
Prueba 10 con n=1000
Sin Bloques-tiempo: 308.51
==25231==
```

Figura 13: Evaluación en términos de cache misses sin bloque kcachegrind

```
Test 11 con n=1100
Sin Bloques-tiempo: 2.726
Con Bloques-tiempo: 1.939
Test 12 con n=1200
Sin Bloques-tiempo: 3.463
Con Bloques-tiempo: 2.527
Test 13 con n=1300
Sin Bloques-tiempo: 4.623
Con Bloques-tiempo: 3.161
Test 14 con n=1400
Sin Bloques-tiempo: 5.415
Con Bloques-tiempo: 4.043
Test 15 con n=1500
Sin Bloques-tiempo: 7.02
Con Bloques-tiempo: 4.877
Test 16 con n=1600
Sin Bloques-tiempo: 9.389
Con Bloques-tiempo: 6.087
Test 17 con n=1700
Sin Bloques-tiempo: 9.984
Con Bloques-tiempo: 7.012
Test 18 con n=1800
Sin Bloques-tiempo: 13.51
Con Bloques-tiempo: 8.426
Test 19 con n=1900
Sin Bloques-tiempo: 16.055
Con Bloques-tiempo: 10.106
Test 20 con n=2000
Sin Bloques-tiempo: 19.654
Con Bloques-tiempo: 11.594
```

Figura 10: muestra la comparación de los tiempos de ejecución de la multiplicación de matrices

```
==25231==
==25231== I   refs:      166,427,952,694
==25231== I1  misses:              2,389
==25231== LLi misses:              2,280
==25231== I1  miss rate:            0.00%
==25231== LLi miss rate:            0.00%
==25231==
==25231== D   refs:       84,734,450,789  (81,704,573,008 rd   + 3,029,877,781 wr)
==25231== D1  misses:      3,688,531,864  ( 3,688,506,523 rd   +        25,341 wr)
==25231== LLd misses:        324,794,976  (   324,777,045 rd   +        17,931 wr)
==25231== D1  miss rate:             4.4% (           4.5%     +           0.0%  )
==25231== LLd miss rate:             0.4% (           0.4%     +           0.0%  )
==25231==
==25231== LL refs:         3,688,534,253  ( 3,688,508,912 rd   +        25,341 wr)
==25231== LL misses:         324,797,256  (   324,779,325 rd   +        17,931 wr)
==25231== LL miss rate:              0.1% (           0.1%     +           0.0%  )
```

Figura 14: Evaluación en términos de cache misses sin bloque kcachegrind

2. Conclusión

El algoritmo está implementado en C++ y alojado en el siguiente https://github.com/ubuangel/Weblatex4/tree/main/Lab_ComputacionParalela Existen muchas formas de implementar la multiplicación de matrices segun la forma de acceder a la memoria, por lo cual cada uno tiene sus propias especificaciones. En esta practica se hizo la multiplicación por bloques y la forma clásica , de lo cual podemos decir que la multiplicaciones por bloques nos ayuda a disminuir la tasa de perdida de cache haciendo mejoras en la localidad espacial y temporal del acceso a la memoria [Pacheco, 2011]

Referencias

[Cormen et al., 2022] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.

[Pacheco, 2011] Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.