Escuela Profesional de Ciencia de la Computación
Algoritmos Paralelos
Laboratorio I

Alumno: Eddy René Cáceres Huacarpuma **Fecha**: 27/03/17

Parte 1:

Implement in C the simple three-nested-loop version of the matrix product and try to evaluate its performance for a relatively large matrix size.

- Implement the blocked version with six nested loops to check whether you can observe a significant gain.

- Execute these algorithms step by step to get a good understanding of data movements between the cache and the memory and try to evaluate their respective complexity in term of distant memory access.

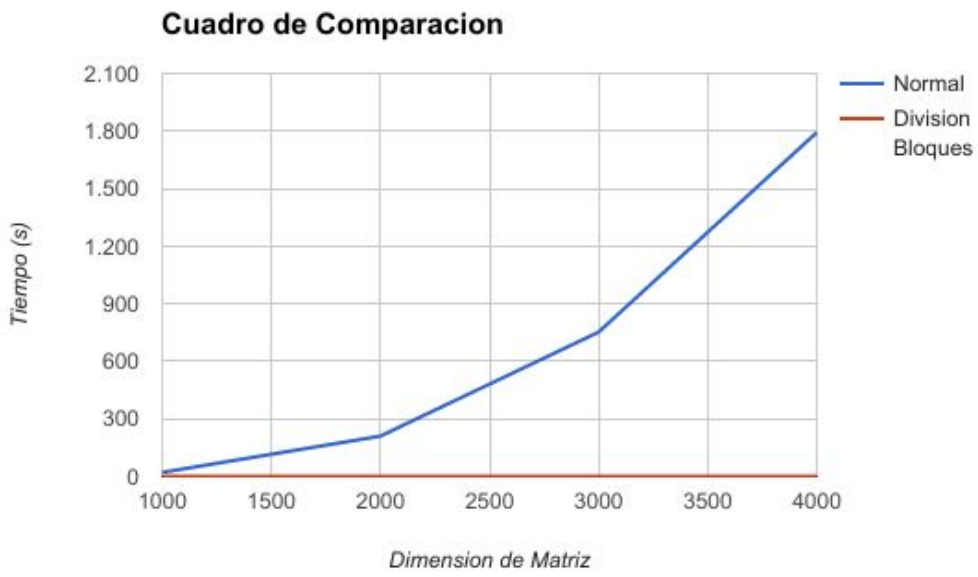
Solución :

Para la primera parte se implementó los dos algoritmos y se hizo las pruebas respectivas bajo las siguientes premisas:

- Las características del equipo utilizado:
  - OS: Deeping Os 15.4, 64 bits - distribución Linux
  - Procesador : Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz x4
  - Memory : 7.67 GB
  -

Level 1 cache size	2 x 32 KB 8-way set associative instruction caches 2 x 32 KB 8-way set associative data caches
Level 2 cache size	**2 x 256 KB 4-way set associative caches**
Level 3 cache size	**3 MB 12-way set associative shared cache**

  -
- El código fue implementado usando la librería **ctime** para calcular es tiempo de las multiplicaciones.
- Se trabajo con matrices cuadradas de 1000, 2000, 3000 y 4000



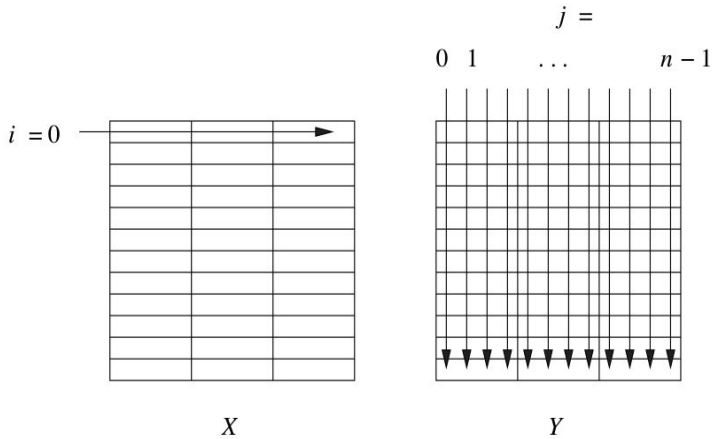
```
Laboratorio
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $ clear

mica@mica-PC ~/Desktop/Paralelos/Laboratorio $ g++ MatMultiplication.cpp -o p
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $ ./p
multiplicacao de matrizes
prueba con matriz de dimension : 1000
1000
terminada 3 en: 21.0332
terminada 6  en: 0.000888
prueba con matriz de dimension : 4000
4000
        terminada 3 en: 1792.87
terminada 6  en: 0.034905
prueba con matriz de dimension : 7000
7000
^C
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $
```

```
Laboratorio
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $ g++ MatMultiplication.cpp -o p
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $ ./p
multiplicacao de matrizes
prueba con matriz de dimension : 2000
terminada 3 en: 210.159
terminada 6  en: 0.006108
prueba con matriz de dimension : 3000
terminada 3 en: 752.246
terminada 6  en: 0.01413
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $
```

Explicación :

Claramente se infiere que es algoritmo de multiplicación normal tiene una complejidad exponencial (O n^3).

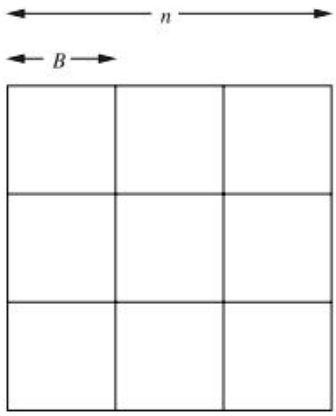
- Se visitan 3*n2 posiciones.
- Fila -> Columna.
- 4 valores = fila caché
  Rectángulo pequeño.
- c=4 n = 12.
- Fila x - > n/c
- n/c fallos (cache miss)
- n2/c fallos en total.
- En Y -> columnas.
- Si cache > n filas . Ok
  - j=c
  - c= n -> n fallos
- Para i=0



    - Entre n2 /c y n 2
- Total
    - n2 / c y n 2/c ,Y <= Cache
    - n2/ c y n 3/ c 1 column <Cache
    - n2/c y n3 , 1 column > cache

Multiplicación en Bloques:

Esta manera de multiplicación es simplemente en la partición imaginaria de la matriz en pequeños bloques, donde se busca que estos bloques pueden ser almacenados temporalmente en memoria caché y así evitar el desplazamiento y perder la información, Como se muestra en la figura de abajo una matriz dividida en bloques de tamaño B.



Debido al orden de los ciclos, en realidad necesitamos cada bloque C en la caché sólo una vez, por lo que no contaremos los fallos de caché debido a C.

El código se encuentra en el repositorio github:

https://github.com/eddyrene/Laboratorio.git

Parte 2 :

- Execute these two versions of the code with valgrind and kcachegrind to get a precise evaluation of their performance in term of cache misses.

Para la prueba de las dos versiones se realizó sobre matrices cuadradas de 500.

Multiplicación normal: que tiene 3 bucles anidados para realizar la multiplicación:

```
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $ valgrind --tool=cachegrind ./deb
==5753== Cachegrind, a cache and branch-prediction profiler
==5753== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==5753== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==5753== Command: ./deb
==5753==
--5753-- warning: L3 cache found, using its data for the LL simulation.
multiplicacao de matrizes
prueba con matriz de dimension : 500
terminada 3 en: 75.9222
==5753==
==5753== I   refs:      15,452,254,015
==5753== I1  misses:             2,012
==5753== LLi misses:             1,905
==5753== I1  miss rate:           0.00%
==5753== LLi miss rate:           0.00%
==5753==
==5753== D   refs:   8,914,545,045  (5,777,531,404 rd   + 3,137,013,641 wr)
==5753== D1  misses:   180,130,547  (  180,033,203 rd   +        97,344 wr)
==5753== LLd misses:        58,513  (        8,485 rd   +        50,028 wr)
==5753== D1  miss rate:        2.0% (          3.1%     +           0.0%  )
==5753== LLd miss rate:        0.0% (          0.0%     +           0.0%  )
==5753==
==5753== LL refs:      180,132,559  (  180,035,215 rd   +        97,344 wr)
==5753== LL misses:         60,418  (       10,390 rd   +        50,028 wr)
==5753== LL miss rate:         0.0% (          0.0%     +           0.0%  )
```

Multiplicación por bloques: que tiene 6 bucles anidados para calcular la multiplicación.

```
mica@mica-PC ~/Desktop/Paralelos/Laboratorio $ valgrind --tool=cachegrind ./deb
==5720== Cachegrind, a cache and branch-prediction profiler
==5720== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==5720== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==5720== Command: ./deb
==5720==
--5720-- warning: L3 cache found, using its data for the LL simulation.
multiplicacao de matrizes
prueba con matriz de dimension : 500
terminada 6  en: 0.022513
==5720==
==5720== I   refs:      78,852,037
==5720== I1  misses:         1,995
==5720== LLi misses:         1,891
==5720== I1  miss rate:       0.00%
==5720== LLi miss rate:       0.00%
==5720==
==5720== D   refs:   39,944,528  (27,361,196 rd   + 12,583,332 wr)
==5720== D1  misses:    115,414  (    18,079 rd   +     97,335 wr)
==5720== LLd misses:     58,177  (     8,168 rd   +     50,009 wr)
==5720== D1  miss rate:     0.3% (       0.1%     +        0.8%  )
==5720== LLd miss rate:     0.1% (       0.0%     +        0.4%  )
==5720==
==5720== LL refs:       117,409  (    20,074 rd   +     97,335 wr)
==5720== LL misses:      60,068  (    10,059 rd   +     50,009 wr)
==5720== LL miss rate:      0.1% (       0.0%     +        0.4%  )
```

Explicación:

- I : Total de accesos a **instrucciones** realizadas.
- D: : Total de accesos a **datos** realizados.
- I1 : Número de accesos a las **instrucciones** en caché L1
- D1 : Número de accesos a **datos** que contiene el cache L1
- LL : **(last-levels caches),** Representa a los demas demas niveles de los caches

- LLi : representa el total de accesos a **instrucciones** en los otros niveles de Caché
- LLd : representa el total de accesos a **datos** en los otros niveles de Caché

“miss” : Son las lecturas hechas donde la información no está disponible, una lectura que no encontró la instrucción o dato necesario para continuar la instrucción.

Revisión KcacheGrind:

Con la ayuda de esta herramienta podemos analizar de manera más precisa los *cache miss* encontrados :

L1 Data Read Miss:
Es análisis realizado se basa en en número de accesos a la caché primaria (L1), debido a que es la memoria estática integrada en es núcleo del procesador que es la más rápida y se utiliza para almacenar información que fue recientemente accedida por es procesador.

Método Normal

```
#    D1mr          D1mw  Source
 0                      --- From '/home/mica/Desktop/Paralelos/Laboratorio/MatMultiplication.cpp' ---
86                                  for(int j=0;j<t;j++)
87                                      for(int k=0;k<t;k++)
88                                      {
89  133 015 501                             C.v[i][j]+=A.v[i][k]*B.v[k][j];
90                                          //cout<<"entra"<<endl;
91                                      }
92                                  //cout<<"se termino con exito"<<endl;
93                                  clock_t end = clock();
94              1                   double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
95              1                   cout<<"terminada 3 en: "<< time_spent<<endl;
96
97                              }
98                              void m6bucles()
```

Método en Bloques

```
#     D1mr  D1mw  Source
  0                --- From '/home/mica/Desktop/Paralelos/Laboratorio/MatMultiplication.cpp' ---
 98                      void m6bucles()
 99                      {
100                          int N=t;
101       1              double SM=32;
102                      clock_t begin= clock();
103                      for (int i1 = 0; i1 < N/SM; i1 += SM)
104                          for (int j1 = 0; j1 < N/SM; j1 += SM)
...                  ...
106                                  for (int i = i1; i<i1+SM&&i<N;i++)
107                                      for     (int j=j1;j<j1+SM&&j<N;j++)
108                                          for     (int k=k1;k<k1+SM&&k<N;k++)
109   256                                           C.v[i][j]+=A.v[i][k]*B.v[k][j];
110
111
```

Conclusiones:

Se concluye de esta parte que :

- Para la llevar a programación de manera eficiente es importante tener en cuenta la arquitectura del computador.
- La multiplicación de matrices con bloques, aprovecha la localidad espacial
- Se muestra que con la medición del tiempo , que la multiplicación para matrices grandes , es algoritmo de multiplicación de matrices regular es más lento que es de bloques.
- Se muestra con ayuda de las herramientas **valgrind & kcachegrind ,** que la lentitud del algoritmo normal se debe a la inmensa cantidad de ***cache miss*** producidos.

Referencias Técnicas :

http://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5-6200U%20Mobile%20processor.html