

# Informe Laboratorio 1

## Algoritmos Paralelos

Lorena Xiomara Castillo Galdos

Marzo 2017

### Contents

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Multiplicación de matrices versión simple</b>	<b>2</b>
<b>3</b>	<b>Multiplicación de matrices versión por bloques</b>	<b>3</b>
<b>4</b>	<b>Experimentos</b>	<b>3</b>
4.1	Análisis . . . . .	4
<b>5</b>	<b>Conclusiones</b>	<b>5</b>

## 1 Introducción

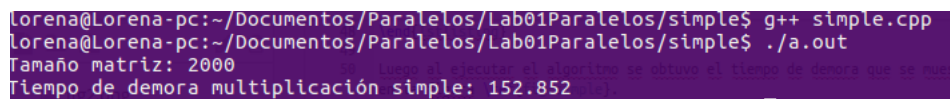
El objetivo de ésta práctica de laboratorio es implementar las versiones simple y de bloques de la multiplicación de matrices y evaluar el movimiento de datos entre la cache y la memoria para una matriz relativamente larga, utilizando el software Valgrind y la herramienta cachegrind que realiza la simulación de cache capturando los accesos a ésta.

## 2 Multiplicación de matrices versión simple

Algoritmo del producto de matrices simple implementado con 3 bucles.

```
void simpleMult(int **&A, int **&B,
               int **&C, int R_A,
               int C_A, int R_B, int C_B)
{
    if(C_A == R_B)
    {
        for(int i=0; i<R_A; i++)
        {
            for(int j=0; j<C_B; j++)
            {
                for(int k=0; k<C_A; k++)
                {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}
```

Luego al ejecutar el algoritmo en matrices de tamaño 2000 se obtuvo el tiempo de demora que se muestra en la figura 1.



```
lorena@lorena-pc:~/Documentos/Paralelos/Lab01Paralelos/simple$ g++ simple.cpp
lorena@lorena-pc:~/Documentos/Paralelos/Lab01Paralelos/simple$ ./a.out
Tamaño matriz: 2000
Tiempo de demora multiplicación simple: 152.852
```

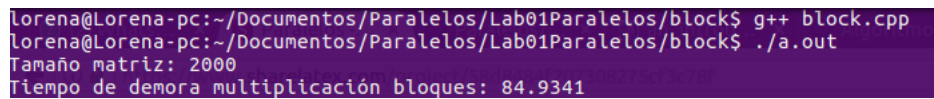
Figure 1: Tiempo de demora multiplicación simple.

### 3 Multiplicación de matrices versión por bloques

Algoritmo del producto de matrices por bloques implementado con 6 bucles.

```
void blockMult(int **&A, int **&B, int **&C,
               int sizeMatrix, int sizeBlock)
{
    for(int i=0; i<sizeMatrix; i+=sizeBlock)
    {
        for(int j=0; j<sizeMatrix; j+=sizeBlock)
        {
            for(int k=0; k<sizeMatrix; k+=sizeBlock)
            {
                for(int i_=i; i_<i+sizeBlock; ++i_)
                {
                    for(int j_=j; j_<j+sizeBlock; ++j_)
                    {
                        for(int k_=k; k_<k+sizeBlock; ++k_)
                        {
                            C[i_][j_]+=A[i_][k_]*B[k_][j_];
                        }
                    }
                }
            }
        }
    }
}
```

Luego al ejecutar el algoritmo en matrices de tamaño 2000 se obtuvo el tiempo de demora que se muestra en la figura 2.



```
lorena@Lorena-pc:~/Documentos/Paralelos/Lab01Paralelos/block$ g++ block.cpp
lorena@Lorena-pc:~/Documentos/Paralelos/Lab01Paralelos/block$ ./a.out
Tamaño matriz: 2000
Tiempo de demora multiplicación bloques: 84.9341
```

Figure 2: Tiempo de demora multiplicación por bloques.

### 4 Experimentos

A continuación se muestran los resultados luego de analizar los códigos con las herramientas Valgrind y cachegrind después de ejecutar los algoritmos de multiplicación simple y por bloques. Los resultados se muestran en las figuras 3 y 4.

```

==20597== I   refs:          78,107,621,173
==20597== I1  misses:         1,467
==20597== LLi misses:         1,402
==20597== I1  miss rate:         0.00%
==20597== LLi miss rate:         0.00%
==20597==
==20597== D   refs:          38,286,278,019 (36,812,849,584 rd + 1,473,428,435 wr)
==20597== D1  misses:         1,674,186,546 ( 1,673,676,465 rd +      510,081 wr)
==20597== LLd misses:         1,407,616,047 ( 1,407,107,158 rd +      508,889 wr)
==20597== D1  miss rate:         4.4% (      4.5% +      0.0% )
==20597== LLd miss rate:         3.7% (      3.8% +      0.0% )
==20597==
==20597== LL refs:          1,674,188,013 ( 1,673,677,932 rd +      510,081 wr)
==20597== LL misses:         1,407,617,449 ( 1,407,108,560 rd +      508,889 wr)
==20597== LL miss rate:         1.2% (      1.2% +      0.0% )

```

Figure 3: Resultados multiplicación simple.

```

==23089== at 0x400C6E: blockMult(int**amp, int**amp, int**amp, int, int) (in /home/lorena/Documentos/Paralelos/Lab01Paralelos/block/a.out)
==23089== by 0x400E1D: main (in /home/lorena/Documentos/Paralelos/Lab01Paralelos/block/a.out)
==23089==
==23089== I   refs:          46,913,814,685
==23089== I1  misses:         1,466
==23089== LLi misses:         1,404
==23089== I1  miss rate:         0.00%
==23089== LLi miss rate:         0.00%
==23089==
==23089== D   refs:          22,995,184,459 (22,115,891,372 rd + 879,293,087 wr)
==23089== D1  misses:         55,095,439 ( 54,583,064 rd +      512,375 wr)
==23089== LLd misses:         973,192 ( 462,008 rd +      511,184 wr)
==23089== D1  miss rate:         0.2% (      0.2% +      0.1% )
==23089== LLd miss rate:         0.0% (      0.0% +      0.1% )
==23089==
==23089== LL refs:          55,096,905 ( 54,584,530 rd +      512,375 wr)
==23089== LL misses:         974,596 ( 463,412 rd +      511,184 wr)
==23089== LL miss rate:         0.0% (      0.0% +      0.1% )

```

Figure 4: Resultados multiplicación por bloques.

## 4.1 Análisis

A continuación en la tabla 1 se muestra la comparación de los resultados.

Table 1: Análisis de ambos códigos

	Simple	Bloques
Instrucciones ejecutadas (I refs)	78,107,621,173	46,913,814,685
Lecturas de memoria (D refs)	38,286,278,019	22,995,184,459
Cache Misses %	4.4%	0.0%
Lectura de datos de cache (LL refs)	1,674,188,013	55,096,905
Cache Misses % (Datos)	1.2%	0.0%

## 5 Conclusiones

Los resultados nos muestran que el código de la versión de multiplicación por bloques tiene menor número de instrucciones a ejecutar que la versión simple. Pero al comparar las lecturas de memoria cache, la versión por bloques tiene una gran ventaja al momento de su ejecución.