Journal of Basic Sciences, Vol. 3 (**7**), pp. 1-11, enero-abril 2017

JOURNAL OF BASIC SCIENCES

MulCoMa un algoritmo concurrente para multiplicar matrices

Roberto Enrique Alberto Lira[1,*], Abel Cortazar May[1], Rubén Armando González García[1], Víctor Alberto Gómez Pérez[2], Carlos de Jesús Alberto Lira[1]

[1] *División Académica de Ciencias Básicas, Universidad Juárez Autónoma de Tabasco, Carret. Cunduacán-Jalpa km 1, A.P. 24, C.P. 86690, Cunduacán, Tabasco, México*

[2] *Instituto de informática, Universidad de la Sierra Sur, Calle Guillermo Rojas Mijangos s/n, Esq. Av. Universidad, Col. Ciudad Universitaria, Miahuatlán de Porfirio Díaz, C.P. 70800, Oaxaca, México*

**roberto.alberto@ujat.mx*

Resumen

En este artículo se presenta a MulCoMa (Multiplicación Concurrente de Matrices), un algoritmo concurrente para multiplicar matrices. Para mostrar el desempeño de MulCoMa, se realiza un análisis comparativo con otros algoritmos secuenciales (i.e., algoritmo tradicional para multiplicar matrices y algoritmo Strassen) y concurrentes (i.e., algoritmo tradicional concurrente) que multiplican matrices. Como medida de desempeño se considera principalmente el tiempo de ejecución y el diseño de los algoritmos.

Palabras claves: Algoritmo concurrente, algoritmo secuencial, multiplicación de matrices, desempeño.

Abstract

This article shows the MulCoMa, it refers to a concurrent algorithm for multiplying matrices. To verify the performance of MulCoMa, It was necessary to make a comparative analysis with other sequential (i.e., traditional algorithm for multiplying matrices and Strassen algorithm) and concurrent (traditional concurrent algorithm) algorithms to multiply matrices. As a performance measure is mainly considered the runtime and the design of the algorithms.

Keywords: Concurrent algorithm, sequential algorithm, matrix multiplication performance.

Recibido: 11 octubre 2016. Aceptado: 08 febrero 2017. Publicado: 30 abril 2017.

1. Introducción

En el análisis de rendimiento es importante conocer cuál es el elemento básico de procesamiento. Hasta no hace muchos años, el elemento básico de procesamiento ha sido una CPU (*Central Processing Unit*). Es importante tener claro que las computadoras de escritorio actuales, ya no tienen una única CPU. Las nuevas arquitecturas de procesadores, que integran múltiples "*cores*" o núcleos en un procesador físico [3, 7] han producido un notorio impacto en el desarrollo de la Informática.

Se denomina medida al valor obtenido mediante un instrumento de medición confiable. Para un programador que está resolviendo un problema mediante diversos algoritmos, el algoritmo con mejor desempeño es aquel que resuelva dicho problema en el menor tiempo. Se evidencia entonces que, la medida del rendimiento del algoritmo es el tiempo.

En el presente artículo compararemos mediante el caso de estudio de la multiplicación de matrices, el desempeño de diversos algoritmos, los cuales se clasifican en los paradigmas de programación concurrente y secuencial. Estos algoritmos han sido probados en un equipo con las siguientes características:

Equipo: HP-Pavilion-14-Notebook-PC
S.O: Ubuntu, 14.04 LTS

JOURNAL OF BASIC SCIENCES

Tipo de S.O: 64 bits
Procesador: AMD A8-6410 con 4 núcleos
Memoria Ram: 6.7 GB
Disco: 131.9 GB
Lenguaje de programación: C de Linux
Compilador: gcc 4.8.4

Es importante mencionar que se utilizará como caso de estudio la multiplicación de matrices inicialmente por sencillez. Por otro lado, la multiplicación de matrices es la base de los problemas de rendimiento en aplicaciones de álgebra lineal [11]. Conviene destacar que la evaluación de los algoritmos mostrados en el presente trabajo se analizan desde la perspectiva de eficiencia en cuanto a tiempo de ejecución.

Este trabajo se encuentra organizado de la siguiente manera: Primeramente, se da una definición de la multiplicación de matrices (cf. Sección 2), la cual es el instrumento empleado para realizar el análisis de desempeño de los algoritmos. Posteriormente, se presenta los trabajos relacionados (cf. Sección 3) a la presente investigación. A continuación, se muestra nuestra propuesta denominado MulCoMa (cf. Sección 4), el cual es un algoritmo concurrente para multiplicar matrices. Seguidamente, se realiza una descripción del experimento realizado (cf. Sección 5), con los resultados obtenidos (cf. Sección 6). Finalmente, se presentas las conclusiones (cf. Sección 7) de la presente investigación.

2. Multiplicación de matrices

En esta sección se presenta la definición de la multiplicación de matrices, la cual es caso de estudio que nos permitirá comparar el desempeño de los diversos algoritmos presentados en las secciones posteriores.

Producto de dos matrices [4]. Sea $A = (a_{ij})$ una matriz de tamaño $m \times n$, y sea $B = b_{ij}$ una matriz de tamaño $n \times p$. Entonces el producto de A y B es una matriz $m \times p$, $C = (c_{ij})$, en donde

c_{ij}= (renglón i de A)·(columna j de B)

Es decir, el elemento ij de AB es el producto punto del renglón i de A y la columna j de B. Si esto se extiende, se obtiene:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

Si el número de columnas de A es igual al número de renglones de B, entonces se dice que A y B son **compatibles bajo la multiplicación**.

Un hecho clave es que el tamaño de los vectores involucrados en el cálculo de cada entrada c_{ij} de la matriz $C = (c_{ij})$ es siempre el mismo, haciendo que el número de multiplicaciones y sumas que se efectúen para obtener cada una de las entradas sea idéntico [8]. Esto, aunado al hecho de que la matriz resultante $C = (c_{ij})$ es de tamaño $m \times p$ (es decir se calculan $m \times p$ entradas), permite concluir que para obtener la matriz $C = (c_{ij})$ el número total de multiplicaciones y sumas que se requieren son, $(n)(mp)$ y $(n-1)(mp)$ respectivamente.

3. Trabajos relacionados

En esta sección se describen diversos algoritmos que resuelven el problema de la multiplicación de matrices [2]. Cabe aclarar que los dos primeros algoritmos presentados pertenecen al paradigma de programación secuencial. El último algoritmo que se muestra, pertenece al paradigma de programación concurrente.

Algoritmo tradicional

El algoritmo tradicional [1] consiste en recorrer cada fila de A y cada columna de B, obtener el producto punto de la fila i en A y columna j en B, el resultado es asignado a la posición i,j en C. Esto toma tiempo $O(N^3)$, el algoritmo 1 se muestra a continuación:

JOURNAL OF BASIC SCIENCES

Algoritmo 1 Multiplicación de matrices tradicional

```
1: para i=0 hasta m hacer
2:   para z=0 hasta p hacer
3:     s ← 0
4:     para j=0 hasta n hacer
5:       s ← A[i][j] × B[j][z] + s
6:     fin para
7:     C[i][z] ← s
8:   fin para
9: fin para
```

Algoritmo Strassen

Strassen[1] propuso en 1969 una forma de calcular el producto de matrices basada en la estrategia divide y conquista, pero en la que solamente se necesitan resolver 7 subproblemas [9]. La propuesta es la siguiente:

Sean A, B dos matrices cuadradas sobre un anillo R. Queremos calcular la matriz C como producto

$$C = AB \qquad A, B, C \in R^{2^n \times 2^n}$$

Partimos A, B y C en matrices de igual tamaño de bloque

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Con

$$A_{i,j}, B_{i,j}, C_{i,j} \in R^{2^n \times 2^n}$$

Entonces

$$M_1 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_2 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_3 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$M_4 = (A_{11} + A_{12})B_{22}$$

$$M_5 = A_{11}(B_{12} - B_{22})$$

$$M_6 = A_{22}(B_{21} - B_{11})$$

$$M_7 = (A_{21} + A_{22})B_{11}$$

[1] https://es.wikipedia.org/wiki/Algoritmo_de_Strassen.

La composición de la solución final está dada por:

$$C_{11} = M_1 + M_2 - M_4 + M_6$$

$$C_{12} = M_4 + M_5$$

$$C_{21} = M_6 + M_7$$

$$C_{22} = M_2 - M_3 + M_5 - M_7$$

Esta solución tiene un tiempo recursivo de $T(N) = 7T\left(\frac{N}{2}\right) + O(N^2)$ cuyo orden es $O(7^{\log_2 N}) \approx O(N^{2.81})$. La complejidad asintótica de este algoritmo es ligeramente menor que la del algoritmo tradicional para multiplicar matrices.

Programa tradicional concurrente

Para la programación del algoritmo concurrente [5, 6], utilizamos hilos de POSIX [10], ya que como previamente se mencionó, representan menor *overhead* para el sistema que los procesos. En este algoritmo, cada hilo se encarga de generar una fila de la matriz resultante.

Algoritmo 2 Principal

```
1: Solicitar memoria para m hilos
2: Solicitar memoria para las matrices A, B y C
3: Llenar con números aleatorios entre 1 y 3 las
   matrices A y B
4:
5: /*Creación de hilos*/
6: para j=0 hasta m − 1 hacer
7:    Crear hilo
8:    idHilo ← j
9:    Indicarle al hilo idHilo que ejecute el código
      de la función multiplica(entero : idHilo)
10: fin para
11:
12: Esperar a que los m hilos se terminen de
    ejecutar
13: Liberar memoria solicitada
```

Con la finalidad de que los hilos creados tengan acceso a los datos (i.e., matriz A, matriz B, matriz C, m, n y p), estos se declaran de forma global en el programa, el cual contiene dos funciones:

- Principal
- Multiplica

La función **Principal** (cf., Algoritmo 2) se encarga de crear m (filas de la matriz C) hilos y esperar hasta que todos finalicen su tarea asignada.

La función **Multiplica** (cf., Algoritmo 3) es accedida por los hilos creados, a los cuales se les asigna un identificador **idHilo**. Recordemos que se crean m (i.e., filas de A y C) hilos. Cada hilo se encarga de obtener una fila de la matriz resultante (matriz C). El **idHilo** le indica al hilo que fila de la matriz resultante debe generar, e.j., hilo con **idHilo**=1,

le corresponde generar la fila 1. Gracias a este **idHilo** se consigue que los hilos no compitan por ingresar a secciones críticas, ya que cada hilo se enfoca en generar la fila que le corresponde.

Algoritmo 3 Multiplica

Entrada: $idHilo$.
Los datos: p (columnas de B), n (columnas de A), matriz A, matriz B, matriz C son globales al programa.

```
1: i ← idHilo
2: para r=0 hasta p − 1 hacer
3:   para j=0 hasta n − 1 hacer
4:     C[i][r] ← C[i][r] + (A[i][j] * B[j][r])
5:   fin para
6: fin para
7: Finalizar ejecución de hilo idHilo
```

4. MulCoMa

Para la programación MulCoMa, utilizamos hilos de POSIX [10]. En este algoritmo, cuando m (i.e., filas de A) es par, cada hilo se encarga de generar dos filas de la matriz resultante. Cuando m es impar, se aplica la misma estrategia, i.e., cada hilo genera dos filas de la matriz resultante hasta la fila $m-1$. Posteriormente, un hilo se encarga de generar la última fila de la matriz, i.e., la fila m.

Con la finalidad de que los hilos creados tengan acceso a los datos (i.e., matriz A, matriz B, matriz C, m, n y p), y este acceso sea eficiente en cuanto a administración de recursos (e.g., memoria compartida, tuberías, cola de mensajes, variables locales, etc), declaramos las matrices de forma global al programa, el cual está integrado por dos funciones:

- Principal
- Multiplica

La función **Principal** (cf., Algoritmo 4) se encarga de crear $m/2$ hilos cuando m es par y $\frac{m}{2}+1$, cuando m es impar. Otra de sus funciones consiste en esperar a que todos los hilos finalicen su tarea asignada.

La función **Multiplica** (cf., Algoritmo 5) es accedida por los hilos creados, a los cuales se les asigna un identificador (**id**) i. Cada hilo se encarga de generar dos filas de la matriz resultante, cuando m es impar el último hilo genera la última fila. El **id** le indica al hilo que filas de la matriz resultante debe generar (e. g., hilo con **id**= 1, le corresponde generar las filas 1 y 2).

Algoritmo 4 Principal

```
1: j ← 0
2: si ((m mod 2) = 0) entonces
3:   mm ← m/2
4: si no
5:   mm ← (m/2 + 1)
6: fin si
7: Solicitar memoria para mm hilos
8: Solicitar memoria para las matrices A, B y C
9: Llenar con números aleatorios entre 1 y 3 las
   matrices A y B
10:
11: /*Creación de hilos*/
12: mientras j < m hacer
13:   Crear hilo
14:   id ← j
15:   Indicarle al hilo id que ejecute el código de
      la función multiplica(entero : id)
16:   j ← j + 2
17: fin mientras
18:
19: Esperar a que los mm hilos se terminen de
    ejecutar
```

JOURNAL OF BASIC SCIENCES

Algoritmo 5 Multiplica

Entrada: id del hilo. Los datos: m (filas de A), n (columnas de A), p (columnas de B), matriz A, matriz B, matriz C son globales al programa.

```
1: i ← id
2: /*condición para generar filas pares*/
3: si ((i + 1) < m) entonces
4:    para r = 0 hasta p − 1 hacer
5:       para j = 0 hasta n − 1 hacer
6:          C[i][r] ← C[i][r] + (A[i][j] * B[j][r])
7:          C[i + 1][r] ← C[i + 1][r] + (A[i + 1][j] *
            B[j][r])
8:       fin para
9:    fin para
10: si no
11:   /*accedido para generar la última fila de C,
      cuando m es impar*/
12:   para r = 0 hasta p − 1 hacer
13:      para j = 0 hasta n − 1 hacer
14:         C[i][r] ← C[i][r] + (A[i][j] * B[j][r])
15:      fin para
16:   fin para
17: fin si
18:
19: Finalizar ejecución de hilo id
```

5. Descripción del experimento

El experimento consistió en multiplicar matrices de forma concurrente y secuencial, midiendo el tiempo de ejecución de cada uno de los algoritmos en las diversas combinaciones que se realizaron. Estas combinaciones están agrupadas en tres casos. Cada caso tiene ocho combinaciones (i. e, 2^3), ya que consideramos total de filas de matriz A, total de columnas de matriz A y total de columnas de matriz B (son los datos requeridos para que se puedan multiplicar dos matrices). En el primer caso el tamaño máximo es 1000 y el mínimo es 10, en el segundo caso, el tamaño máximo es 5000 y el mínimo es 10. En el tercer caso, el tamaño máximo es 10000 y el mínimo 10. La finalidad de alternar la posición (es decir, fila A, columna A, columna B) de los tamaños máximos en cada una de las combinaciones, consiste en formar grupos como en binario para abarcar todos los casos posibles. Es decir, 2^3nos genera ocho combinaciones de números binarios (000,001...), en donde el 1 nos representaría el tamaño máximo y el 0 nos representaría el tamaño mínimo.

6. Resultados

Para interpretar las lecturas de las tablas, consideramos como ejemplo una celda que contiene de valor 0m0.003s, entonces este dato se leerá como: 0 minutos, 0.003segundos.

Dimensiones de las matrices A y B (m, n y p)	**Algoritmo de Strassen**	**Algoritmo tradicional**	**Algoritmo Concurrente tradicional**	**MulCoMa**
10X10X10	0m0.003s	0m0.002s	0m0.005s	0m0.003s
10X10X1000	No aplica	0m0.005s	0m0.004s	0m0.004s
10X1000X10	No aplica	0m0.006s	0m0.005s	0m0.005s
10X1000X1000	No aplica	0m0.245s	0m0.117s	0m0.104s
1000X10X10	No aplica	0m0.004s	0m0.056s	0m0.029s
1000X10X1000	No aplica	0m0.086s	0m0.089s	0m0.063s
1000X1000X10	No aplica	0m0.117s	0m0.111s	0m0.075s
1000X1000X1000	0m11.048s	0m20.979s	0m8.455s	0m6.272s

Tabla 1. Caso 1

En el caso 1 (cf., Tabla 1) se observa que de las ocho combinaciones presentadas, en las tres primeras y en la seis y siete, todos los algoritmos (a excepción de *Strassen*, que solo aplica en dos de las ocho combinaciones por ser exclusivo para matrices cuadradas) tienen un desempeño similar. Por su parte, se nota una mejora en el desempeño de los algoritmos concurrentes cuando la fila de A tiene el tamaño mínimo (i.e., 10) y las columnas de A y B tienen el tamaño máximo (i.e., 1000). Se visualiza también que cuando la fila de A tiene el tamaño máximo y las columnas de A y B tienen el tamaño mínimo, entonces, tiene mejor desempeño el algoritmo tradicional. Notamos que, en la última combinación, i.e., cuando las filas y columnas de A y B tienen el tamaño máximo, los algoritmos concurrentes tienen mejor desempeño (cf. Figura 1 y Figura 2).

De acuerdo a los datos recabados, para este primer caso, nuestra propuesta presentó mejor desempeño que el algoritmo concurrente tradicional en la mayoría de las combinaciones. Por su parte, el algoritmo de *Strassen* tuvo mejor desempeño que el algoritmo tradicional cuando las filas y las columnas de A y B tuvieron el tamaño máximo.

Dimensiones de las matrices A y B (m, n y p)	**Algoritmo de Strassen**	**Algoritmo tradicional**	**Algoritmo Concurrente tradicional**	**MulCoMa**
10X10X10	0m0.002s	0m0.002s	0m0.004s	0m0.002s
10X10X5000	No aplica	0m0.012s	0m0.010s	0m0.011s
10X5000X10	No aplica	0m0.015s	0m0.013s	0m0.014s
10X5000X5000	No aplica	0m6.548s	0m5.068s	0m3.516s
5000X10X10	No aplica	0m0.015s	0m0.240s	0m0.111s
5000X10X5000	No aplica	0m1.981s	0m2.296s	0m2.256s
5000X5000X10	No aplica	0m2.669s	0m1.700s	0m1.496s
5000X5000X5000	35m5.244s	38m39.110s	31m4.041s	22m8.759s

Tabla 2. Caso 2

En el caso 2 (cf., Tabla 2) se observa que en las tres primeras combinaciones, i.e., cuando la fila de A tiene el tamaño mínimo (i.e., 10), el desempeño de todos los algoritmos es similar. Sin embargo, en la cuarta combinación, que también la fila de A tiene el tamaño mínimo, pero las columnas de A y B tienen el máximo (i.e., 5000), se observa un mejor desempeño en los algoritmos concurrentes; en particular, en MulCoMa. En las combinaciones cinco y seis se observa un mejor desempeño en el algoritmo tradicional. Por su parte, en las dos últimas combinaciones se observa un mejor desempeño en los algoritmos concurrentes (cf. Figura 1 y Figura 2).

En la última combinación, i.e., cuando las filas y columnas tienen el tamaño máximo el algoritmo *Strassen* tiene mejor desempeño que el algoritmo tradicional. También observamos que, MulCoMa sigue mostrando mejor desempeño que el algoritmo concurrente tradicional en la mayoría de las combinaciones.

Dimensiones de las matrices A y B (m, n y p)	**Algoritmo de Strassen**	**Algoritmo Secuencial**	**Algoritmo Concurrente tradicional**	**MulCoMa**
10X10X10	0m0.002s	0m0.003s	0m0.005s	0m0.005s
10X10X10000	No aplica	0m0.019s	0m0.013s	0m0.017s
10X10000X10	No aplica	0m0.022s	0m0.019s	0m0.017s
10X10000X10000	No aplica	0m47.779s	0m40.766s	0m24.258s
10000X10X10	No aplica	0m0.017s	0m0.454s	0m0.221s
10000X10X10000	No aplica	0m7.858s	0m9.302s	0m8.366s
10000X10000X10	No aplica	0m10.609s	0m5.959s	0m5.689s
10000X10000X10000	236m2.878s	259m0.011s	546m22.694s	298m35.214s

Tabla 3. Caso 3

Al igual que en el caso 2, en el caso 3 (cf., Tabla 3) se observa un desempeño similar de los algoritmos en las tres primeras combinaciones, i.e., cuando las filas de A tienen el tamaño mínimo (i.e., 10). Se sigue visualizando un mejor

desempeño de los algoritmos concurrentes cuando la fila de A tiene el tamaño mínimo y las columnas de A y B tienen el tamaño máximo (i.e., 10000). En particular, en esta cuarta combinación, se observa un desempeño sobresaliente con MulCoMa. De igual forma que en los casos anteriores, se observa un mejor desempeño en el algoritmo tradicional sobre los concurrentes en las combinaciones cinco y seis. En cuanto a la combinación siete, notamos un mejor desempeño en los algoritmos concurrentes. Para la última combinación, ya no se sigue manteniendo el patrón de los dos casos anteriores, ya que ahora, los algoritmos secuenciales muestran un mejor desempeño en comparación con los algoritmos concurrentes (cf. Figura 1 y Figura 2).

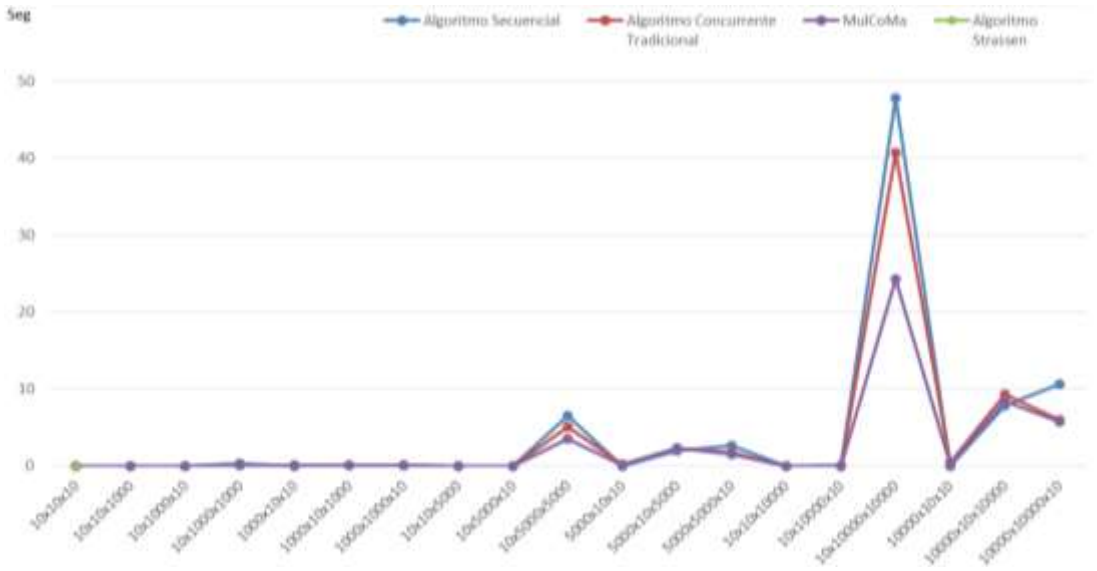

Figura 1. Gráfica del desempeño de los algoritmos en segundos

Se observa que MulCoMa sigue siendo más eficiente que el algoritmo concurrente tradicional en la mayoría de las combinaciones. Por su parte, el algoritmo *Strassen* se mantiene como más eficiente que el algoritmo tradicional, cuando la combinación tiene los valores máximos.

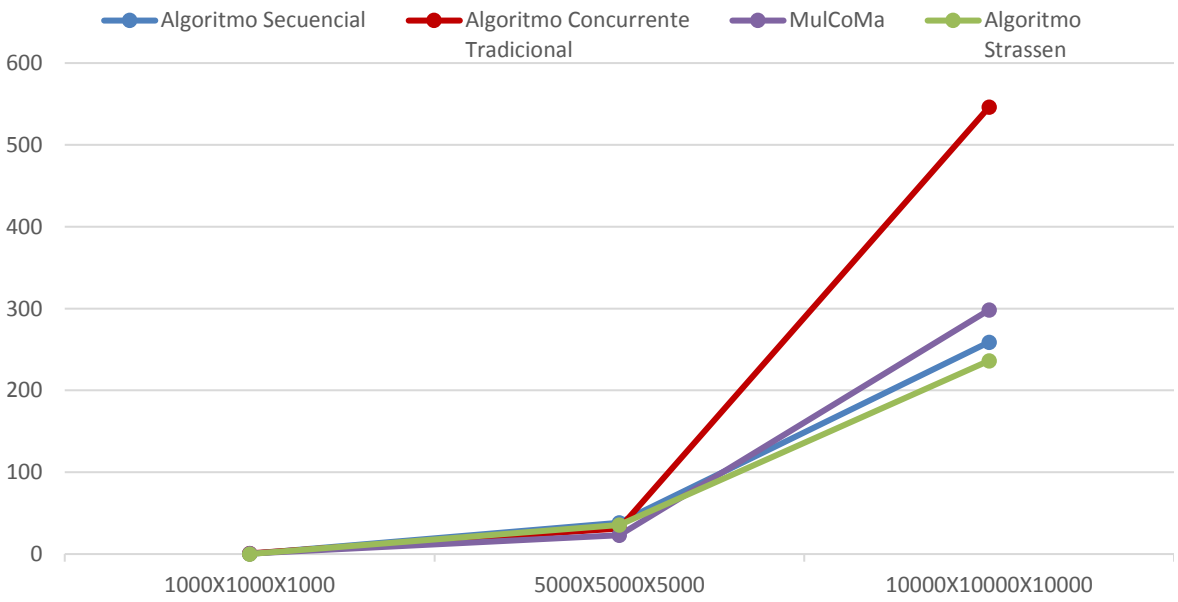

Figura 2. Gráfica del desempeño de los algoritmos en minutos

A continuación, se presentan dos gráficas que muestran el desempeño de los algoritmos. Para ambas graficas el eje X representa las combinaciones (i.e., filas de A, columnas de A y columnas de B) y el eje Y representa el tiempo (segundo para la primera gráfica, minutos para la segunda gráfica). La primer grafica (cf., Figura 1) está conformada por aquellas combinaciones de las tres tablas presentadas anteriormente, cuyo tiempo de ejecución de los algoritmos es menor a 1 minuto.

La segunda gráfica (cf., figura 2) representa las combinaciones de las tres tablas presentadas anteriormente, que generan tiempos de ejecución de los algoritmos de varios minutos.

Este criterio de representar en dos graficas las combinaciones de las tres tablas se tomó, debido a que es poco visible el comportamiento de los algoritmos, cuando los tiempos de ejecución son de segundos comparado con aquellas combinaciones que generan tiempos de ejecución de varios minutos.

Algoritmo	**Memoria mínima requerida (es decir, con dimensiones 10X10X10)**	**Memoria máxima requerida (es decir, con dimensiones 10000X10000X10000)**
Algoritmo Strassen	3.1 KB	3.1 GB
Algoritmo tradicional	1.2 KB	1.2 GB
Algoritmo concurrente tradicional	1.28 KB	1.20008 GB
MulCoMa	1.24 KB	1.20004 GB

Tabla 4. Memoria requerida por los algoritmos

En cuanto a memoria requerida por los algoritmos considerando que los datos que procesaron eran de tipo entero, se obtuvieron los siguientes resultados:

En la tabla 4 se observa que los algoritmos: tradicional, concurrente tradicional y MulCoMa, consumen aproximadamente la misma cantidad de memoria. La diferencia en estos tres algoritmos es que en los casos de concurrente tradicional y MulCoMa, se considera la creación de los hilos, lo cual origina un ligero aumento del consumo de memoria. Por su parte, el algoritmo Strassen presenta un aumento considerable en la memoria requerida comparado con los otros tres algoritmos. Se observa también qu MulCoMa, consume menos memoria que el tradicional concurrente.

7. Conclusiones

Este artículo presentó la evaluación del desempeño de algoritmos concurrentes y secuenciales para el caso de estudio de la multiplicación de matrices. De acuerdo con los resultados obtenidos, se concluye que los algoritmos concurrentes y en particular MulCoMa, tienen mejor desempeño que los algoritmos secuenciales en la mayoría de los casos. Lo que permitió que MulCoMa fuera más eficiente que los otros algoritmos fue el empleo de hilos, ya que un hilo consume menos recursos que por ejemplo un proceso. Cabe mencionar que el algoritmo tradicional concurrente también implementa hilos para multiplicar las matrices. Sin embargo, MulCoMa crea solo la mitad de hilos (cada hilo genera dos filas de la matriz resultante) para llevar a cabo el mismo proceso. Evidentemente, el hecho de crear solo la mitad de hilos implica un ahorro de recursos tanto de tiempo como de memoria.

Cabe destacar que para el equipo de prueba, se observa que a medida que el tamaño máximo (i.e., 1000, 5000, 10000) va en aumento, tanto en las filas como en las columnas de las matrices A y B; los algoritmos concurrentes van perdiendo eficiencia respecto a los algoritmos tradicionales. Por su parte el algoritmo *Strassen* mostro ser más eficiente que el algoritmo tradicional, cuando las filas y columnas de A y B tienen el valor máximo.

8. Agradecimientos

Los autores agradecen a la Universidad Juárez Autónoma de Tabasco por el apoyo otorgado para poder llevar a cabo el presente trabajo de investigación.

9. Referencias

[1] Aho, A. V. Hopcroft, J. E. & Ullman, J. (1983). Data Structures and Algorithms (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Albrecht, M. Bard, G. & Hart, W. (2010). Algorithm 898: Efficient multiplication of dense matrices over GF(2). ACM Trans. Math. Softw. 37, 1, Article 9 (January 2010), 14 pages. DOI=http://dx.doi.org/10.1145/1644001.1644010

[3] Gepner, P. & Kowalik, M. F., "Multicore processors: New way to achieve high system performance", In Proceeding of International Symposium on Parallel Computing in Electrical Engineering 2006 (PAR ELEC 2006), 2006, 9–13.

[4] Grossman, S. I. (1996), "Álgebra lineal", México: McGRAW-HILL/INTERAMERICANA DE MEXICO, Edo. de México , S.A. de C.V.

[5] Ismail, M. A. Mirza, S. H. Altaf, Talat. (2011). Concurrent Matrix Multiplication on Multi-core Processors. Published in International Journal of Computer Science and Security (IJCSS). vol. 5. 208 - 220

[6] Marquez, F. (2005). Unix - Programacion Avanzada. Alfaomega Grupo Editor.

[7] McCool, M., "Scalable programming models for massively parallel multicores", Proceedings of the IEEE, 96(5), 2008, 816–831.

[8] Pachuca, Y., "Multiplicación Rápida de Matrices", Master's thesis, Escuela Superior de Física y Matemáticas, México, D.F., 2008.

[9] STRASSEN, V. 1969. Gaussian elimination is not optimal. Nummerische Mathematik 13, 354-356.

[10] Tanenbaum, A. S., "Sistemas operativos modernos", Pearson Educación, tercera edición, México, 2009.

[11] Wolfmann, G. & Tinnetti, F., "Cómputo intensivo en clusters de nodos multicore: Análisis de speed up y eficiencia", In 37 Jornadas Argentinas de Informática e Investigación Operativa, 2008.