Implementación de Algoritmos de Multiplicación de Matrices Densas Cuadradas utilizando MPI

Germán Hüttemann
Universidad Nacional de Asunción, Facultad Politécnica,
Asunción, Paraguay
ghuttemann@gmail.com

y

Marcelo D. Rodas
Universidad Nacional de Asunción, Facultad Politécnica,
Asunción, Paraguay
rodas.marcelo@gmail.com

Resumen

Los algoritmos de multiplicación de matrices densas son una de las soluciones más utilizadas para mostrar el funcionamiento de algoritmos paralelos, y por ende, se convierten en una herramienta para la solución de problemas difíciles en un tiempo razonable. Lo esencial en las soluciones de este problema, principalmente, se enfocan en el patrón para distribuir los datos de una matriz, de tal forma que el cálculo de las operaciones básicas pueda ejecutarse lo más independientemente posible. Particularmente, este documento presenta 4 soluciones al problema mencionado, el algoritmo DNS presentado en [1], el algoritmo 2D con particionamiento por bloque cíclico presentado en [1], el algoritmo 2D diagonal presentado en [2] y el algoritmo de Cannon presentado en [1]. Todos los algoritmos fueron implementados utilizando MPI (Message Passing Interface). Se presentarán las características de cada algoritmo, conjuntamente con los resultados obtenidos de las pruebas realizadas, donde se podrá observar las aceleraciones óptimas obtenidas (incluso superescalar) en caso de mapeos uno-a-uno entre procesos y procesadores para los algoritmos DNS, Cannon y 2D diagonal, teniendo una gran disminución conforme se asignen más procesos a cada procesador. Con estos resultados, se podrá ver que la paralelización del problema presentado, solo será efectiva si se procesa en un ambiente con soporte para la paralelización, o sea, que haya tantos procesadores individuales para el nivel de paralelización esperado.

Palabras clave: Algoritmos de multiplicación de matrices, Algoritmo DNS, Algoritmo 2D con particionamiento por bloque cíclico, Algoritmo 2D Diagonal, Algoritmo de Cannon, Matrices densas, MPI.

1 Introducción

Este trabajo plantea la solución del problema de la multiplicación de matrices densas cuadradas. Se plantean 4 soluciones, implementadas utilizando el modelo de paso de mensajes, el algoritmo DNS presentado en [1], el algoritmo 2D con particionamiento por bloque cíclico presentado en [1], el algoritmo 2-D diagonal presentado en [2] y el algoritmo de Cannon presentado en [3]. El planteamiento de la solución, para cada algoritmo, se basa fundamentalmente en sus referencias bibliográficas, con simples adaptaciones necesarias para que puedan ejecutarse con MPI.
El trabajo esta organizado de la siguiente manera: en la sección 2 se describe el problema a solucionar. En la sección 3, se presenta el modelo matemático para la multiplicación de matrices. En la sección 4, se presenta la descripción del algoritmo secuencial. Luego, en la sección 5, se presenta la descripción de los algoritmos paralelos. En la sección 6 se presentan los resultados obtenidos en las pruebas; en la sección 7, las conclusiones obtenidas y, finalmente, en la sección 8, se proponen algunos trabajos futuros para mejorar el análisis del problema abordado.

2 Descripción del Problema

El problema consiste en realizar la multiplicación de matrices densas cuadradas. Para ello se busca aprovechar las características de las operaciones básicas de multiplicación de matrices, que permiten la paralelización del proceso completo. La paralelización debe ser realizada utilizando el mecanismo de comunicación por paso de mensajes, comúnmente conocido como Message Passing Interface (MPI).
En el modelo de MPI, varios procesos (unidades lógicas de procesamiento), interactúan entre sí comunicándose a través del paso de mensajes. Estos mensajes pueden ser de tamaño reducido, como para la sincronización entre los procesos, o de gran tamaño, para transmitir bloques enteros de datos que sirven de entrada para, o son el resultado de, un determinado procesamiento. En el caso de la multiplicación de matrices, la mayor parte de las comunicaciones se realiza para transmitir distintos bloques de las matrices de entrada y salida. La comunicación realizada puede involucrar a dos o más procesos, existiendo así comunicaciones punto-a-punto y comunicaciones colectivas, tal como se describe en el capítulo 4 de [1].

3 Modelo Matemático

La multiplicación de matrices tiene como entrada dos matrices *A* y *B* multiplicables, de dimensiones *MxN* y *NxR*, respectivamente. Como resultado, se obtiene otra matriz de tamaño MxR, normalmente llamada matriz *C*. Los elementos de las matrices pueden tomar valores de conjuntos de números enteros, reales, complejos, etc. Cuando una matriz *A* es de dimensión *NxN*, la misma recibe el nombre de matriz cuadrada de dimensión *N*. La multiplicación de matrices cuadradas es un caso particular de la multiplicación general de matrices, en la que *A* y *B* son matrices cuadradas de dimensión *N*. El resultado obtenido, matriz *C*, tiene las mismas características. En la Figura 1 puede apreciarse un ejemplo de multiplicación de matrices cuadradas.

$A_{0,0}$	$A_{0,1}$	…	$A_{0,N-1}$		$B_{0,0}$	$B_{0,1}$	…	$B_{0,N-1}$		$C_{0,0}$	$C_{0,1}$	…	$C_{0,N-1}$
$A_{1,0}$	$A_{1,1}$	…	$A_{1,N-1}$		$B_{1,0}$	$B_{1,1}$	…	$B_{1,N-1}$		$C_{1,0}$	$C_{1,1}$	…	$C_{1,N-1}$
…	…	…	…	x	…	…	…	…	=	…	…	…	…
…	…	…	…		…	…	…	…		…	…	…	…
$A_{N-1,0}$	$A_{N-1,1}$	…	$A_{N-1,N-1}$		$B_{N-1,0}$	$B_{N-1,1}$	…	$B_{N-1,N-1}$		$C_{N-1,0}$	$C_{N-1,1}$	…	$C_{N-1,N-1}$

Figura 1. Ejemplo Gráfico de la multiplicación de matrices cuadradas A x B = C

4 Descripción del Algoritmo Secuencial

El algoritmo secuencial más sencillo para multiplicar dos matrices A y B, cuadradas de dimensión *N*, está presentado en la Figura 2. Para cada valor C_{ij} de la matriz resultado, se multiplican escalarmente la *fila i* de la matriz A y la *columna j* de la matriz B. Considerando a la combinación de sumas y multiplicaciones como operaciones elementales, el costo asintótico de este algoritmo es N^3, ya que se necesitan calcular N^2 valores de la matriz C, donde cada uno de los cálculos consiste en *N* operaciones elementales.

```
multiplicacionMatrices(A, B, C, N)
inicio
  desde i = 0 hasta N-1
    desde j = 0 hasta N-1
      C[i,j] = 0
      desde k = 0 hasta N-1
        C[i,j] = C[i,j] + A[i,k] * B[k,j]
      fin-desde
    fin-desde
  fin-desde
fin
```

Figura 2. Algoritmo secuencial de multiplicación de matrices cuadradas de dimensión N.

5 Descripción de los Algoritmos Paralelos

Para paralelizar la tarea de computar la multiplicación de matrices, se utilizan distintos tipos de particionamiento de datos. Dependiendo del caso, el particionamiento puede realizarse sobre los datos de entrada (matrices A y B), datos de salida (matriz C) o datos intermedios (cálculos parciales de la matriz C). El capítulo 3 de [1] describe en detalle los diferentes esquemas de particionamiento de las matrices para el cómputo paralelo. Los algoritmos paralelos presentados en este trabajo, utilizan particionamiento de datos de salida y de datos intermedios, tal como se verá más adelante.

De forma general el tiempo de procesamiento de los algoritmos presentados se puede clasificar en dos: Tiempo de Operaciones MPI (TOM) y Tiempos de Computo Local (TCL). El TOM se refiere a las operaciones de comunicaciones utilizando el paso de mensajes, incluyendo su tiempo de inicialización local (t_s) y tiempo de comunicación por palabra (t_w). Los costos de operaciones de comunicación utilizados son los presentados en el capítulo 4 de [1]. El TCL se refiere específicamente a tiempos de operaciones de construcción de mensajes y tiempos de multiplicación de matrices.

Un factor relevante en el cálculo de tiempos, es el tipo de dato utilizado para los elementos de la matriz, ya que el tamaño del elemento es factor en todas las operaciones medidas. En este trabajo se utilizó números punto flotante, correspondiente al tipo de dato MPI_FLOAT, que en una máquina de 32 bits, corresponde a una palabra.

5.1 Algoritmo 2D con particionamiento por bloque cíclico

El algoritmo 2D con particionamiento por bloque, en su caso más simple, particiona la matriz de salida en bloques de dos dimensiones del mismo tamaño. Cada uno de los bloques se convierte en una tarea, que puede ser computada independientemente por un proceso determinado. Se busca que la cantidad T de tareas sea igual a la cantidad P de procesos. En la Figura 3 se puede ver el mapeamiento de las tareas entre los procesos.

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

Figura 3. Mapeamiento de tareas a procesos con particionamiento por bloque ($T = P = 16$)

P_0	P_1	P_2	P_0
P_1	P_2	P_0	P_1
P_2	P_0	P_1	P_2
P_0	P_1	P_2	P_0

Figura 4. Mapeamiento de tareas a procesos con particionamiento por bloque cíclico ($T = 16$ y $P = 3$)

El algoritmo 2D con particionamiento por bloque cíclico es idéntico al algoritmo anterior, en cuanto al particionamiento, diferenciándose en el tamaño y cantidad de tareas. En el particionamiento por bloque cíclico se particiona la matriz de salida en bloques más pequeños, tal que $T >> P$. Luego, cada tarea es mapeada entre los procesos de una manera cíclica. En la Figura 4 se puede ver el mapeamiento de las tareas entre los procesos, para el caso de T=16 y P=3. Con esta variación, se consigue aliviar ciertos problemas de desbalanceo de carga y ociosidad, tal como se menciona en el capítulo 3 de [1]. Además, cada proceso necesita menor cantidad de memoria y la cantidad de datos transmitidos en cada comunicación es menor.

Cabe destacar, que en el presente trabajo se trata el problema de multiplicación de matrices densas, por lo que las tareas son equivalentes en cuanto al cómputo necesario para resolverlas. Por tanto, no existirían problemas de desbalanceo de carga entre los procesos, más que cuando T no sea múltiplo de P, tal como en la Figura 4. En el peor

de los casos, los primeros $\frac{P}{2}$ procesos recibirán una tarea más que el resto de los procesos. En el mejor de los casos, los primeros *P-1* procesos recibirán una tarea más que el último proceso, o el primer proceso recibirá una tarea más que el resto. En general, los primeros *T % P* procesos recibirán una tarea más que el resto de los procesos.
En la implementación de este trabajo, se utilizó un esquema maestro-esclavo. En este esquema, existe un proceso maestro que construye las tareas, las distribuye cíclicamente entre los procesos esclavos y espera las respuestas de cada proceso esclavo en el mismo orden en el que fueron enviadas las tareas. Finalmente, el proceso maestro almacena los resultados recibidos. El ciclo se repite mientras existan tareas disponibles. En las Figuras 5(a) y 5(b) se puede ver el algoritmo tal como fue implementado. La dimensión de cada bloque 2D, *blockSize*, que representa una tarea, es calculado como $\sqrt{N}$, donde *N* es la dimensión de las matrices. Así, la cantidad de tareas es igual a *N*, lo que se obtiene dividiendo el tamaño total de cada matriz, N^2, entre el tamaño de cada bloque que representa una tarea, *N*. Cabe destacar que con esta elección, la dimensión *N* de las matrices debe ser cuadrado perfecto.

```
procesoMaestro(A, B, C, N, P)
inicio
  ...

  blkSize = sqrt(N)

  /* Identificar posición de cada tarea en la matriz */
  desde i=0 hasta N-1 incremento blkSize
    desde j=0 hasta N-1 incremento blkSize
      T[k] = Guardar posición inicial i,j de tarea k
    fin-desde
  fin-desde

  /* Enviar primer grupo de tareas a procesos desde 1, 2, ..., P-1 */
  tareas_disponibles = N
  tareas_por_recibir = 0
  desde procRank = 1 hasta P-1
    M = Construir mensaje con blkSize filas de A y blkSize columnas de B
    MPI_Send(M, blkSize * N * 2, MPI_FLOAT, procRank, tag, MPI_COMM_WORLD)
    tareas_por_recibir = tareas_por_recibir + 1
    tareas_disponibles = tareas_disponibles - 1
  fin-desde

  /* Recibir resultados calculados, enviar nueva tareas, guardar resultado */
  procRank = 1
  mientras (tareas_por_recibir > 0)
    MPI_Recv(R, N, MPI_FLOAT, procRank, tag, MPI_COMM_WORLD, &status)
    tareas_por_recibir = tareas_por_recibir - 1
    si (tareas_disponibles > 0)
      M = Construir mensaje con blkSize filas de A y blkSize columnas de B
      MPI_Send(M, blkSize * N * 2, MPI_FLOAT, procRank, tag, MPI_COMM_WORLD)
      tareas_por_recibir = tareas_por_recibir + 1
      tareas_disponibles = tareas_disponibles - 1
    fin-si
    Guardar resultado R en correspondiente posición de C
    Incrementar circularmente procRank entre 1 y P-1
  fin-mientras
fin
```

Figura 5(a). Algoritmo 2D con particionamiento por bloque cíclico (proceso maestro)

```
procesoEsclavo(N, P, rank)
inicio
  tareas_a_recibir = N / (P-1)

  /*
   * Los primeros N % (P-1) procesos deben
   * recibir una tarea más.
   */
  si (rank <= N % (P-1))
    tareas_a_recibir = tareas_a_recibir + 1
  fin-si

  desde k = 0 hasta tareas_a_recibir
    MPI_Recv(M, blkSize * N * 2, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status)
    R = Multiplicar filas de A y columnas de B recibidas en M
    MPI_Send(R, N, MPI_FLOAT, 0, tag, MPI_COMM_WORLD)
  fin-desde
fin
```

Figura 5(b). Algoritmo 2D con particionamiento por bloque cíclico (proceso esclavo)

5.1.1. ***Análisis de costo***

Cada una de las *N* tareas está representada por un bloque de dimensión $\sqrt{N}$ en la matriz de salida. Para calcular la multiplicación correspondiente a dicho bloque, cada proceso necesita $\sqrt{N}$ filas de la matriz A y $\sqrt{N}$ columnas de la matriz B. Así, el proceso maestro necesita enviar *N* mensajes de tamaño $2N\sqrt{N}$. En el caso de los procesos esclavos, cada uno necesita enviar los resultados calculados, que tienen un tamaño $\sqrt{N} * \sqrt{N} = N$. Debido a que todos los procesos envían sus resultados aproximadamente en paralelo, en total, entre todos los procesos esclavos, deben enviar aproximadamente $\frac{N}{P-1}$ resultados. El costo total de toda la comunicación está dado por la siguiente expresión:

$$\left(N + \frac{N}{P-1}\right) * t_s + \left(2N^2\sqrt{N} + \frac{N^2}{P-1}\right) * t_w$$

En cuanto al cómputo realizado por los procesos, el maestro debe identificar la posición *(i, j)* de cada tarea en la matriz de salida, lo cual tiene un costo de *N*. Además, antes del envío de cada mensaje, el maestro debe inicializar el mensaje, lo que consiste en copiar los datos a un buffer contiguo de memoria y es igual a $2N^2\sqrt{N}$. Finalmente, tras recibir cada resultado, debe copiarlo en la posición correspondiente de la matriz de salida, lo que tiene un costo de *N*. Así, el costo total de cómputo realizado por el proceso maestro es de $2N^2\sqrt{N} + 2N$. Por parte de los procesos esclavos, en total deben computar aproximadamente $\frac{N}{P-1}$ multiplicaciones de bloques (filas de A y columnas de B) de tamaño $N\sqrt{N}$. Así, el cómputo total de los procesos esclavos está dado por $\frac{N^3}{P-1}$. El costo total de todo el cómputo está dado por la siguiente expresión:

$$\frac{N^3}{P-1} + 2N^2\sqrt{N} + 2N$$

Cabe destacar, que entre el inicio del cómputo de cada proceso esclavo y el cómputo del proceso maestro para copiar el resultado recibido en la matriz de salida, existe cierto paralelismo, por lo que la expresión anterior es una sobrestimación del costo real. Sin embargo, el cálculo del costo exacto es bastante complejo y sobrepasa los límites de este trabajo. El costo total del algoritmo está dado por la siguiente expresión:

$$\frac{N^3}{P-1} + 2N^{2.5} + 2N + \left(N + \frac{N}{P-1}\right) * t_s + \left(2N^2\sqrt{N} + \frac{N^2}{P-1}\right) * t_w$$

5.2 Algoritmo de Cannon

El algoritmo de Cannon implementado está basado en el presentado en [1]. En este algoritmo se realiza un particionamiento de datos de salida. Se supone una malla 2D de procesos de tamaño $\sqrt{P} * \sqrt{P}$, donde *P* es la cantidad de procesos. Los procesos en la malla son capaces de comunicarse cíclicamente por filas y por columnas. Este tipo de malla 2D recibe el nombre de *malla toroidal*. Cada proceso de la malla tiene asignado el cálculo de un bloque de dimensión $\frac{N}{\sqrt{P}}$ de la matriz C, e inicialmente posee el bloque correspondiente de las matrices A y B. Esto impone una restricción al tamaño de las matrices respecto al tamaño de la malla de procesos.

El proceso que se sigue para realizar la multiplicación de las matrices consta de tres fases y consiste básicamente en que cada proceso calcule la multiplicación del bloque que tiene asignado y luego realice un corrimiento del bloque de A hacia la izquierda y del bloque de B hacia arriba. En la primera fase del algoritmo, se realiza un alineamiento inicial de los bloques de A y B, de manera tal a que el bloque de A en cada proceso coincida con el correspondiente bloque de B para dar inicio a la multiplicación. El alineamiento inicial realizado puede verse en las Figuras 6(a) y 6(b). El desplazamiento en el corrimiento que cada proceso realiza de su bloque de A (B) está dado por la fila (columna) en la que se encuentra dicho proceso dentro de la malla 2D.

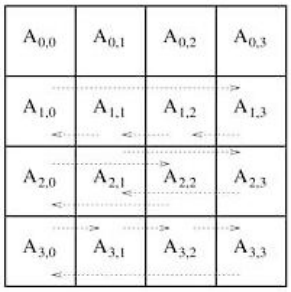

Figura 6(a). Alineamiento inicial de bloques de A

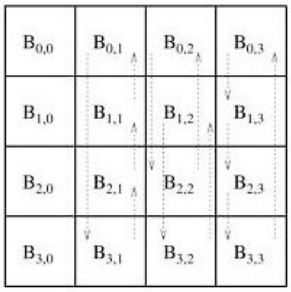

Figura 6(b). Alineamiento inicial de bloques de B

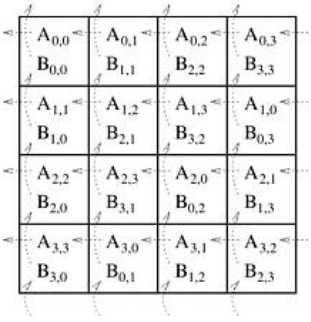

Figura 7(a). Bloques de A y B tras el alineamiento inicial

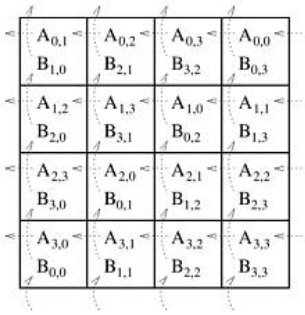

Figura 7(b). Bloques de A y B tras el primer corrimiento

En la segunda fase, se procede a realizar $\sqrt{P}$ pasos multiplicación-corrimiento. Cada uno de estos pasos consiste en multiplicar los bloques de A y B, y luego realizar un corrimiento de los mismos, hacia la izquierda para A y hacia arriba para B. En la Figura 7(a) se puede ver cómo quedan los bloques de A y B tras el alineamiento inicial y en la Figura 7(b), cómo quedan los mismos tras el primer corrimiento. En la tercera y última fase, se restaura la distribución inicial de los bloques de A y B. Si bien esta última fase no es estrictamente necesaria, para probar la correctitud del algoritmo fue necesario implementarlo, ya que cada proceso podía imprimir sus correspondientes bloques. El algoritmo de implementado puede verse en la Figura 8. Cabe destacar que en el contexto del algoritmo nos referimos a A, B y C en relación a los bloques de las respectivas matrices, pertenecientes a cada proceso.

```
algoritmoCannon(N, A, B, C, P, comm_2d)
inicio
  ...

  nlocal = N / sqrt(P)

  /* FASE 1: Realizamos el alineamiento inicial de la matriz A */
  MPI_Cart_shift(comm_2d, 1, -myCoords[0], &origen, &destino)
  MPI_Sendrecv_replace(A, nlocal*nlocal, MPI_FLOAT, destino, tag, origen, tag, comm_2d, &status)

  /* FASE 1: Realizamos el alineamiento inicial de la matriz B */
  MPI_Cart_shift(comm_2d, 0, -myCoords[1], &origen, &destino)
  MPI_Sendrecv_replace(B, nlocal*nlocal, MPI_FLOAT, destino, tag, origen, tag, comm_2d, &status)

  /* FASE 2: Calculamos los ranks de los corrimientos hacia la izquierda y hacia arriba */
  MPI_Cart_shift(comm_2d, 1, -1, &derecha, &izquierda)
  MPI_Cart_shift(comm_2d, 0, -1, &abajo, &arriba)

  /* FASE 2: Ciclo multiplicación-corrimiento */
  desde k = 0 hasta sqrt(P)-1
    multiplicar(A, B, C, nlocal)
    MPI_Sendrecv_replace(A, nlocal*nlocal, MPI_FLOAT, izquierda, tag, derecha, ...)
    MPI_Sendrecv_replace(B, nlocal*nlocal, MPI_ELEMENT_T, arriba, tag, abajo, ...)
  fin-desde

  /* FASE 3: Restauramos la distribución original de las matrices A y B */
  MPI_Cart_shift(comm_2d, 1, +myCoords[0], &origen, &destino)
  MPI_Sendrecv_replace(A, nlocal*nlocal, MPI_FLOAT, destino, tag, origen, ...)
  MPI_Cart_shift(comm_2d, 0, +myCoords[1], &origen, &destino)
  MPI_Sendrecv_replace(B, nlocal*nlocal, MPI_FLOAT, destino, tag, origen, ...);
fin
```

Figura 8. Algoritmo de Cannon

5.2.1. ***Análisis de costo***

Cada proceso realiza el alineamiento inicial de los bloque de cada matrices y luego restaura la distribución inicial de los mismos, enviando mensajes de tamaño $\frac{N^2}{P}$. Durante el proceso de multiplicación-corrimiento se realizan $\sqrt{P}$ corrimientos para los bloques de cada matriz, cada uno del mismo tamaño anterior. El costo total de la comunicación está dado por la siguiente expresión:

$$4 * \left(t_s + t_w * \frac{N^2}{P} \right) + 2\sqrt{P} * \left(t_s + t_w * \frac{N^2}{P} \right) = \left(4 + 2\sqrt{P} \right) * \left(t_s + \frac{N^2}{P} t_w \right)$$

El cómputo realizado por cada proceso consiste en $\sqrt{P}$ multiplicaciones de bloques de las matrices, cada uno de tamaño $\frac{N}{\sqrt{P}}$, por lo cual el costo de cada multiplicación es de $\frac{N^3}{P\sqrt{P}}$, y el del cómputo total es de $\frac{N^3}{P}$. La siguiente expresión da el costo total del algoritmo:

$$\frac{N^3}{P} + \left(4 + 2\sqrt{P} \right) * \left(t_s + \frac{N^2}{P} t_w \right)$$

5.3 Algoritmo 2D diagonal

El algoritmo 2D diagonal implementado está basado en el presentado en [2]. En este algoritmo se realiza un particionamiento de datos de salida y datos intermedios. Se considera una malla 2D de procesos de tamaño *q x q*, donde $q = \sqrt{P}$, $blockSize = \frac{N}{q}$ y P es la cantidad de procesos utilizados para las multiplicaciones. Según se muestra en la Figura 9, inicialmente las matrices de entrada, A y B, se segmentan en *q* conjuntos de columnas y en *q* conjuntos de filas, respectivamente, formándose submatrices de tamaño *N*blockSize* para A y submatrices de tamaño *blockSize*N* para B.

Matriz A

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$

Matriz B

$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{1,4}$
$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{2,4}$
$B_{3,1}$	$B_{3,2}$	$B_{3,3}$	$B_{3,4}$
$B_{4,1}$	$B_{4,2}$	$B_{4,3}$	$B_{4,4}$

Malla de Procesos

$A_{*,1}$	$A_{*,2}$		
$B_{1,*}$	$B_{2,*}$		
		$A_{*,3}$	$A_{*,3}$
		$B_{3,*}$	$B_{4,*}$

Figura 9. Ejemplo del proceso de distribución inicial para el Algoritmo 2D diagonal con N = 4 y P = 4.

Inicialmente, en la malla de procesos (Figura 9), cada procesador $P_{j,j}$ de la diagonal, contiene el j^{esimo} grupo de columnas de A (matriz A de la Figura 9) y el j^{esimo} grupo de filas de B (matriz B de la Figura 9). El conjunto de procesadores $P_{*,j}$ posee la tarea de realizar el producto externo de las columnas de A y partes de las filas de B, inicialmente almacenadas en $P_{j,j}$. Esto se consigue realizando un *one-to-all peronalized broadcast* (MPI_Scatter) del grupo de filas de B y *one-to-all broadcast* (MPI_Bcast) del grupo de columnas de A, a lo largo de la dirección-y. Una vez realizado el producto externo, el último paso consiste en realizar una operación *all-to-one reduction* (MPI_Reduce) para reducir los resultados adicionando a lo largo de la dirección-x y el resultado de la matriz C es obtenida a lo largo de los procesadores diagonales, alineados de la misma manera que la distribución Inicial. El algoritmo implementado se puede ver en la Figura 10. Cabe destacar que en el contexto del algoritmo nos referimos a A, B, C en relación a los bloques de las respectivas matrices, pertenecientes a cada proceso.

```
Distribución Inicial: Cada procesador diagonal p_{j,j} tiene el j^esimo conjunto
de columnas y filas de las matrices A y B respectivamente.

algoritmo2D-Diagonal(N, A, B, C, P, comm2_d)
inicio
  ...

  nlocal = N / sqrt(P)

  /*
   * PASO 1.1:
   * Broadcast de A_{*,j} desde p_{j,j} a todos los procesos p_{*,j}
   */
  MPI_Bcast(A, N*nlocal, MPI_FLOAT, P_jj, comm_col)

  /*
   * PASO 1.2:
   * Scatter de B_{i,*} desde p_{j,j} a todos los procesos p_{*,j}. En este proceso, B_{i,*}
   * se divide proporcionalmente a la cantidad de procesos que se encargan
   * de procesar el conjunto j de columnas. Antes del envío del mensaje
   * deben copiarse los datos a memoria contigua en el buffer Baux.
   */
  MPI_Scatter(Baux, nlocal*nlocal, MPI_FLOAT, B, nlocal*nlocal,
                MPI_FLOAT, P_jj, comm_col)

  /*
   * PASO 2:
   * Multiplicación: Calcular el producto M_{*,j} = A_{*,j} x B_{i,p}
   */
  Multiplicar(A, B, M, N, nlocal)

  /*
   * PASO 3:
   * Reducción en C_{i,i} (ubicado en P_{i,i}), de todos los M_{*,j} (ubicados en P_{i,*})
   */
  MPI_Reduce(M, C, nlocal*N, MPI_FLOAT, MPI_SUM, P_ii, comm_row);
Fin
```

Figura 10. Algoritmo 2D-Diagonal

*5.3.1. **Análisis de costo***

En el primer paso del algoritmo los procesos de la diagonal realizan una operación de broadcast de las columnas de la matriz A, con mensajes tamaño igual a $\frac{N^2}{\sqrt{P}}$. Luego, realizan un scatter de las filas de la matriz B, con mensajes de tamaño igual a $\frac{N^2}{P}$. En último paso, se realiza una reducción con mensajes de tamaño $\frac{N^2}{\sqrt{P}}$. Así, el costo total de las comunicaciones, considerando que las operaciones de reducción y broadcast tienen el mismo costo están dados por las siguientes expresiones:

$$2\log\sqrt{P}\left(t_s + \frac{N^2}{\sqrt{P}}t_w\right) \qquad \text{(broadcast y reducción)}$$

$$\left(t_s \log\sqrt{P} + \frac{N^2}{P}\left(\sqrt{P}-1\right)t_w\right) \qquad \text{(scatter)}$$

Para el cómputo de las multiplicaciones, se tiene bloques de A de tamaño $N \times \frac{N}{\sqrt{P}}$ y bloques de la matriz B de tamaño $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$. El costo de la multiplicación es de $N \times \frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}} = \frac{N^3}{P}$. Además, para el scatter realizado, debe copiarse el mensaje a ser enviado en memoria contigua, lo que tiene un costo de $N \times \frac{N}{\sqrt{P}} = \frac{N^2}{\sqrt{P}}$. El costo total del algoritmo está dado por la siguiente expresión:

$$\frac{N^3}{P} + \frac{N^2}{\sqrt{P}} + 2\log\sqrt{P}\left(t_s + \frac{N^2}{\sqrt{P}}t_w\right) + \left(t_s \log\sqrt{P} + \frac{N^2}{P}\left(\sqrt{P}-1\right)t_w\right)$$

5.4 Algoritmo DNS

El algoritmo DNS (Dekel, Nassimi, Sahni) implementado está basado en el presentado en [1]. En este algoritmo se realiza un particionamiento de datos de salida y de datos intermedios. Se supone una malla 3D de procesos de tamaño $\sqrt[3]{P} * \sqrt[3]{P} * \sqrt[3]{P}$, donde P es la cantidad de procesos. Cada proceso de la malla tiene asignado el cálculo de una parte de un bloque de dimensión $\frac{N}{\sqrt[3]{P^2}}$ de la matriz C. Específicamente, el proceso P_{ijk} de la malla tiene asignado el cálculo de la multiplicación entre los bloques A_{ik} y B_{ki}, perteneciente al bloque C_{ij}. El hecho de tener una malla 3D de procesos, impone una restricción al tamaño de las matrices respecto al tamaño de la malla de procesos, tal que $\frac{N}{\sqrt[3]{P}}$ sea un número entero. El proceso para llevar a cabo la multiplicación de las matrices consta de tres fases principales. Inicialmente, las matrices A y B están distribuidas por bloques entre los $\sqrt[3]{P^2}$ procesos del plano $k=0$, de manera tal que el proceso P_{ij0}, posea los bloques A_{ij} y B_{ij}. Esta distribución inicial se puede ver en la Figura 11(a). En la primera fase del algoritmo, cada proceso P_{ij0} debe transmitir A_{ij} al proceso P_{ijj} (j > 0) y B_{ij} al proceso P_{iji} (i > 0). Esto es llevado a cabo a través de una comunicación punto-a-punto entre cada par de proceso, y está representado por las flechas hacia arriba en la Figura 11(a).

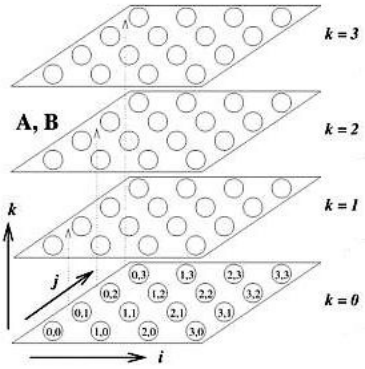

Figura 11(a). Distribución inicial de A y B.
Las tres flechas verticales indican cada comunicación punto-a-punto entre P_{ij0} y P_{ijj} (j>0)

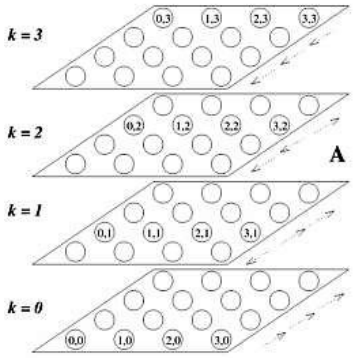

Figura 11(b). Luego de la comunicación punto-a-punto de los bloques A_{ij}. Las flechas en cada plano indican el broadcast de éstos mismo bloques.

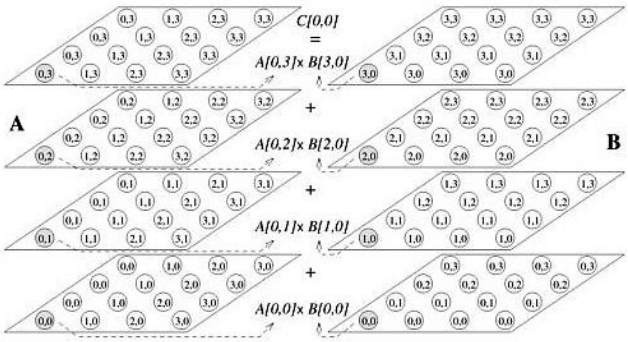

Figura 12. Multiplicación y Reducción para el bloque C_{00}

En la segunda fase del algoritmo, una vez que cada bloque A_{ij} y B_{ij} fue transmitido al proceso correspondiente, se realiza dos comunicaciones *one-to-all broadcast*, en cada plano. El primer broadcast es realizado por el proceso P_{ijj} para transmitir el bloque A_{ij} a lo largo del *eje j* a los procesos P_{i*j}. El segundo broadcast es realizado por el proceso P_{iji} para transmitir el bloque B_{ij} a lo largo del *eje i* a los procesos P_{*ji}. Al final de esta fase, cada proceso P_{ijk} posee los bloques A_{ik} y B_{kj} y realiza la multiplicación de los mismos, computando parte del bloque C_{ij}. Una vez calculada la multiplicación de por cada proceso P_{ijk}, en la tercera fase, se procede a sumar todas estas partes, cada una correspondiente al bloque C_{ij} de la matriz C. Esta última fase se realiza a través de una comunicación *all-to-one reduction* a lo largo del *eje k*, en la que cada proceso P_{ijk} (k > 0) envía su parte al proceso P_{ij0}. Este último paso junto con el de multiplicación pueden apreciarse en la Figura 12, para el bloque C_{00}. El algoritmo implementado puede verse en la Figura 13. Cabe destacar que en el contexto del algoritmo nos referimos a A, B, C y R (buffer para resultado de la reducción) en relación a los bloques de las respectivas matrices, pertenecientes a cada proceso.

```
algoritmoDNS(N, A, B, C, R, P, comm_3d)
inicio
  ...

  nlocal = N / cbrt(P)

  /*
   * FASE 1.1:
   * Los procesos Pij0 (plano 0) deben transmitir
   * A a los procesos Pijj (j > 0).
   */
  MPI_Send(A, nlocal*nlocal, MPI_FLOAT, Pijj, tag, comm_3d)

  /*
   * FASE 1.2:
   * Los procesos Pijk (k > 0 && j=k) deben recibir A de Pij0.
   */
  MPI_Recv(A, nlocal*nlocal, MPI_FLOAT, Pij0, tag, comm_3d, &status)

  /*
   * FASE 1.3:
   * Los procesos Pij0 (plano 0) deben transmitir
   * B a los procesos Piji (i > 0).
   */
  MPI_Send(B, nlocal*nlocal, MPI_FLOAT, Piji, tag, comm_3d);

  /*
   * FASE 1.4:
   * Los procesos Pijk (k > 0 && i=k) deben recibir B de Pij0.
   */
  MPI_Recv(B, nlocal*nlocal, MPI_FLOAT, Pij0, tag, comm_3d, &status);

  /*
   * FASE 2.1:
   * Broadcast de Aij por el proceso Pijj a los procesos Pi*j
   */
  MPI_Bcast(A, nlocal*nlocal, MPI_FLOAT, Pijj, comm_cols)

  /*
   * FASE 2.2:
   * Broadcast de Bij por el proceso Piji a los procesos P*ji
   */
  MPI_Bcast(B, nlocal*nlocal, MPI_FLOAT, Piji, comm_rows)

  /*
   * FASE 2.3:
   * Multiplicación de bloques Aik y Bkj
   */
  Multiplicar(A, B, C, nlocal)

  /*
   * FASE 3:
   * Cada proceso Pijk (k > 0) debe realizar una operación
   * all-to-one reduction hacia el proceso Pij0 en la matriz R (resultado)
   */
  MPI_Reduce(C, R, nlocal*nlocal, MPI_FLOAT, MPI_SUM, Pij0, comm_reduction)
Fin
```

Figura 13. Algoritmo DNS

*5.4.1. **Análisis de costo***

En la primera fase del algoritmo cada proceso realiza dos transmisiones punto-a-punto, cada una de tamaño $\frac{N^2}{\sqrt[3]{P^2}}$. Luego, en la segunda fase, se realizan dos broadcast entre $\sqrt[3]{P}$ procesos, cada uno de de tamaño $\frac{N^2}{\sqrt[3]{P^2}}$. En la tercera fase, se realiza una operación de reducción entre $\sqrt[3]{P}$ procesos, cada uno de de tamaño $\frac{N^2}{\sqrt[3]{P^2}}$. El costo total de la comunicación está dado por la siguiente expresión:

$$2\left(t_s + t_w \frac{N^2}{P^{2/3}}\right) + 3\left(t_s + t_w \frac{N^2}{P^{2/3}}\right)\log P^{1/3} = \left(2 + 3\log P^{1/3}\right) * \left(t_s + t_w \frac{N^2}{P^{2/3}}\right)$$

En cuanto al cómputo realizado, cada proceso realiza la multiplicación de bloques de A y B de dimensión $\frac{N}{\sqrt[3]{P}}$, lo que tiene un costo de $\frac{N}{\sqrt[3]{P}} \times \frac{N}{\sqrt[3]{P}} \times \frac{N}{\sqrt[3]{P}} = \frac{N^3}{P}$. El costo total del algoritmo está dado por la siguiente expresión:

$$\frac{N^3}{P} + \left(2 + 3\log\sqrt[3]{P}\right) * \left(t_s + t_w \frac{N^2}{\sqrt[3]{P^2}}\right)$$

6 Resultados

6.1 Ambiente de Prueba

Las pruebas fueron realizadas sobre 9 computadores con las siguientes características:

- Sistema operativo: Windows XP Professional Service Pack 2
- Procesador: Intel Pentium D (dual core) 3.00 GHz
- Memoria: 1 GB
- Compilador: GCC 3.4.4 (Cygwin)
- Distribución MPI: MPICH NT 1.2.5

Se utilizó una topología de red en bus de una velocidad de 100MBps (Fast Ethernet). Los porcentajes de utilización de procesador y memoria, en el momento de las pruebas, fueron de aproximadamente 2% y 30%, respectivamente.

6.2 Métricas

Las métricas utilizadas para realizar el análisis de rendimiento de los algoritmos son las siguientes:

1. Tiempo de ejecución secuencial (T_S): medido a partir del algoritmo secuencial, ejecutado en una computadora de las mismas características mencionadas más arriba.
2. Tiempo de ejecución paralelo (T_P): medido desde el momento en que inicia el cómputo paralelo hasta el momento en que el último proceso termina su ejecución.
3. Aceleración (A): medida como la razón entre el tiempo del algoritmo secuencial ejecutado en un proceso y el tiempo de cada algoritmo paralelo.

$$A = \frac{T_S}{T_P}$$

4. Eficiencia (E): medido como la razón entre la aceleración obtenida por cada algoritmo y la cantidad de procesos utilizados para las ejecuciones de los mismos.

$$E = \frac{A}{P} = \frac{T_S}{P * T_P}$$

6.3 Análisis de resultados

Las pruebas fueron realizadas utilizando P=8 y P=27 para DNS. Para los demás algoritmos se utilizó P=9 y P=25. Esta elección se debe a las restricciones impuestas por cada algoritmo respecto al valor de P. En cuanto al valor de N, fueron utilizados tres valores, superior y lo más próximo posible a 1000, 2000 y 4000, tal que cumplan con las restricciones impuestas por los algoritmos respecto al valor de N. El valor más restrictivo es el del algoritmo 2D con particionamiento por bloque cíclico, que requiere que N sea cuadrado perfecto. Así, para este algoritmo, se utilizaron valores de N iguales a 1024, 2025 y 4096. En general, para cada algoritmo se utilizó una cantidad variable de N según la cantidad de procesos. En la Tabla 1 se muestra las equivalencias para valores de 1000, 2000 y 4000, para cada algoritmo. Cuando nos refiramos a valores de N como 1000, 2000, 4000, estaremos haciendo referencia a los valores reales presentados en la Tabla 1.

Algoritmo	**P = 8 ó 9**			**P = 25 ó 27**		
	1000	**2000**	**4000**	**1000**	**2000**	**4000**
2D-Cíclico	1024	2025	4096	1024	2025	4096
Cannon	1023	2025	4095	1025	2025	4095
2D-Diagonal	1023	2025	4095	1025	2025	4095
DNS	1024	2024	4096	1023	2025	4095

Tabla 1. Valores de N para cada algoritmo según P

Si hacemos que t_s y t_w tiendan a uno, en los costos de cada uno de los algoritmos y, además, calculamos el valor asintótico de T_P en función a N, para cada uno de los valores de P, se pueden obtener un ranking teórico entre los algoritmos, tal como se muestra en la Tabla 2. En base a este ranking se realiza el análisis de las pruebas. Cabe destacar que, a pesar del ranking obtenido, la diferencia entre los tres primeros algoritmos es muy reducida.

Ranking	**Cantidad de procesos (*P*)**	
	8 ó 9	**25 ó 27**
1º	2D-Diagonal	DNS
2º	DNS	2D-Diagonal
3º	Cannon	Cannon
4º	2D-Cíclico	2D-Cíclico

Tabla 2. Ranking teórico de los algoritmos según el valor de *P*

*6.3.1. **Tiempo paralelo de los algoritmos para P = 8 ó 9***

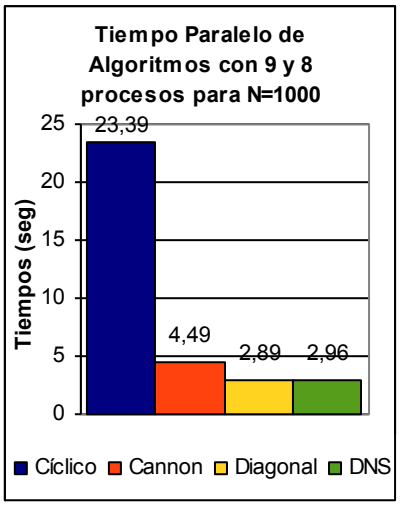

Figura 14(a) Comparación de los 4 algoritmos con N=1000

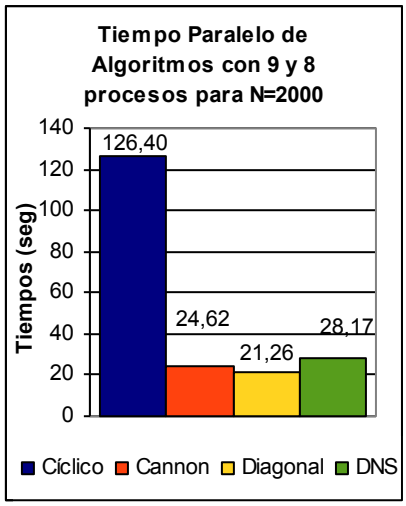

Figura 14(b) Comparación de los 4 algoritmos con N=2000

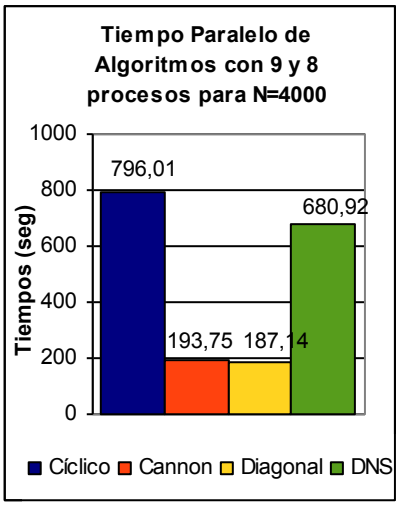

Figura 14(c) Comparación de los 4 algoritmos con N=4000

Podemos notar en las Figuras 14(a) y 14(b) que el patrón del ranking se cumple aproximadamente. Sin embargo, para la Figura 14(c), a pesar de también cumplirse, la diferencia es muy grande en contra del algoritmo DNS. Se estima que el motivo de esto es la diferencia en los valores de N y P. En DNS se tiene un tamaño de matriz mayor, por lo que el cómputo realizado es mayor, el tamaño de cada mensaje es mayor y, se tiene un proceso menos. Aún así, creemos que la diferencia es bastante grande, más aún al ser comparable con el 2D Cíclico. Se realizaron 3 corridas para el DNS de manera a verificar si existió algún inconveniente en particular, obteniendo siempre similares resultados.

*6.3.2. **Aceleración de los algoritmos para P = 8 ó 9***

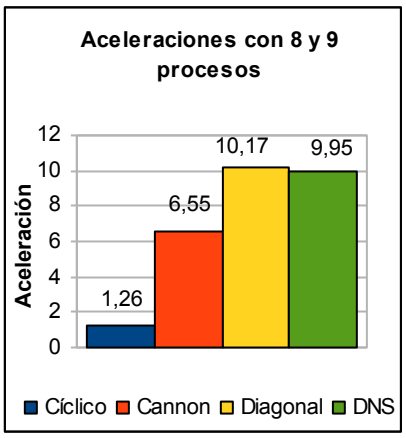

Figura 15(a) Aceleraciones de los 4 algoritmos con N=1000

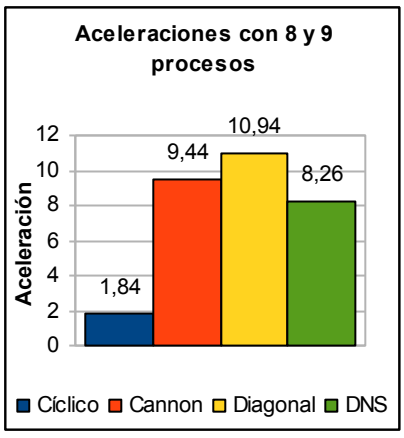

Figura 15(b) Aceleración de los 4 algoritmos con N=2000

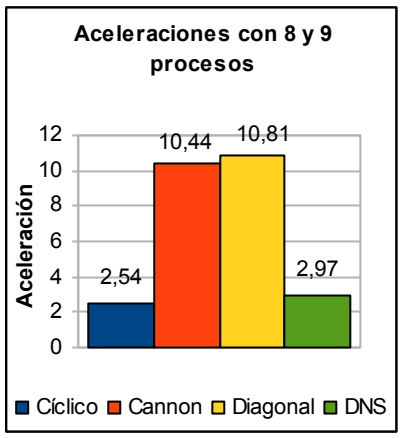

Figura 15(c) Aceleración de los 4 algoritmos con N=4000

Lo llamativo en las aceleraciones de las Figuras 15(a), 15(b) y 15(c) es que en ciertos casos es superior a *P*. Según [1], esto recibe el nombre de *aceleración superescalar*. En este caso, este fenómeno puede deberse a que los datos requeridos para los algoritmos paralelos son menores para cada proceso en comparación con el algoritmo secuencial. Los datos utilizados para el algoritmo secuencial pueden no caber completamente en memoria, lo que puede reducir el rendimiento debido a la paginación realizada. Obviamente este tipo de aceleración solo puede ser alcanzada ya que se cuenta con una misma cantidad de procesadores y procesos, por lo que el mapeamiento es uno-a-uno.

*6.3.3. **Eficiencia de los algoritmos para P = 8 ó 9***

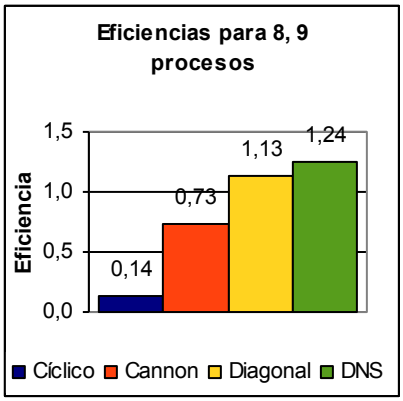

Figura 16(a) Eficiencia de los 4 algoritmos con N=1000

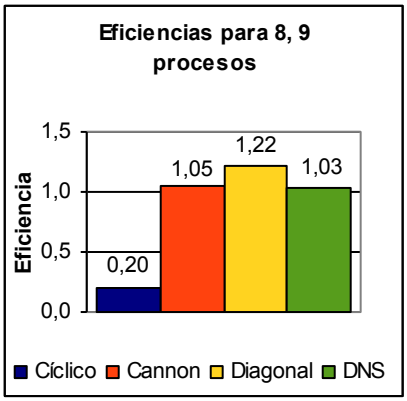

Figura 16(b) Eficiencia de los 4 algoritmos con N=2000

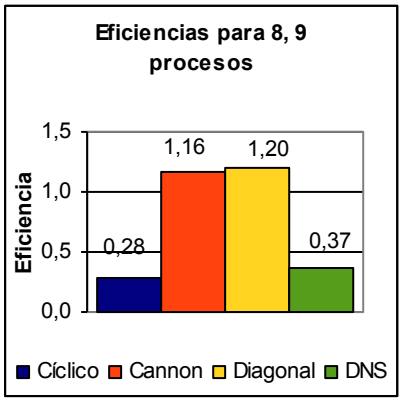

Figura 16(c) Eficiencia de los 4 algoritmos con N=4000

De acuerdo al análisis teórico realizado al inicio de esta sección, en general el algoritmo más eficiente es el algoritmo 2D-Diagonal. Debido a las grandes aceleraciones obtenidas y a que el mapeamiento de procesos a procesadores es de uno-a-uno, las eficiencias son también mayores a la unidad en ciertos casos. Puede notarse que para el caso de *N*=1000, el DNS fue un tanto más eficiente, mejora que no pudo observarse en la aceleración, debido a la diferencia de uno en *P*.

*6.3.4. **Tiempo paralelo de los algoritmos para P = 25 ó 27***

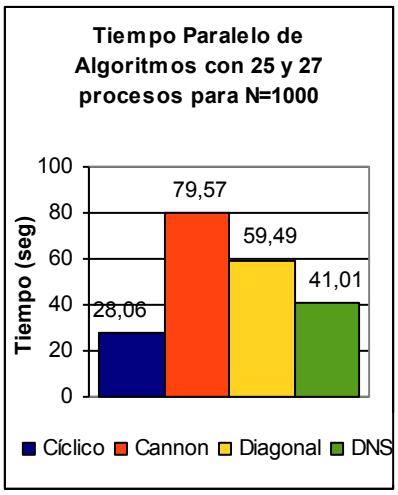

Figura 17(a) Comparación de los 4 algoritmos con N=1000

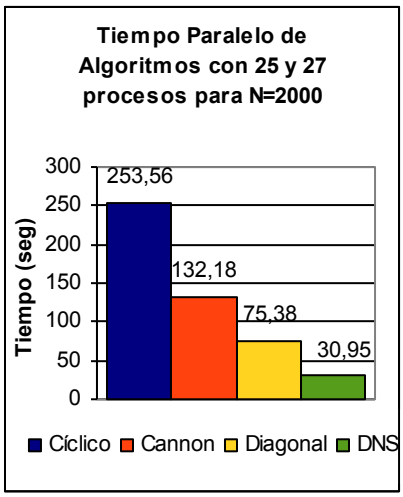

Figura 17(b) Comparación de los 4 algoritmos con N=2000

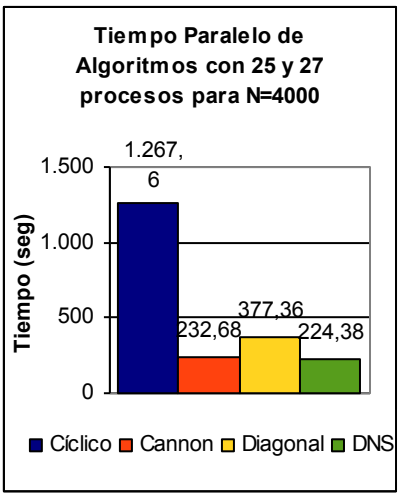

Figura 17(c) Comparación de los 4 algoritmos con N=4000

En la Figura 17(a) puede notarse algo bastante inesperado. Aquí también fueron realizadas 3 corridas para el 2D-Cíclico, obteniendo los mismos resultados. Para el caso de la gran mejora de DNS, el mismo se debe que ahora se ejecutó con 2 procesos más que los demás algoritmos, por lo que el poder de cómputo fue mayor y también se redujo el tamaño de los mensajes en las comunicaciones. Puede notarse, que el ranking teórico también se cumple aproximadamente.

6.3.5. *Aceleración de los algoritmos para P = 25 ó 27*

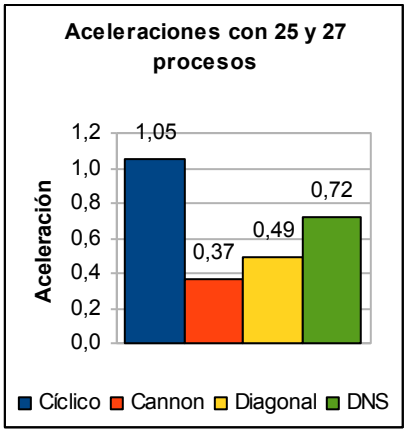

Figura 18(a) Aceleraciones de los 4 algoritmos con N=1000

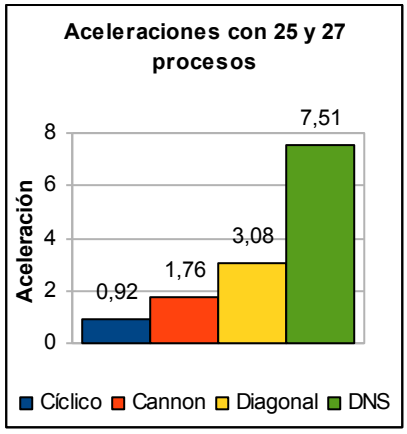

Figura 18(b) Aceleración de los 4 algoritmos con N=2000

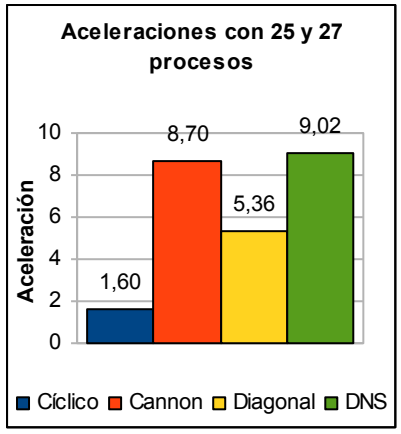

Figura 18(c) Aceleración de los 4 algoritmos con N=4000

Lo notable aquí es que las aceleraciones se redujeron significativamente respecto a las anteriores pruebas. Esto se debe a que el mapeamiento entre procesos y procesadores ya no es uno-a-uno sino aproximadamente tres-a-uno. Sin embargo, tras el crecimiento de *N*, las aceleraciones fueron aumentando. Con esto puede notarse mejor el problema de paginación para el caso del algoritmo secuencial.

6.3.6. *Eficiencia de los algoritmos para P = 25 ó 27*

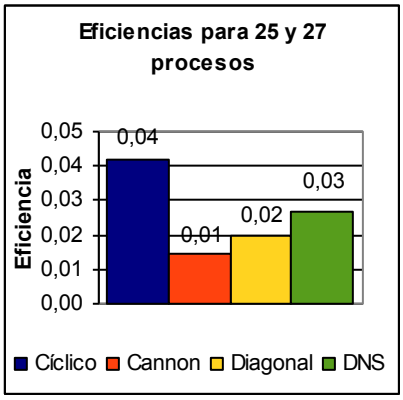

Figura 19(a) Eficiencia de los 4 algoritmos con N=1000

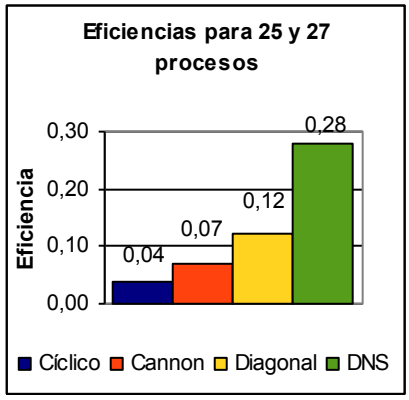

Figura 19(b) Eficiencia de los 4 algoritmos con N=2000

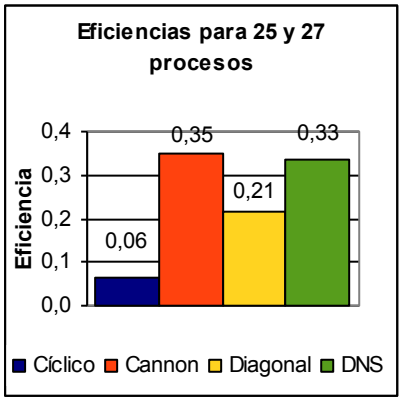

Figura 19(c) Eficiencia de los 4 algoritmos con N=4000

La reducida eficiencia aquí se debe a que a pesar de tener mayor cantidad de procesos, para la ejecución de los algoritmos, el mapeamiento no es uno-a-uno. En las Figuras 17(a), 17(b) y 17(c) se supuso que la mejora de DNS se debió al hecho de contar con mayor cantidad de procesos. Sin embargo, las eficiencias obtenidas aquí demuestran que, de cierta manera, el algoritmo DNS se comportó mejor, lo que apoya el ranking teórico calculado.

7 Conclusiones

A partir de las aceleraciones obtenidas con 8 ó 9 procesos y con 25 ó 27 procesos, se puede observar que el nivel de paralelización hardware soportado es un factor importante para el desempeño de los algoritmos paralelos. Cuando cada proceso se mapea a un procesador (en el caso de 8 ó 9 procesos) la aceleración llega a ser óptima (incluso superescalar). En el caso en que el mapeo sea mayor que uno-a-uno, la aceleración disminuye notablemente según *N* vaya aumentando.

Como es de esperar, las grandes aceleraciones para 8 ó 9 procesos, induce un comportamiento similar para las eficiencias. Para 25 ó 27 procesos, es un tanto más complicado explicar las eficiencias, principalmente porque no se tiene control sobre qué procesos se deberán procesar en cada uno de los procesadores. Suponemos que para el caso de DNS, se realizó una mejor distribución de procesos entre procesadores.

Finalmente, podemos decir, a pesar de las dos anomalías presentadas, que se cumplió aproximadamente el rendimiento teórico que se esperaba de los algoritmos implementados. En general, el 2D Cíclico queda en último lugar por tener mucha mayor cantidad de operaciones de comunicación, todas ellas dirigidas a un solo proceso,

ocasionando el problema que se conoce como contención [1]. En el caso de Cannon, que en general se mantiene tercero, también se tiene bastante interacción entre procesos. En cada una de ellas los procesos deben sincronizarse, probablemente ocasionando esperas innecesarias.

8 Trabajos Futuros

Las pruebas de rendimiento realizadas en el presente trabajo no fueron muchas. Sería interesante poder realizar mayor cantidad de pruebas con mayor cantidad de valores para cada uno de los parámetros (tamaño de matriz, cantidad de procesos, cantidad de procesadores). Así, sería más sencillo encontrar tendencias y arribar a mejores conclusiones, permitiéndonos incluso poder seleccionar qué algoritmo tiene mejor rendimiento dado ciertos valores de parámetros.
Además, se podría plantear un método, para cada algoritmo, que permita la multiplicación de matrices cuadradas de tamaños arbitrarios. De esta manera, los valores de N para cada uno de los algoritmos al realizar las pruebas, podrán ser iguales, permitiéndonos realizar mejores comparaciones.
Otro punto interesante sería probar los algoritmos con mayor cantidad de procesadores, al menos 64 o en su defecto 32 con doble núcleo. La cantidad de 64 es el menor número para el cual la cantidad de procesos para todos los algoritmos puede coincidir. Con la elección de este parámetro, se podrían utilizar tamaños de matrices mucho más grandes y también realizar comparaciones mucho más precisas entre los distintos algoritmos. También, otro punto importante que se debe considerar para la optimalidad es que la topología física de la red utilizada para las pruebas debe aproximarse lo más posible a las topologías cartesianas de cada uno de los algoritmos. De esta manera, las comunicaciones serían mucho más eficientes.
Finalmente, algo muy complejo de solucionar sería el problema de la memoria disponible para la carga de las matrices en el espacio de direccionamiento del programa, ya que para matrices muy grandes el rendimiento podría verse muy afectado por el paginamiento de porciones de las matrices.

Referencias

[1] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Introduction to Parallel Computing, Segunda Edición. Addison-Wesley. Enero 2003.

[2] Himanshu Gupta y P. Sadayappan, Communication efficient matriz multiplication on hypercubes, SPAA ’94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures, 1994, ISBN 0-89791-671-9, págs. 320—329, ACM, New York, NY, USA.

[3] Hyuk-Jae Lee, James P. Robertson y José A. B. Fortes, Generalized Cannon’s algorithm for parallel matrix multiplication, ICS ’97: Proceedings of the 11th international conference on Supercomputing, 1997, ISBN 0-89791-902-5, págs. 44—51, ACM, New York, NY, USA.

[4] Documentación en línea de MPI: http://www-unix.mcs.anl.gov/mpi/www/