Capítulo 2: Multiplicación de Matrices

Casi desde el principio de la aplicación del procesamiento paralelo a los problemas numéricos se han estudiado, diseñado, implementado y experimentado distintas formas de paralelizar la multiplicación de matrices.

Desde el punto de vista del problema mismo, es muy útil tener una optimización de esta operación matricial, ya que siempre es posible encontrarla en las distintas aplicaciones que se deben resolver en el ámbito numérico, y por lo tanto, tener optimizada esta operación implica optimizar una parte de muchas aplicaciones en las cuales sea necesario multiplicar matrices.

Desde un punto de vista más cercano a la investigación, este problema tiene muchas características que lo hacen adecuado para su estudio extensivo e intensivo. Las dos más importantes son su simplicidad y la posibilidad de extender sus resultados a otras operaciones similares.

En este capítulo se incluirán algunos aspectos considerados importantes de esta operación para su paralelización, así como las características sobresalientes de los algoritmos de paralelización ya desarrollados.

2.1 Definición de la Multiplicación de Matrices

La definición de la operación de multiplicación de matrices es muy sencilla, lo que también simplifica su comprensión. Dada una matriz $A^{(m \times r)}$ de m filas y r columnas, donde cada uno de sus elementos se denota a_{ij} con $1 \leq i \leq m$, y $1 \leq j \leq r$; y una matriz $B^{(r \times n)}$ de r filas y n columnas, donde cada uno de sus elementos se denota b_{ij} con $1 \leq i \leq r$, y $1 \leq j \leq n$; la matriz C resultante de la operación de multiplicación de las matrices A y B, $C = A \times B$, es tal que cada uno de sus elementos que se denota como c_{ij} con $1 \leq i \leq m$, y $1 \leq j \leq n$, y se calcula de la siguiente manera

$$c_{ij} = \sum_{k=1}^{r} a_{ik} \times b_{kj} \qquad (2.1)$$

Como la mayoría de las operaciones básicas de álgebra lineal, la cantidad de operaciones entre escalares, (o "flops" en [59], como contracción de "*floating point operations*") necesaria para el cálculo de la matriz resultante es conocida (calculable) de manera exacta. Tal como ha sido definida previamente la multiplicación, son necesarias exactamente

$$cant_op = m \times n \times (2r\text{-}1) \qquad (2.2)$$

operaciones (multiplicaciones y sumas entre escalares). Por razones de simplicidad, normalmente se hace todo el análisis en función de matrices cuadradas de orden n y de esta manera se llega a que la cantidad de operaciones básicas entre escalares es exactamente

$$cant_op = 2n^3\text{-}n^2 \qquad (2.3)$$

Esta cantidad de operaciones es la que usualmente se denomina como *complejidad* de la multiplicación de matrices, y es la que determina el tiempo de ejecución necesario para ser resuelto por una computadora. En este contexto, también lo usual es encontrar que la multiplicación de matrices es $O(n^3)$ ("de orden n^3"), enfatizando que el término dominante en la Ecuación (2.3) es de grado cúbico y dejando de lado las constantes de multiplicación y todos los términos de menor grado.

Es muy importante hacer notar que esta cantidad de operaciones es independiente del algoritmo y/o la computadora que se utilicen para resolver el problema. En este sentido, también es importante (aunque no necesariamente "esencial" en este caso) diferenciar esta cantidad de operaciones de, por ejemplo, la cantidad de operaciones realizadas durante la ejecución de un programa secuencial basado normalmente en la asignación

$$c_{ij} = c_{ij} + a_{ik} \times b_{kj}$$

que implica la ejecución de $2n^3$ operaciones aritméticas entre escalares. Este es un ejemplo inmediato de que los programas no necesariamente resuelven los problemas con la cantidad mínima de operaciones. Como se aclaró antes, este ejemplo no presenta demasiados problemas en cuanto al tiempo de ejecución total, pero sí es útil para mostrar que la

cantidad de operaciones resueltas por una computadora no necesariamente es la mínima. Siempre se debería tener en cuenta este hecho a la hora de evaluar el rendimiento de las computadoras para la resolución de un problema en particular.

En el caso de las computadoras paralelas se debe ser más cuidadoso aún, dado que muchas veces conviene replicar cómputo (y por lo tanto aumentar la cantidad de operaciones efectivamente resueltas por los procesadores) para evitar comunicación o sincronización que llevaría más tiempo de ejecución que el cómputo que se replica. En todo el análisis de rendimiento a realizarse en la experimentación, se utilizará la cantidad de operaciones de la Ecuación (2.3) como valor de referencia para evitar al máximo conclusiones equivocadas por la cantidad de operaciones que se ejecutan en el/los procesadores.

2.2 Operaciones de Algebra Lineal

Casi desde el comienzo mismo de la utilización de computadoras se ha buscado que el software desarrollado tenga la mayor calidad en términos de índices considerados claves, tales como: rendimiento, reusabilidad y portabilidad. En este sentido, el área encargada de resolver problemas numéricos en general y los problemas de álgebra lineal en particular no han sido una excepción.

En el contexto de las operaciones de álgebra lineal, desde hace mucho tiempo se han definido, propuesto y desarrollado varias bibliotecas destinadas al establecimiento de un conjunto lo más reducido y también lo más general posible de rutinas u operaciones básicas que se utilicen en la mayoría (sino en todas) las aplicaciones de álgebra lineal. Uno de los primeros ejemplos es EISPACK [46], basado en un conjunto de rutinas enumeradas en [145].

La biblioteca que se ha convertido en el estándar de facto en el área de álgebra lineal es LAPACK (Linear Algebra PACKage), desarrollada a finales de la década de 1980 [36] [7] [8]. Además de haber aprovechado la experiencia de bibliotecas anteriores como EISPACK y LINPACK, junto con LAPACK (o al menos como una evolución lógica relacionada con esta biblioteca) se agregan al menos dos conceptos fundamentales en cuanto a claridad de la especificación misma de la biblioteca y también en cuanto a la posibilidad de la máxima optimización local (de acuerdo a la arquitectura de cómputo). Estos dos conceptos son:

- Operaciones básicas en niveles.
- Algoritmos de bloque.

En realidad ambos conceptos están muy relacionados, pero la división de las operaciones básicas por niveles se hace desde el punto de vista mismo de LAPACK como biblioteca de resolución de problemas de álgebra lineal. Por otro lado, los algoritmos de bloque son una consecuencia de reconocer que la arquitectura de la mayoría de las computadoras (independientemente de que sea paralela o no) tienen una estructura de memoria jerárquica, donde los niveles más cercanos al procesador mismo (niveles 1 y 2 de cache) deberían ser explotados al máximo para obtener la máxima capacidad de procesamiento.

2.2.1 BLAS: Basic Linear Algebra Subprograms y Rendimiento

A partir de las subrutinas incluidas y definidas en LAPACK, un conjunto de subprogramas ha sido reconocido como básico y de hecho a estas subrutinas se las ha denominado Basic Linear Algebra Subprograms (BLAS) [80] [81] [43]. BLAS normalmente se divide en tres clases (conocidas como niveles), en función de la cantidad datos sobre los que operan y en función de la cantidad de operaciones que requiere cada una de ellas. Se tienen tres niveles de BLAS [46]:

- Nivel 1 (o L1 BLAS): para subrutinas que operan entre vectores, tal como $y = \alpha x + y$
- Nivel 2 (o L2 BLAS): para subrutinas que realizan operaciones con matrices y vectores, tal como $y = \alpha \mathrm{A}x + \beta y$
- Nivel 3 (o L3 BLAS): para subrutinas que operan con matrices, tal como $\mathrm{C} = \alpha \mathrm{AB} + \beta \mathrm{C}$

donde A, B, y C representan matrices, x e y representan vectores y α y β representan escalares.

Más allá de la utilidad de esta clasificación para la caracterización e identificación de las operaciones, se estableció teniendo en cuenta que:

- La cantidad de datos sobre los que operan las subrutinas de nivel 1 es de $O(n)$, donde n representa la longitud de un vector, y la cantidad de operaciones básicas entre escalares también es de $O(n)$.
- La cantidad de datos sobre los que operan las subrutinas de nivel 2 es de $O(n^2)$, donde n representa el orden de matrices cuadradas (cantidad de filas y columnas), y la cantidad de operaciones básicas entre escalares también es de $O(n^2)$.
- La cantidad de datos sobre los que operan las subrutinas de nivel 3 es de $O(n^2)$, donde n representa el orden de matrices cuadradas (cantidad de filas y columnas), y la cantidad de operaciones básicas entre escalares es de $O(n^3)$.

Esto implica que, por un lado, las subrutinas incluidas en BLAS de nivel 3 son las que tienen mayor requerimiento en cuanto a capacidad de procesamiento. De hecho, la diferencia con BLAS de nivel 2 es tan grande, $O(n^3)$ vs. $O(n^2)$, que la mayoría de las veces (sino todas), optimizando BLAS de nivel 3 se tiene toda la optimización necesaria. Por otro lado, es claro que en cuanto a la optimización de las subrutinas, las de BLAS de nivel 3 son las más apropiadas, dado que tienen mayores requerimientos de cómputo que los otros dos niveles. Normalmente se considera que [46]

- Las subrutinas de L1 BLAS no pueden lograr alto rendimiento en la mayoría de las supercomputadoras. Aún así, son útiles en términos de portabilidad.
- Las subrutinas de L2 BLAS son especialmente apropiadas para algunas computadoras vectoriales (en términos de rendimiento) aunque no en todas, dado el movimiento de datos que imponen entre los distintos niveles de la jerarquía de memoria.
- Las subrutinas de L3 BLAS son las más apropiadas para lograr el máximo rendimiento en las supercomputadoras actuales, donde la jerarquía de memoria juega un rol muy importante en el rendimiento de todo el acceso a los datos que se procesan en la/s CPU/s.

De hecho, aunque LAPACK esté implementado (o pueda ser implementado directamente) en términos de L1 BLAS [46], actualmente se reconoce que está diseñado para explotar al

máximo L3 BLAS [LAPACK] porque actualmente estas subrutinas son casi las únicas con las que se puede lograr un alto rendimiento (cercano al máximo rendimiento de cada procesador utilizado), en las supercomputadoras. Por lo tanto, no cabe ninguna duda que en términos de rendimiento es esencial poner especial atención en las subrutinas definidas como BLAS nivel 3.

¿Qué es lo que hace que las subrutinas de BLAS nivel 3 sean especialmente apropiadas para su optimización? La respuesta tiene que ver con la forma en que se implementan los algoritmos, que se han denominado algoritmos de bloques [42] [5] [85] [104] [20] [144]. Estos algoritmos optimizan los accesos a memoria dado que maximizan la cantidad de operaciones que se llevan a cabo por cada dato que se referencia. En general, organizan el cómputo de manera tal que un bloque de datos es accedido y por lo tanto se asigna implícitamente en memoria cache/s, cambiando lo que sea necesario (orden de las iteraciones, niveles de “loop unroll”, etc.) para que todas (o la mayoría) de las operaciones en las que interviene ese bloque de datos se ejecute inmediatamente y por lo tanto se aproveche al máximo. De esta manera, se logra reducir el tiempo efectivo a memoria dado que de hecho se aumenta al máximo el “cache hit” o la cantidad de veces que un dato al que se hace referencia desde el procesador se encuentra inmediatamente en memoria cache.

Los algoritmos de bloque se pueden adaptar a cada arquitectura (o, más específicamente, a cada jerarquía de niveles de cache/memoria) y por esta razón se suele utilizar el término *transportable* en vez de *portable*. Se tiende a que las rutinas con mayores posibilidades de optimización o con mayor potencial para obtener el máximo rendimiento posible se “adapten” a la arquitectura subyacente [46]. La mayoría de las empresas que diseñan y comercializan procesadores también proveen todas las subrutinas definidas como BLAS (y aún otras similares) de manera tal que aprovechen al máximo la capacidad de procesamiento [CXML] [SML] [SCSL1] [SCSL2]. Se suele indicar que estas subrutinas son optimizadas para los procesadores aún a nivel del lenguaje del procesador (assembly language) y por lo tanto el esfuerzo y el costo puesto en su implementación es también un indicador de la importancia de estas subrutinas.

2.2.2 L3 BLAS y Multiplicación de Matrices

La especificación de L3 BLAS es hecha originalmente para el lenguaje FORTRAN y las las subrutinas definidas/incluidas son [42] [BLAS]:

a. Productos de matrices “generales” (subrutinas con nombres que terminan con GEMM):

   $C \leftarrow \alpha \, op(A) \, op(B) + \beta C$

   donde op(X) puede ser X, X^T o X^H

b. Productos de matrices donde una de ellas es real o compleja simétrica o compleja hermítica (subrutinas con nombres que terminan con SYMM o HEMM):

   $C \leftarrow \alpha AB + \beta C$ o $C \leftarrow \alpha BA + \beta C$

donde A es simétrica para SYMM o hermítica para HEMM y está a izquierda o derecha de la multiplicación dependiendo de un parámetro (SIDE) de la subrutina.

c. Productos de matrices donde una de ellas triangular (subrutinas con nombres que terminan con TRMM):

$B \leftarrow \alpha\, op(A)\, B$ o $B \leftarrow \alpha\, B\, op(A)$

donde A es una matriz triangular, está a izquierda o derecha de la multiplicación dependiendo de un parámetro (SIDE) de la subrutina, y op(A) puede ser A, A^T o A^H.

d. Actualizaciones de rango *k* (rank-*k* update) de una matriz simétrica (subrutinas con nombres que terminan con SYRK):

$C \leftarrow \alpha AA^T + \beta C$ o $C \leftarrow \alpha A^T A + \beta C$

donde C es simétrica y A está a derecha o a izquierda de la multiplicación por A^T dependiendo de un parámetro de la subrutina (TRANS).

e. Actualizaciones de rango *k* (rank-*k* update) de una matriz hermítica (subrutinas con nombres que terminan con HERK):

$C \leftarrow \alpha AA^H + \beta C$ o $C \leftarrow \alpha A^H A + \beta C$

donde C es hermítica y A está a derecha o a izquierda de la multiplicación por A^H dependiendo de un parámetro de la subrutina (TRANS).

f. Actualizaciones de rango 2*k* (rank-2*k* update) de una matriz simétrica (subrutinas con nombres que terminan con SYR2K):

$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ o $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$

donde C es simétrica y A está a derecha o a izquierda de la multiplicación por B^T dependiendo de un parámetro de la subrutina (TRANS).

g. Actualizaciones de rango 2*k* (rank-2*k* update) de una matriz hermítica (subrutinas con nombres que terminan con HER2K):

$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C$ o $C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C$

donde C es hermítica y A está a derecha o a izquierda de la multiplicación por B^H dependiendo de un parámetro de la subrutina (TRANS).

h. Soluciones a sistemas de ecuaciones triangulares (subrutinas con nombres que terminan con TRSM):

$B \leftarrow \alpha$ op(A) B o $B \leftarrow \alpha$ B op(A)

donde A es una matriz triangular, está a izquierda o derecha de la multiplicación dependiendo de un parámetro (SIDE) de la subrutina, y op(A) puede ser A^{-1}, A^{-T} o A^{-H}.

Dejando de lado las operaciones de $O(n^2)$ tales como al cálculo de op(A) = A^T, se puede notar intuitivamente que todas las subrutinas de L3 BLAS tienen como operación predominante (en cuanto a cantidad de operaciones aritméticas) la multiplicación de matrices. Además, en [77] se muestra cómo todo el nivel 3 de BLAS puede ser implementado en términos de la operación de multiplicación de matrices manteniendo el rendimiento cercano al óptimo posible de cada computadora. En el contexto comercial, se puede citar un ejemplo de Intel que, junto con la comercialización del Pentium III puso a disposición en Internet el documento [74] donde explica cómo aprovechar al máximo la capacidad de cálculo del procesador en términos de multiplicación de matrices, además de poner también a disposición de los usuarios del procesador una biblioteca de funciones de cálculo matricial.

2.3 La Multiplicación de Matrices como Benchmark

La caracterización del rendimiento de las computadoras se ha utilizado con varios propósitos, entre los cuales se pueden mencionar [63]:

- Estimación de la capacidad de resolución de problemas, tanto en lo referente al *tamaño* de los problemas que se pueden resolver como al *tiempo de ejecución* necesarios.
- Verificación y/o justificación del costo de las computadoras, no solamente en cuanto al hardware sino también en cuanto al software de base y de aplicación necesarios.
- Elección de la computadora más adecuada para el problema o clase de problemas que se deben resolver. Implícitamente en este caso se utiliza el índice de rendimiento como un parámetro de comparación de las computadoras posibles de utilizar.

Tradicionalmente, la capacidad de cómputo numérico de una computadora se ha caracterizado con la cantidad de operaciones de punto flotante por unidad de tiempo (Mflop/s: millones de operaciones de punto flotante por segundo) o por un número que lo identifique de forma unívoca [64] [SPEC]. Tradicionalmente también se han tomado dos líneas generales para el cálculo de este índice de rendimiento:

1. Análisis del hardware de procesamiento: unidad/es de punto flotante, diseño de las unidades de punto flotante (pipelines, registros internos, etc.), memoria/s cache (niveles, tamaños, etc.), capacidad de memoria principal, etc.
2. Ejecución de un programa o conjunto de programas específicos de cálculo denominados *benchmarks*.

El análisis del hardware de procesamiento normalmente da lugar a lo que se conoce como *rendimiento pico*, o rendimiento máximo *teórico* de la computadora. Esta línea de caracterización del rendimiento ha sido adoptada normalmente por los fabricantes de las computadoras y también ya es aceptado que es muy poco probable de obtener por una aplicación específica.

La utilización de benchmarks se hizo cotidiana dada la separación que puede existir entre el rendimiento pico y el rendimiento real que las aplicaciones obtienen normalmente en su ejecución en las computadoras. Es muy difícil la elección de un conjunto de programas que logren reunir las características de representar a *toda* la gama de posibles aplicaciones que se pueden ejecutar sobre una computadora y por lo tanto existen muchos benchmarks utilizados y aún muchos más propuestos.

Si está bien definido el tipo de aplicaciones específicas sobre el cual se utilizarán las computadoras, sigue siendo muy útil la caracterización en este campo específico de aplicaciones sin utilizar los benchmarks más generales. Este es el caso de las aplicaciones definidas en términos de las multiplicaciones de matrices, y por lo tanto lo más preciso que se puede obtener en este campo es el rendimiento de la multiplicación de matrices misma, que se toma como el *benchmark* de referencia en cuanto a rendimiento.

Utilizar un *benchmark* tan específico y tan cercano a la aplicación que se debe resolver tiene, en el contexto de los programas paralelos que se ejecutan sobre hardware heterogéneo, una ventaja más: define con precisión la velocidad relativa de las computadoras para el procesamiento local. Si bien este índice (velocidad relativa de cómputo) no es tan necesario ni importante en el contexto de las computadoras paralelas con elementos de procesamiento homogéneos, se torna indispensable para el cómputo paralelo con elementos de procesamiento heterogéneos. Sin este tipo de información es muy difícil llegar a tener balance equilibrado de la carga computacional (al menos de manera estática).

2.3.1 "Benchmark" del Nivel 3 de BLAS

Es claro que si se elige implementar todo el nivel 3 de BLAS directamente en términos de la multiplicación de matrices [77] el rendimiento obtenido será casi directamente el de la multiplicación de matrices misma. Pero si se elige implementar cada subrutina (L3 BLAS) aprovechando sus particularidades de cálculo de manera óptima e independiente de la multiplicación de matrices, aún es de esperar que el rendimiento sea muy similar al de la multiplicación de matrices misma. De esta manera, la multiplicación de matrices es un buen "representante" (y con esto se constituye en un *benchmark*) en cuanto a rendimiento de todas las rutinas del nivel 3 de BLAS.

Un argumento quizás más sólido para considerar a la multiplicación de matrices como representativa con respecto a rendimiento de todo el nivel 3 de BLAS es que al menos en el ámbito secuencial está demostrado experimentalmente que el rendimiento obtenible con cada una de las subrutinas del nivel 3 de BLAS es similar al que se puede obtener con la multiplicación de matrices [144]. En este sentido, conociendo el rendimiento que se obtiene con la multiplicación de matrices se puede tener una idea bastante concreta del rendimiento obtenible con todas las subrutinas del nivel 3 de BLAS.

2.3.2 Como Benchmark "General"

En el campo de los benchmarks en general, es decir de los programas que se intentan utilizar para identificar la capacidad de cálculo de las computadoras (sean paralelas o no), la representatividad de la multiplicación de matrices es mucho más discutible. De hecho, existe una gran cantidad de investigadores y empresas que consideran que lo *único* que puede ser un benchmark es una aplicación real [64]. Sin embargo, en algunas distribuciones de benchmarks de uso libre para máquinas paralelas [68] [94] se lo sigue incluyendo al menos como "benchmark de bajo nivel".

De una manera u otra, se siguen reportando los resultados en cuanto a rendimiento de la multiplicación de matrices en computadoras paralelas y en computadoras secuenciales. Una de las razones más importantes consiste en que con la multiplicación de matrices se puede lograr rendimiento cercano al óptimo teórico de la computadora utilizada. En este sentido, la multiplicación de matrices se ha transformado de una manera o de otra en una métrica de calidad de la implementación de algoritmos numéricos o al menos de los algoritmos relacionados con las operaciones de álgebra lineal. Un ejemplo en principio académico lo constituye ATLAS [144] [ATLAS] y, sólo por dar un ejemplo comercial, en [SCSL2] se intenta mostrar cuán *buena* es la biblioteca de cómputo científico provista asegurando que para máquinas monoprocesador el rendimiento excede el 95% teórico y para máquinas paralelas de 64 procesadores el rendimiento relativo global excede el 85% teórico. La idea en este sentido es: "existe código que resuelve al menos un problema de álgebra lineal con rendimiento cercano al óptimo teórico de cada procesador", con la intención de que esta idea se extrapola al menos a un subconjunto de los problemas que se quieren resolver en la/s computadora/s.

2.4 Paralelización de la Multiplicación de Matrices

Quizás por razones académicas (relacionadas con la simplicidad), uno de los primeros algoritmos paralelos que se explican en los libros dedicados a explicar procesamiento paralelo es el de la multiplicación de matrices [56] [82] [79] [146] [10] [58] [52] [3]. Pero más allá de su importancia académica, la investigación se ha mantenido activa a través del tiempo por su importancia como problema a resolver, y esto se demuestra por las numerosas publicaciones al respecto, algunas de las cuales son las mencionadas previamente y otras (en una lista resumida) son [23] [35] [30] [142] [26] [83].

La multiplicación de matrices tiene características muy específicas en lo que se refiere al diseño e implementación de un algoritmo paralelo en el contexto de los algoritmos paralelos en general:

- Cada elemento que se calcula de la matriz resultado C, c_{ij}, es, en principio, independiente de todos los demás elementos. Esta independencia es sumamente útil dado que permite un amplio grado de flexibilidad en lo referente a la paralelización.
- La cantidad y el tipo de operaciones a realizar es independiente de los datos mismos. La excepción en este caso la constituyen los algoritmos de multiplicación de matrices denominadas *ralas* (*sparse*), donde se intenta aprovechar el hecho de que la mayoría de

los elementos de las matrices a multiplicar (y por lo tanto de la matriz resultado) son iguales a cero.
- Regularidad de la organización de los datos y de las operaciones que se realizan sobre los datos. Los datos están organizados en estructuras bidimensionales (las matrices mismas) y las operaciones son básicas, de multiplicación y suma.

La primera característica hace que la multiplicación de matrices sea especialmente apropiada para las máquinas paralelas denominadas multiprocesadores, donde un conjunto de procesadores, o elementos de procesamiento, comparten una misma memoria. Los algoritmos paralelos para multiprocesadores suelen seguir las ideas básicas de descomposición o división de los datos a calcular y/o de *Divide-y-Conquista* (*Divide-and-Conquer*) recursivamente. En general en todos ellos se establece un período previo estático o dinámico de particionamiento o división de la cantidad de datos (o partes de la matriz resultado de la multiplicación) a ser calculados en cada procesador y eventualmente un período posterior de utilización de cálculos intermedios para calcular el resultado final.

Las últimas dos características hacen que los algoritmos propuestos para la multiplicación de matrices en paralelos sigan en general el modelo de cómputo paralelo SPMD (Single Program - Multiple Data) [52] [135]. De esta manera, un mismo programa se ejecuta asincrónicamente en cada procesador de la máquina paralela y eventualmente se sincroniza y/o comunica con los demás procesadores. Es de destacar que el modelo de cómputo SPMD es independiente de que la implementación se haga sobre una máquina paralela multiprocesador o multicomputadora o sobre una máquina paralela con arquitectura de procesamiento tan distribuida como una red de computadoras.

En general, es muy difícil encontrar en las publicaciones (sobre todo en los libros dedicados a explicar cómo llevar a cabo procesamiento paralelo), algoritmos paralelos presentados *para* un determinado tipo de arquitectura de cómputo paralelo. Si bien es cierto que los algoritmos se pueden adaptar de una manera más o menos compleja a cada una de las arquitecturas de procesamiento paralelo disponibles, también es cierto que hay un costo de adaptación a nivel algorítmico y de implementación y, quizás más importante en el contexto de las aplicaciones con grandes requerimientos de cómputo, el costo en términos de rendimiento obtenido puede ser demasiado alto. Es así que en realidad, en la mayoría de los casos, se puede encontrar una estrecha relación entre cada algoritmo y una arquitectura de cómputo paralelo en particular. Es por eso que en el resto de esta sección se asociará directamente cada uno de los algoritmos mencionados con la arquitectura subyacente de cómputo paralelo con la cual se obtendrán los mejores resultados de acuerdo al rendimiento obtenido o posible de obtener.

2.4.1 Algoritmos Paralelos *para* Multiprocesadores

Como se mencionó antes, los algoritmos que siguen los principios de división o descomposición de los cálculos y de “Divide-y-Conquista” recursivamente se consideran como los más apropiados para las máquinas paralelas de memoria compartida o multiprocesadores. De hecho, en el cálculo de $C = A \times B$ lo primero que se puede intentar es directamente dividir el cálculo de C en tanta cantidad de partes como procesadores se puedan utilizar. En este sentido, el hecho de las matrices A y B se acceden solamente para

lectura de sus elementos y la matriz C es la única que se accede para *escritura* (lo que se explicó antes en cuanto a que cada elemento de la matriz resultado se calcula independientemente de los demás) de sus elementos es ampliamente favorable.

Particionamiento Directo. Como lo muestra la Figura 2.1 con una determinada trama para cada procesador P_1, ..., P_4, cada uno de ellos puede acceder a los datos de las matrices A y B sin necesidad de sincronización con los demás (excepto a nivel físico, dependiendo de la organización de la memoria compartida) dado que los datos de ambas matrices se acceden solamente para lectura. De la misma manera, cada procesador puede acceder a la matriz C independientemente de los demás para almacenar cada uno de los elementos que debe calcular en función de los elementos de A y de B.

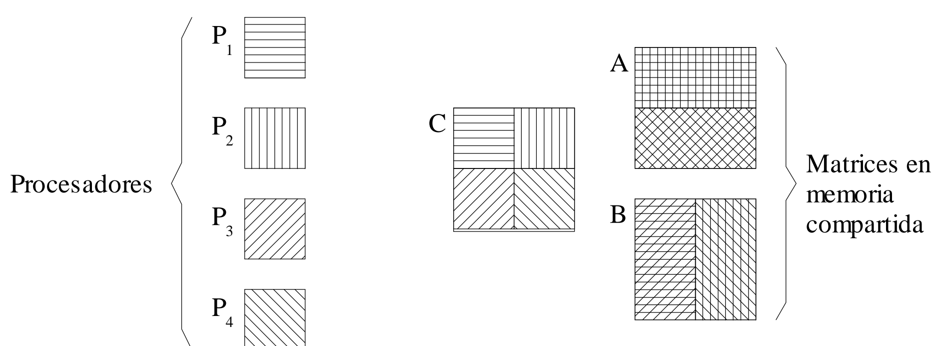


Figura 2.1: División del Cálculo de la Multiplicación en Multiprocesadores.

Esta forma de realizar los cálculos a lo sumo necesitaría una fase inicial para determinar la parte que cada procesador debe procesar y a lo sumo una sincronización final para determinar cuándo todos los procesadores han terminado los cálculos y por lo tanto el resultado está completamente calculado. Por otro lado, no se agrega ningún tipo de replicación de los datos a pesar de que algunas partes de las matrices A y B son utilizadas por más de un procesador dado que todos los datos se almacenan en la memoria compartida.

El hecho de que más de un procesador accede a la misma parte de una matriz (A o B) puede generar inconvenientes referidos a simultaneidad de accesos a memoria. En este sentido, y dependiendo del diseño de la memoria compartida, pueden llegar a ser secuencializados y por lo tanto penalizados en cuanto a rendimiento. Sin embargo, estos problemas son de solución muy simple dado que:

- Se pueden intercalar en el acceso a distintas partes de una misma matriz. En el ejemplo de la Figura 2.1, por ejemplo, el procesador P_1 podría comenzar el acceso a la matriz A desde la primera fila en adelante y el procesador P_2 desde la última fila que le corresponde acceder hacia la primera.
- Los distintos niveles de memoria cache intermedias (entre cada procesador y la memoria principal compartida) más los algoritmos de cálculo de bloques reducen de forma significativa la cantidad de accesos a la memoria compartida.

Es interesante notar la simplicidad de la división, aprovechando las características de la

propia operación de multiplicación más la homogeneidad de los multiprocesadores en cuanto a la capacidad de cálculo de cada elemento de cálculo (procesadores).

Divide-y-Conquista Recursivo. La idea de llevar a cabo la multiplicación por partes o submatrices es aprovechada en este tipo de algoritmos [70] [62] para la paralelización del procesamiento. El algoritmo en pseudo-código puede ser expresado como lo muestra la Figura 2.2 [146]:

```
mat_mul(A, B, C, s)
/* A, B: matrices a multiplicar */
/* C: matriz resultado */
/* s: tamaño de las matrices */
{
   if (multiplicación secuencial)
   {
      C = A×B;
   }
   else
   {
      mat_mul(A00, B00, C000, s/2);    /* (1) */
      mat_mul(A01, B10, C100, s/2);    /* (2) */
      mat_mul(A00, B01, C001, s/2);    /* (3) */
      mat_mul(A01, B11, C101, s/2);    /* (4) */
      mat_mul(A10, B00, C010, s/2);    /* (5) */
      mat_mul(A11, B10, C110, s/2);    /* (6) */
      mat_mul(A10, B01, C011, s/2);    /* (7) */
      mat_mul(A11, B11, C111, s/2);    /* (8) */
   }
   C00 = C000 + C100;
   C01 = C001 + C101;
   C10 = C010 + C110;
   C11 = C011 + C111;
}
```

Figura 2.2: Pseudo-Código de Divide-y-Conquista Recursivo.

donde:

- Cada una de las matrices A, B, y C se divide en cuatro partes iguales, tal como lo muestra la Figura 2.3. Esta cantidad de partes de cada matriz tiene relación directa con la cantidad de llamadas recursivas que se deben llevar a cabo para la obtención de los cálculo intermedios.
- La mayor cantidad de operaciones se lleva a cabo en las llamadas recursivas a `mat_mul` y las últimas cuatro operaciones de suma entre las matrices de cálculos intermedios $C0_{ij}$ y $C1_{ij}$ ($0 \leq i, j \leq 1$) se deben realizar para obtener el resultado correcto de cada una de las submatrices de la matriz C, tal como se puede identificar en la Figura 2.3. Estas últimas operaciones se pueden llevar a cabo en un subconjunto de los procesadores utilizados para resolver las multiplicaciones de las llamadas recursivas.
- Cada una de las llamadas recursivas a `mat_mul` numeradas de (1) a (8) puede ser ejecutada en un procesador diferente, dependiendo de la cantidad de procesadores disponibles y del rendimiento obtenido de acuerdo a la cantidad de datos de las matrices

a multiplicar.
- La condición (`multiplicación secuencial`) puede estar dada en función del tamaño de las matrices a multiplicar (s = 1 en el caso extremo), o la cantidad de llamadas recursivas, que determina a su vez la cantidad de procesadores a utilizar simultáneamente para el cálculo de resultados parciales.

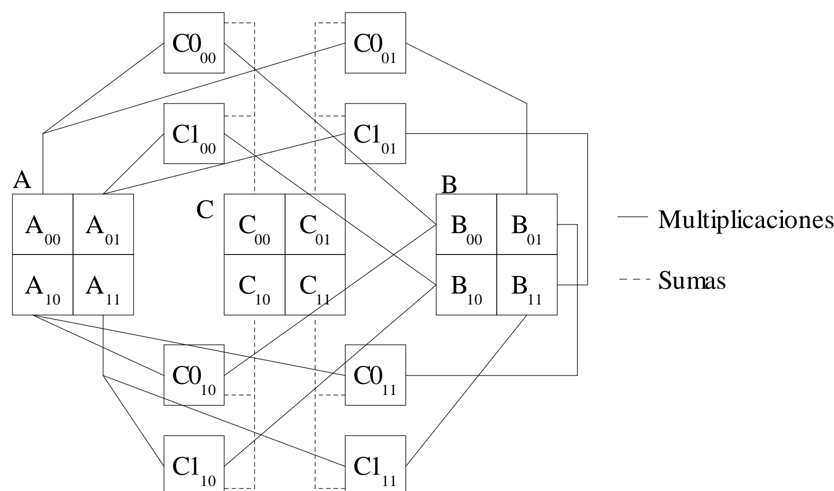


Figura 2.3: Submatrices y Cálculos de Divide-y-Conquista.

Para este algoritmo, tal como para el anterior de particionamiento directo, también es interesante notar que la división de los cálculos (y la consiguiente paralelización) se ve ampliamente favorecida por la homogeneidad en los elementos de procesamiento de los multiprocesadores.

Se debe notar también que, tal como está enunciado, el espacio requerido para los datos aumenta considerablemente teniendo en cuenta que por cada bloque de la matriz resultado C, C_{ij} se tienen dos bloques de datos intermedios $C0_{ij}$ y $C1_{ij}$. Sin embargo, con algunas modificaciones, tal como la reducción del paralelismo en la cantidad de llamadas recursivas o aumentando la dependencia entre los cálculos intermedios con bloques de datos, este requerimiento extra de memoria se puede evitar.

La Figura 2.4 muestra la modificación al pseudo-código de la Figura 2.2 para evitar que la cantidad de memoria requerida sea mayor que la cantidad de memoria requerida para el algoritmo secuencial. De esta manera, se modifica `mat_mul` para que realice una multiplicación y una suma (al estilo de BLAS _GEMM) en vez de una multiplicación solamente, transformándose en `mat_mul_sum`. Todo el procesamiento por bloques se mantiene, aunque ahora se tienen pares de llamadas recursivas a `mat_mul_sum` que utilizan un mismo bloque de C. Estas serían las llamadas numeradas, con (1) y (2), (3) y (4), (5) y (6) y (7) y (8) respectivamente. Esta utilización de un mismo bloque de la matriz C en pares de llamadas recursivas tiene como consecuencias:
- Ya no son necesarios (con respecto a `mat_mul`) dos bloques de datos intermedios para un único bloque de la matriz resultado.
- Las llamadas recursivas numeradas de (1) a (8) ya no son totalmente independientes entre sí, sino que entre pares de llamadas hay dependencias de datos y por lo tanto no

podrían ejecutarse simultáneamente. En este sentido, se reduce de 8 a 4 la cantidad de multiplicaciones que se pueden llevar a cabo simultáneamente (en distintos procesadores).

- Ya no son necesarias las sumas finales que aparecen en la Figura 2.3 porque se resuelven directamente en la misma subrutina `mat_mul_sum`.

```
mat_mul_sum(A, B, C, s) /* C = A×B + C */
/* A, B: matrices a multiplicar */
/* C: matriz resultado */
/* s: tamaño de las matrices */
{
   if (multiplicación secuencial)
   {
      C = A×B + C;
   }
   else
   {
      mat_mul(A00, B00, C00, s/2);    /* (1) */
      mat_mul(A01, B10, C00, s/2);    /* (2) */
      mat_mul(A00, B01, C01, s/2);    /* (3) */
      mat_mul(A01, B11, C01, s/2);    /* (4) */
      mat_mul(A10, B00, C10, s/2);    /* (5) */
      mat_mul(A11, B10, C10, s/2);    /* (6) */
      mat_mul(A10, B01, C11, s/2);    /* (7) */
      mat_mul(A11, B11, C11, s/2);    /* (8) */
   }
}
```

Figura 2.4: Modificación de Divide-y-Conquista Recursivo.

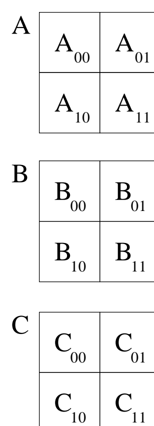
Lo que en el algoritmo de particionamiento directo es la fase inicial de división de las matrices, en este algoritmo serían las llamadas recursivas que serían resueltas por diferentes procesadores.

Paralelización del Método de Strassen. El método de Strassen es uno de los más innovadores en cuanto a la multiplicación de matrices resueltos de manera secuencial [114] y en [59] se lo denomina también como algoritmo de Divide-y-Conquista y se lo presenta además como un algoritmo recursivo. La Figura 2.5-a) muestra los cálculos intermedios asumiendo que las matrices a multiplicar se dividen en cuatro partes o submatrices o bloques iguales como los mostrados en la Figura 2.5-b).

Aunque la cantidad de operaciones aritméticas para este método es un poco más difícil de calcular de manera exacta que para la multiplicación de matrices clásica, se suele calcular (o estimar en términos de órdenes de magnitud), la cantidad de operaciones de multiplicaciones asumiendo que la cantidad de sumas es aproximadamente igual [59]. Con esta consideración, el método de Strassen tiene a su favor que reduce la complejidad o cantidad de cálculos entre números de punto flotante a $O(n^{\log_2 7})$ considerando como referencia el método convencional de multiplicación, que es de $O(n^3)$. También se podría mencionar como una ventaja el hecho de que puede ser implementado utilizando recursión.

$P_0 = (A_{00} + A_{11}) \times (B_{00} + B_{11})$
$P_1 = (A_{10} + A_{11}) \times B_{00}$
$P_2 = A_{00} \times (B_{01} - B_{11})$
$P_3 = A_{11} \times (B_{10} - B_{00})$
$P_4 = (A_{00} + A_{01}) \times B_{11}$
$P_5 = (A_{10} - A_{00}) \times (B_{00} + B_{01})$
$P_6 = (A_{01} - A_{11}) \times (B_{10} + B_{11})$
$C_{00} = P_0 + P_3 - P_4 + P_6$
$C_{01} = P_2 + P_4$
$C_{10} = P_1 + P_3$
$C_{11} = P_0 + P_3 - P_1 + P_5$

a) Cálculos con submatrices



b) Partición de las matrices

Figura 2.5: Método de Strassen.

Desde el punto de vista de la implementación secuencial del método de Strassen:

- Normalmente se puntualiza que las operaciones entre los elementos de las matrices son distintas de las definidas por el método convencional y por lo tanto los efectos de redondeo y estabilidad numérica pueden ser diferentes dependiendo de los valores de los elementos de las matrices a multiplicar [59].
- De forma similar a lo que sucede con el algoritmo mencionado antes como Divide-y-Conquista recursivo, son necesarios datos intermedios para llegar a los valores definitivos a calcular: los bloques P_0, ..., P_6 de la Figura 2.5-a). A diferencia del algoritmo presentado como Divide-y-Conquista recursivo la eliminación de esos bloques de datos intermedios es muy difícil o imposible y de hecho no se considera, por lo que los requerimientos de memoria del método de Strassen son bastante mayores a los del método tradicional.

Desde el punto de vista de la paralelización del método de Strassen:

- Considerando los multiprocesadores de memoria compartida es inmediata, ya que, tal como el método de Divide-y-Conquista recursivo, existen varias multiplicaciones que se pueden llevar a cabo simultáneamente. Específicamente en el caso del método de Strassen son siete: el cálculo de cada P_k, con $0 \leq k \leq 6$.
- Existe cierto desbalance de carga computacional tanto en el cálculo de los bloques intermedios P_i como en el cálculo de los bloques definitivos C_{ij} a partir de los P_k. Para calcular P_0 por ejemplo, son necesarias dos sumas de bloques y una multiplicación, pero para el cálculo de P_1 son necesarias una suma y una multiplicación. Esto afecta tanto a la cantidad de datos que se acceden como a la cantidad de operaciones entre escalares que se deben llevar a cabo. De todas maneras se debe puntualizar que estas diferencias son a nivel de operaciones de $O(n^2)$ (sumas o restas de matrices) frente a operaciones de multiplicación que tienen complejidad de $O(n^3)$ u $O(n^{\log_2 7})$.

Es importante recordar que la idea de dividir las matrices a utilizar/calcular en bloques es intensiva y extensivamente utilizada en todos los algoritmos matriciales secuenciales y también paralelos dado que, como se puntualizó antes, permite la organización del

cómputo por bloques y de esta manera aumenta notablemente la utilización de la/s memoria/s cache/s. Dado que las matrices a procesar normalmente son muy grandes (en general, se tiende a ocupar toda la memoria principal disponible y en algunos casos también el espacio de memoria *swap*) esta organización de los cálculos es imprescindible para obtener rendimiento aceptable. Expresado de otra manera, sin procesamiento por bloques el tiempo de acceso a los datos es varios órdenes de magnitud más grande de lo que el procesador requiere para operar a su máxima velocidad o al menos a una fracción importante de la máxima posible.

2.4.2 Algoritmos Paralelos *para* Multicomputadoras

La mayoría de los reportes de investigación de algoritmos paralelos para la multiplicación de matrices (y problemas similares) corresponden a los diseñados teniendo en cuenta que la arquitectura de cómputo subyacente será la de una multicomputadora [59] [136]. Por un lado, las multicomputadoras siempre han sido consideradas más escalables que los multiprocesadores y por otro lado, el diseño y desarrollo de las multicomputadoras se ha mantenido constante a través del tiempo, lo que las ha hecho más atractivas también para el desarrollo de algoritmos paralelos.

Arreglo Sistólico o Malla de Procesadores. Aunque esta forma de multiplicar matrices es pensada inicialmente para computadoras del tipo SIMD o simplemente para ser implementada directamente en hardware [82] [146], puede ser aplicada en general considerando bloques de matrices tal como en los algoritmos anteriores. La Figura 2.6 muestra la disposición inicial de los datos y los elementos de procesamiento para multiplicar dos matrices de 3×3 elementos.

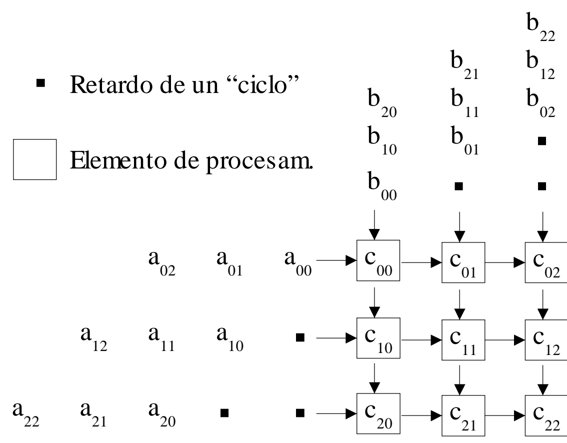


Figura 2.6: Multiplicación de 3×3 en un Arreglo Malla.

Claramente, los elementos de procesamiento o procesadores están interconectados en forma de malla o arreglo bidimensional. Normalmente se cuenta como un "ciclo" o "paso" del procesamiento las operaciones de comunicación que indican las flechas y todas las operaciones de multiplicación y suma que cada elemento de procesamiento pueda realizar

dependiendo de los datos que tenga disponibles.

En la Figura 2.7-a) se muestra el primer paso del procesamiento donde:

- Todos los elementos de las matrices A y B "avanzan" en la dirección de las flechas correspondientes y los elementos a_{00} y b_{00} llegan al procesador dedicado a calcular c_{00}.
- Se lleva a cabo la primera operación para el cálculo de c_{00}, es decir $c_{00}^{(1)} = a_{00} \times b_{00}$ (el superíndice indica la iteración que corresponde).

En la Figura 2.7-b) se muestra el segundo paso del procesamiento donde:

- Una vez más todos los elementos de las matrices A y B "avanzan" en la dirección de las flechas correspondientes, los elementos
  - a_{01} y b_{10} llegan al procesador dedicado a calcular c_{00},
  - a_{00} b_{01} llegan al procesador dedicado a calcular c_{01},
  - a_{10} y b_{00} llegan al procesador dedicado a calcular c_{10}.
- Se llevan a cabo las operaciones posibles en este paso para el cálculo de c_{00}, c_{01} y c_{10}, es decir $c_{00}^{(2)} = c_{00}^{(1)} + a_{01} \times b_{10}; c_{01}^{(2)} = a_{00} \times b_{01}; c_{10}^{(2)} = a_{10} \times b_{00}$.

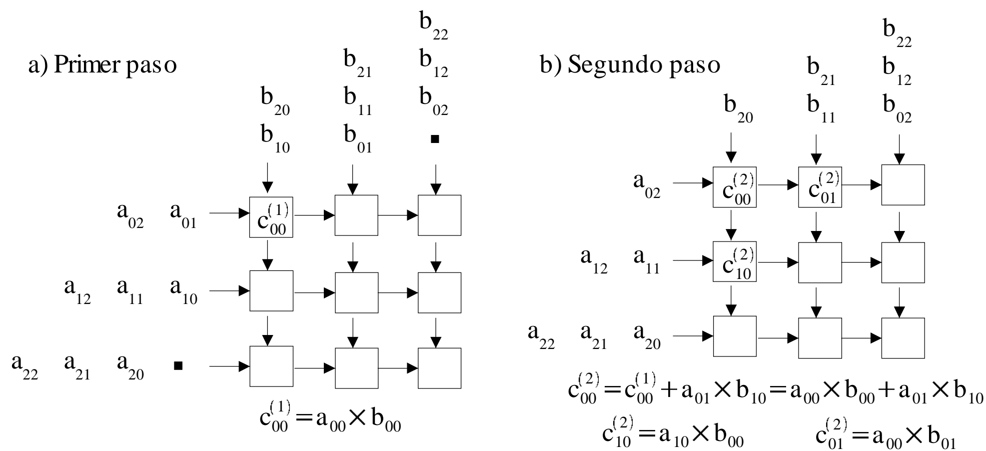


Figura 2.7: Primeros Dos Pasos de la Multiplicación en una Malla.

Para que este tipo de procesamiento llegue a tener rendimiento óptimo se debería tener:

- Todas las comunicaciones se llevan a cabo simultáneamente. En el caso de la Figura 2.6 y la Figura 2.7, esto implica que todos los elementos de procesamiento pueden realizar simultáneamente (solapadamente en el tiempo) hasta:
  - dos comunicaciones para recepción de datos
  - dos comunicaciones para envío de datos
- Cómputo solapado con las comunicaciones, es decir que se realizan operaciones aritméticas a la vez que se envían y reciben datos. En este caso, en los pasos explicados previamente se solapa (realiza simultáneamente) el cómputo de $c_{00}^{(1)} = a_{00} \times b_{00}$ con las comunicaciones correspondientes al segundo paso del procesamiento.
- Operaciones simultáneas de I/O, considerando que las comunicaciones en dirección a la primera fila y primera columna del arreglo bidimensional de la Figura 2.6 y la Figura 2.7 implican I/O. En caso de no contar con I/O de datos en paralelo se deberían tener todos los datos de las matrices inicialmente en la primera fila y primera columna de procesadores, lo que llevaría a su vez a distintos requerimientos de memoria para los

distintos procesadores dependiendo de la posición que ocupan en la malla.

Esta forma de multiplicar matrices es sumamente sencilla por varias razones, siendo las más destacables:

- Simplicidad y naturalidad de la distribución de los cálculos: básicamente un procesador o elemento de procesamiento se dedica exclusivamente al cálculo de una parte de la matriz resultado, sea un elemento o una submatriz o bloque. En este punto una vez más se asume que los elementos de procesamiento son homogéneos y por lo tanto la distribución es trivial.
- Patrón de comunicaciones "fijo" y conocido a priori: todas las transmisiones de datos son punto a punto (entre dos procesadores) y conocidas de antemano en cuanto a cantidad y tipo de transmisiones (cuántos datos y entre cuáles procesadores).
- Requerimientos de memoria iguales para todos los procesadores: no hay ningún procesador que deba tener más o menos datos que los demás, aún si se cuentan los *buffers* necesarios para las comunicaciones, dado que son las mismas en todos los procesadores.

Las primeras dos características hacen que el balance de carga sobre los procesadores (en cuanto a cálculos que se deben llevar a cabo) y sobre la red de interconexión (en cuanto a las transferencias de datos que se deben realizar entre los procesadores) sea muy sencilla.

Algoritmo de Cannon. También está propuesto para un arreglo bidimensional de elementos de procesamiento [23] [79], interconectados como una malla y además con los extremos de cada fila y columna interconectados entre sí, es decir formando la estructura denominada toro tal como lo muestra la Figura 2.8 para 3×3 elementos de procesamiento.

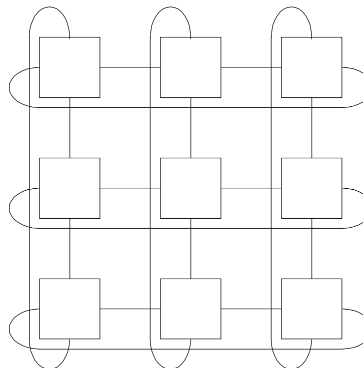
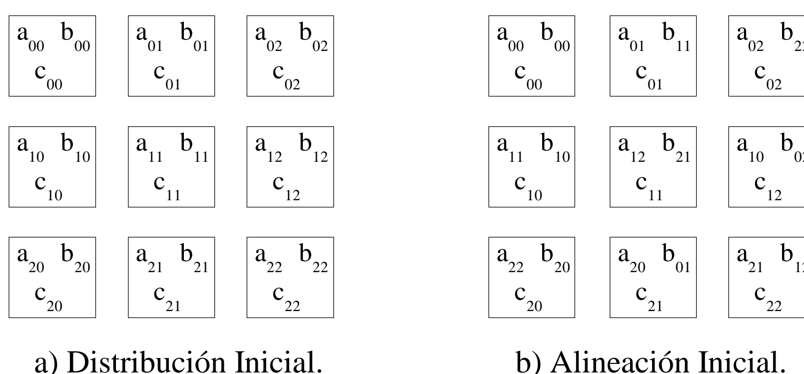


Figura 2.8: Toro de 3×3 Elementos de Procesamiento.

Inicialmente, la distribución de los datos de las matrices A, B, y C es similar a la definida previamente en la malla, es decir que si se numeran los procesadores de acuerdo a su posición en el arreglo bidimensional, el procesador P_{ij} ($0 \leq i, j \leq$ P-1), contiene los elementos o bloques de la posición *ij* ($0 \leq i, j \leq$ P-1), de las matrices A, B y C. Para simplificar la explicación, se utilizarán elementos de matrices en vez de bloques. A partir de esta distribución de datos, se "realinean" o reubican los datos de las matrices A y B de forma tal que si se tiene un arreglo bidimensional de P×P procesadores, el procesador P_{ij}

tendrá asignado el elemento o submatriz de A en la fila *i* y columna (*j*+*i*) mod P, $a_{i,(j+i)\text{modP}}$, y también el elemento o submatriz de B en la fila (*i*+*j*) mod P y columna *j*, $b_{(i+j)\text{modP},i}$. Puesto de otra manera, cada dato de la fila *i* ($0 \leq i \leq$ P-1) de elementos o submatrices de A se traslada o rota (*shift*) hacia los procesadores de la izquierda *i* veces y cada dato de la columna *j* ($0 \leq j \leq$ P-1) de elementos o submatrices de B se traslada o rota (*shift*) hacia los procesadores de arriba *j* veces. En la Figura 2.9-a) se puede ver la asignación inicial, y en la Figura 2.9-b) la reubicación inicial, impuesta por el algoritmo de Cannon para matrices de 3×3 elementos en un toro de 3×3 procesadores (para simplificar no se muestran en la figura las interconexiones).



a) Distribución Inicial. b) Alineación Inicial.

Figura 2.9: Ubicación de 3×3 Datos para el Algoritmo de Cannon.

A partir de la reubicación inicial, se realizan iterativamente los pasos de:

- Multiplicación local de los datos asignados en cada procesador para el cálculo de un resultado parcial.
- Rotación a izquierda de los elementos o submatrices de A.
- Rotación hacia arriba de los elementos o submatrices de B.

y después de P de estos pasos se tienen los valores de la matriz C totalmente calculados.

Resumiendo, las características sobresalientes de esta forma de realizar la multiplicación de matrices son:

- Por la forma en que se comunican los datos de las matrices A y B se puede decir que este es un algoritmo de "alineación inicial y rotaciones".
- El balance de carga tanto en lo referente a cómputo como a comunicaciones está asegurado siempre que los elementos de procesamiento sean homogéneos.
- Como en el procesamiento definido para la grilla de procesadores, el tiempo de ejecución se minimiza si se tiene la capacidad de solapar en el tiempo cómputo con comunicaciones.
- La distribución de los datos de las matrices A y B no es la inicial al terminar el cálculo de la multiplicación de matrices.

Es de destacar que la última características de las enumeradas previamente se torna bastante importante dado que, como se ha mencionado anteriormente y como con todas las rutinas de L3 BLAS, se supone que son parte de alguna aplicación y no necesariamente la aplicación misma. Por lo tanto, cualquier procesamiento siguiente deberá tener en cuenta esta nueva distribución de las matrices, o después del algoritmo de Cannon se deberán realinear las matrices para recuperar la asignación inicial de los datos.

Algoritmo de Fox. También está propuesto para un arreglo bidimensional de elementos de procesamiento [57] [56] [58] [79], interconectados como una malla y además con los extremos de cada fila y columna interconectados entre sí, es decir formando la estructura denominada toro. Una vez más, la distribución de los datos de las matrices A, B, y C es similar a la definida antes, es decir que si se numeran los procesadores de acuerdo a su posición en el arreglo bidimensional, el procesador P_{ij} ($0 \leq i, j \leq$ P-1), contiene los elementos o bloques de la posición *ij* ($0 \leq i, j \leq$ P-1), de las matrices A, B y C, tal como se muestra en la Figura 2.9-a) y en la Figura 2.10-a). En este caso, no se define ningún paso inicial de reubicación de los datos, sino que a partir de aquí se define iterativamente el algoritmo. En la iteración o paso *k*:

- El bloque o dato de la matriz A almacenado en el procesador de la posición *i*, *i*+*k* mod P (fila *i*, columna *i*+*k* mod P) se envía a todos los procesadores de la fila *i* (su misma fila) en un *broadcast por fila* del arreglo bidimensional.
- Se multiplican en cada procesador los datos de A recibidos con los datos de B almacenados localmente, obteniendo un resultado parcial de la matriz C.
- Se rotan los datos de B hacia arriba tal como en el algoritmo de Cannon.

La Figura 2.10 muestra el procesamiento correspondiente al primer paso del algoritmo de Fox en un arreglo bidimensional de 3×3 procesadores, y la Figura 2.11 muestra el procesamiento correspondiente al segundo paso del algoritmo de Fox en un arreglo bidimensional de 3×3 procesadores.

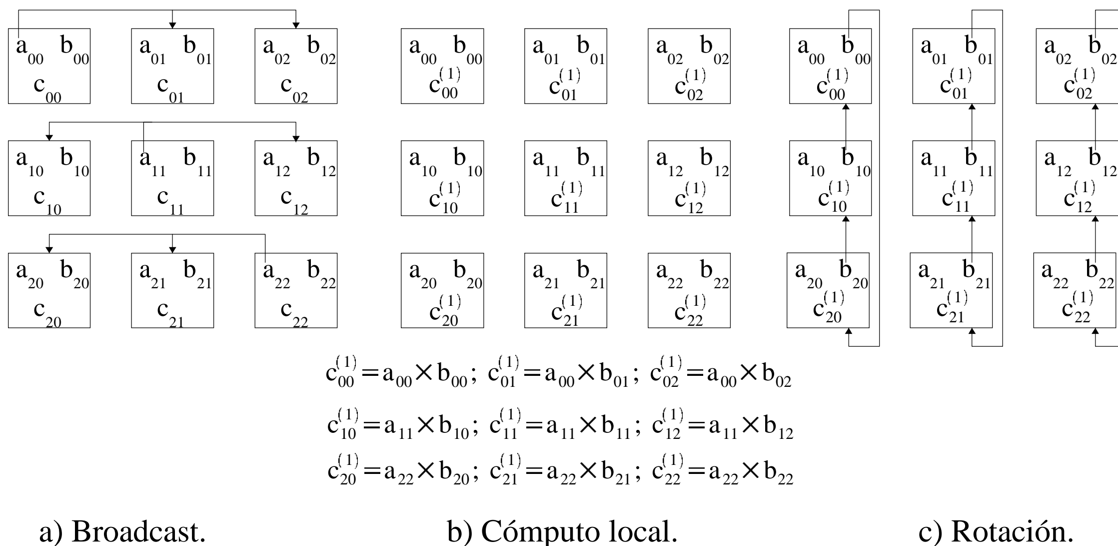


Figura 2.10: Primer paso del Algoritmo de Fox, Matrices de 3×3.

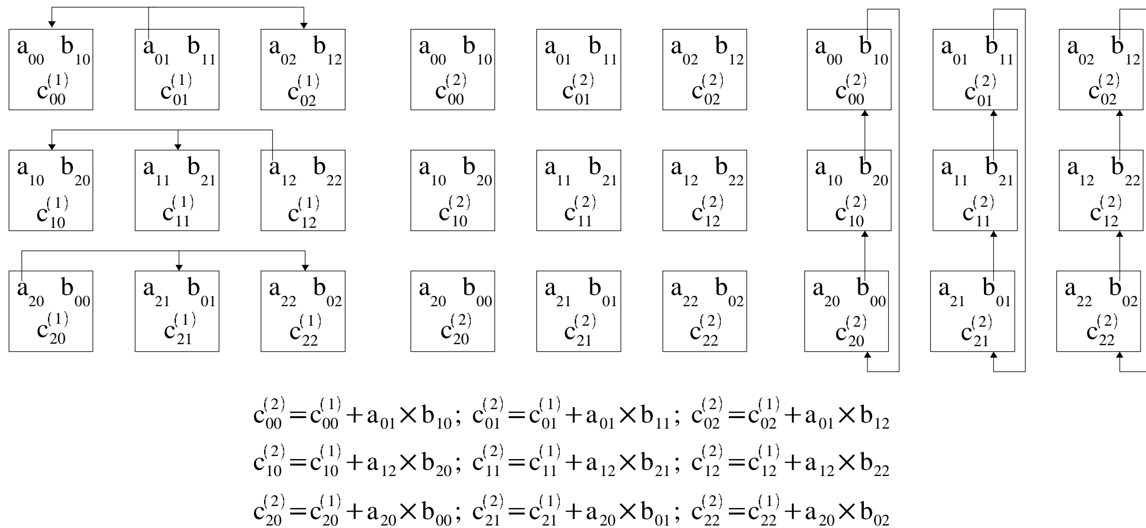
Las principales características del algoritmo de Fox que se pueden enumerar son:

- Como en los casos anteriores, el balance de carga en cuanto a cómputo que debe realizar cada procesador es muy sencillo asumiendo que los procesadores son homogéneos.
- El balance de carga en cuanto a la red de intercomunicaciones no es tan sencillo de analizar como en el algoritmo de Cannon, ya que se deben tener en cuenta no solamente comunicaciones punto a punto sino también comunicaciones colectivas: broadcasts por

filas. De todas maneras, se tendría balance de carga de la red de comunicaciones en cuanto a comunicaciones por columnas, dado que son todas punto a punto; y por filas, dado que son todas comunicaciones broadcast. Desde el punto de vista de las comunicaciones que se generan en cada procesador, todos generan la misma carga sobre la red de interconexión:

- un envío de datos de la matriz B hacia arriba
- una recepción de datos de la matriz B desde abajo
- un envío o una recepción de broadcast de datos de la matriz A hacia los demás procesadores de fila o desde un procesador de la misma fila respectivamente.

- Dado que todos los procesadores tienen la misma cantidad y tipo de comunicaciones, en caso de ser necesarios buffers, éstos serán los mismos en todos los procesadores.
- Los requerimientos de memoria en cada procesador son mayores en comparación con el algoritmo de Cannon, ya que todos los procesadores deberían tener:
    - Un bloque o submatriz de la matriz A local
    - Un bloque o submatriz de la matriz B local
    - Un bloque o submatriz de la matriz C local
    - Un bloque más, para recibir el broadcast por filas del bloque de A que en cada paso se comunica.
- A diferencia del algoritmo de Cannon, al finalizar la multiplicación de matrices, los datos de cada una de las matrices queda como en la asignación inicial.



a) Broadcast. b) Cómputo local. c) Rotación.

Figura 2.11: Segundo paso del Algoritmo de Fox, Matrices de 3×3.

En lo que se refiere a las comunicaciones broadcast por filas, se debe tener en cuenta que tanto en una malla como en un toro las únicas comunicaciones que se podrían denominar *predefinidas* son las que se llevan a cabo entre procesadores *vecinos*. En este sentido, se requiere algún esfuerzo adicional posiblemente penalizado en cuanto a rendimiento, para realizar cualquiera de las comunicaciones colectivas, incluyendo el broadcast por filas, que también puede ser definido como un multicast desde el punto de vista de la red completa de comunicaciones. En este sentido, existen muchas publicaciones que se dedican a

implementar o de alguna manera adaptar el algoritmo para la red de interconexión estática toro [30][142][26]. La idea básica consiste en implementar un broadcast vía múltiples comunicaciones punto a punto solapadas, es decir que mientras un broadcast se está llevando a cabo con comunicaciones punto a punto pueden comenzar las transmisiones punto a punto para el broadcast siguiente. Este tipo de adaptaciones al algoritmo de Fox es la que finalmente se implementa en la biblioteca ScaLAPACK [21].

También es posible reducir el requerimiento de memoria, que en principio puede ser bastante mayor que el definido en el algoritmo de Cannon (más específicamente, 33.3...% mayor). La clave de la reducción está dada en la comunicación por subbloques de los datos de la matriz A asignada en cada procesador. En vez de hacer un broadcast de todos los datos de la matriz A se podrían, por ejemplo, hacer dos broadcasts cada uno con la mitad de los datos de la matriz A asignada localmente. Esto automáticamente reduce el requerimiento extra (tomando como referencia el algoritmo de Cannon) de memoria a la mitad de los datos de la matriz A que cada procesador tiene localmente. De la misma manera, se pueden hacer diez broadcasts, cada uno con la décima parte de la matriz A asignada localmente y esto reduce a la décima parte la cantidad *extra* de memoria necesaria.

Es de destacar que al finalizar el procesamiento, los datos de las matrices quedan asignados en cada procesador exactamente igual que antes de comenzar la multiplicación de las matrices. En este sentido, recordando lo que se mencionó antes en el algoritmo de Cannon, que la multiplicación de matrices se debe hacer en el contexto de otras operaciones, es una ventaja considerable.

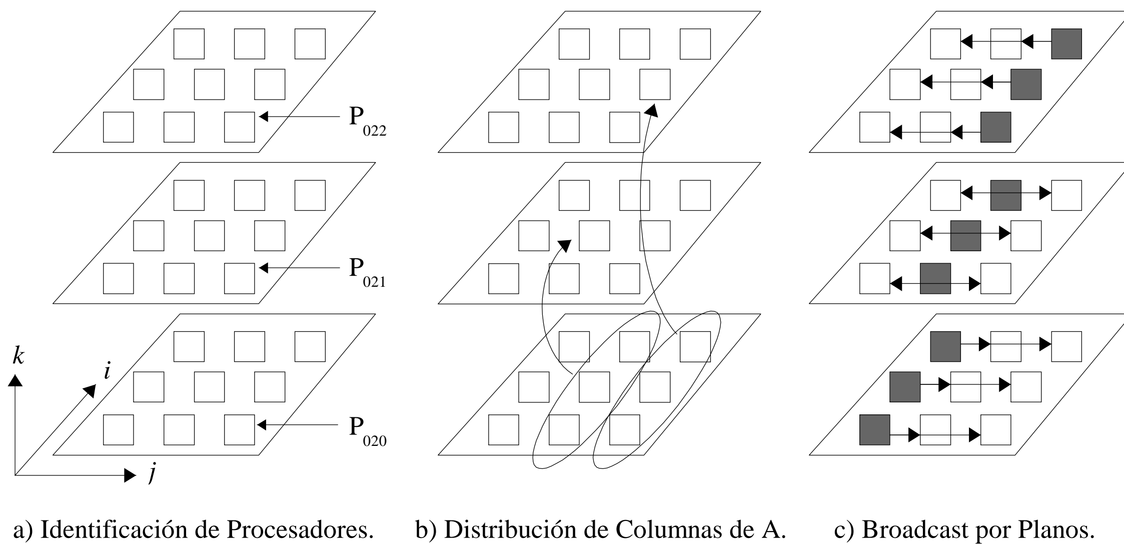
De hecho, tanto para resolver el problema de hacer un broadcast en un toro como para reducir la cantidad de memoria necesaria para almacenar los datos localmente se utiliza el mismo principio que para acelerar el acceso a los datos en la jerarquía de memoria asumiendo la existencia de uno o más niveles de memoria cache: procesamiento por bloques. En vez de manejar todos los datos de las matrices A, B y C asignados localmente, cada una de las submatrices se considera por bloques, y son estos bloques los que se procesan localmente (haciendo las multiplicaciones de matrices parciales), y también son los que se comunican a través de las conexiones punto a punto en una misma fila de procesadores formando un pipeline de comunicaciones con el que finalmente se resuelve el broadcast. Por lo tanto, dado que son subbloques o partes de la matriz A asignada localmente, la cantidad de memoria extra que se necesita en cada procesador se reduce a un subbloque o conjunto de subbloques que se transmite en una misma comunicación punto a punto entre procesadores de una misma fila.

Método DNS y Mallas de Arboles. El algoritmo DNS (por las letras iniciales de los apellidos de sus autores: Dekel, Nassimi, Sahni) [35] [100] [79] propone multiplicar matrices cuadradas de orden n en una multicomputadora con sus procesadores interconectados en un hipercubo de tres dimensiones. Una vez más, la descripción básica del procesamiento se hará para elementos de matrices, pero su extensión/aplicación a bloques o submatrices es inmediata.

Como punto de partida para el algoritmo DNS, se focaliza la atención directamente en cada

producto de elementos de matrices, $a_{ik}\times b_{kj}$, que son exactamente n^3 operaciones entre escalares en el caso de multiplicar matrices cuadradas de orden n. Teniendo n^3 procesadores, cada procesador se hace cargo de exactamente una multiplicación y luego se deben sumar los valores obtenidos de cada multiplicación de la manera correcta para obtener cada uno de los n^2 elementos de la matriz resultado. Más específicamente, numerando los procesadores como si estuvieran organizados en un arreglo de tres dimensiones (es decir $n\times n\times n$ procesadores), el procesador P_{ijk}, $0 \leq i, j, k \leq (n\text{-}1)$, es el encargado de llevar a cabo la multiplicación $a_{ik}\times b_{kj}$. Luego, acumulando (sumando) los valores $P_{i,j,0}...P_{i,j,n-1}$ se obtiene el valor c_{ij} de la matriz resultado.

Si se consideran los datos (elementos o submatrices) de las matrices distribuidos en los procesadores P_{ij0} (es decir que a_{ik}, b_{kj} y c_{ij} se asignan el procesador P_{ij0}), antes de realizar las multiplicaciones se deben distribuir/replicar los datos. Para este paso también conviene visualizar los procesadores como si estuvieran organizados en un arreglo de tridimensional, tal como lo muestra la Figura 2.12-a) para veintisiete procesadores, es decir como sucesivos planos de 3×3 procesadores (correspondientes a los índices ij) superpuestos para distintos valores de k.



a) Identificación de Procesadores. b) Distribución de Columnas de A. c) Broadcast por Planos.

Figura 2.12: Datos en DNS sobre un Arreglo de 3×3×3 Procesadores.

A partir de la asignación inicial de elementos de matrices a los procesadores P_{ij0}, en el plano inferior en la Figura 2.12-a), la columna j-ésima de la matriz A se envía a los procesadores correspondientes (columna j-ésima de procesadores) en el plano j-ésimo, es decir desde los procesadores P_{ij0} a los procesadores P_{ijj} ($0 \leq i \leq n\text{-}1$, $1 \leq j \leq n\text{-}1$), como lo muestra la Figura 2.12-b). Como último paso en la distribución de los datos de la matriz A, los procesadores en cada plano con los datos de A, P_{ijj} ($0 \leq i, j \leq n\text{-}1$), hacen un broadcast por filas de procesadores tal como lo muestra la Figura 2.12-c) y de esta manera se logra que en cada P_{ijk} se tenga el elemento a_{ik}. La forma de distribuir los datos de la matriz B es análoga pero en vez de distribuir por columnas se hace por filas. Una vez que todos los procesadores tienen los datos, pueden ejecutar la multiplicación y luego se suman los resultados en los procesadores $P_{i,j,0}...P_{i,j,n-1}$ tal como se mencionó previamente.

Las características sobresalientes de esta forma de calcular la multiplicación de matrices se pueden resumir en:

- Puede multiplicar matrices de orden n en $O(\log(n))$ pasos.
- Utiliza (hasta) n^3 procesadores, aunque está explicado para elementos de matrices se puede aplicar a submatrices o bloques.
- Un mismo dato de cada matriz A se utiliza y por lo tanto se copia en toda una fila de procesadores, tal como lo muestra la Figura 2.12-c), y de forma análoga un mismo dato de la matriz B se copia en toda una columna de procesadores, y por lo tanto se tienen numerosos datos replicados.
- Está claramente orientado a computadoras paralelas con procesadores interconectados como si estuvieran en una estructura tridimensional, y de hecho esto produce la reducción en la cantidad de cálculos a $O(\log(n))$ y a su vez la replicación de datos en varios procesadores.

Estas características son similares a las que tiene el método de multiplicación de matrices sobre una malla de árboles de procesadores [82] [98]. De hecho, este método también sigue la secuencia:

- Asignación inicial de los n^2 elementos de cada matriz a n^2 procesadores o elementos de procesamiento.
- Comunicación-Replicación de los elementos de las matrices para que cada procesador tenga los datos necesarios para hacer las multiplicaciones $a_{ik} \times b_{kj}$.
- Suma de los resultados de las multiplicaciones para obtener los elementos de la matriz C.

2.5 Resumen del Capítulo

Los temas más importantes que se han introducido y/o explicado en este capítulo, además de la definición misma de la multiplicación de matrices y la cantidad de operaciones necesarias para su resolución, son:

- La identificación del contexto de las aplicaciones de álgebra lineal, dada por la biblioteca LAPACK, las operaciones básicas incluidas en BLAS y, aún más específicamente, la relación entre la multiplicación de matrices. No solamente la multiplicación de matrices es similar al resto de las operaciones de L3 BLAS en cuanto a requerimientos de cómputo y memoria, sino que está demostrado que todas las operaciones de L3 BLAS se puede implementar en utilizando la multiplicación de matrices.
- La utilización de la multiplicación de matrices como benchmark. Si bien es muy acotada su validez en general, sí se puede afirmar que es un buen benchmark de todo L3 BLAS. Además, es utilizada para mostrar la calidad de optimización que se puede lograr cuando se relaciona el rendimiento obtenido con el máximo teórico del hardware utilizado.
- Se describieron los algoritmos paralelos para llevar a acabo la multiplicación de matrices en multiprocesadores. En general son sencillos en cuanto a descripción como en cuanto a su análisis teórico de rendimiento.

- Se describieron también los algoritmos paralelos para multiplicación de matrices en multicomputadoras. Normalmente se puede identificar claramente la relación de cada algoritmo con la interconexión de procesadores *subyacente* (en la cual el algoritmo obtiene su mejor rendimiento). También se describieron las adaptaciones del algoritmo de Fox en particular a las redes de interconexión de procesadores estáticas y bidimensionales (grillas y toros), que es el que se implementa en la biblioteca ScaLAPACK.
- Todos los algoritmos paralelos comparten características comunes relativamente importantes a nivel conceptual:
  - Adoptan el modelo de procesamiento SPMD.
  - Asumen que los nodos de procesamiento son homogéneos y *aprovechan* esta homogeneidad para obtener balance de carga.
  - Utilizan el procesamiento por bloques para optimizar la jerarquía de memoria de las computadoras, específicamente orientados a la optimización en el acceso a los datos en memoria/s cache/s.

Se necesita conocer cada uno de estos puntos para, una vez que se haga la descripción del hardware de procesamiento paralelo que proveen las redes locales, como mínimo se pueda analizar si los algoritmos de cómputo paralelo para multiplicar matrices son apropiados o no.