Paralelización y Speedup Superlineal en Supercomputadoras Ejemplo con Multiplicación de Matrices

Fernando G. Tinetti
Becario de Formación Superior UNLP
fernando@ada.info.unlp.edu.ar

Mónica Denham
Becaria Alumna LIDI
mdenham@lidi.info.unlp.edu.ar

LIDI[1]-CeTAD[2]

Resumen

En este artículo se muestra cómo se obtiene *speedup* (*aceleración*) superlineal a partir de código secuencial no optimizado como resultado de la paralelización de las aplicaciones de cómputo intensivo en supercomputadoras paralelas de memoria compartida del tipo SGI Origin 2000 tal como Clementina 2. Se muestra vía experimentación la principal razón por la cual se combinan el cómputo paralelo con la optimización local *derivada* de la paralelización para obtener speedup superlineal. Además, se muestra cómo un algoritmo orientado a las computadoras paralelas de memoria distribuida se aprovecha en el entorno de las arquitecturas paralelas de memoria compartida, siempre y cuando el rendimiento de las operaciones de pasaje de mensajes utilizadas sea apropiado. En este sentido se explica también cómo el concepto de granularidad tiene gran influencia no solamente en la frecuencia o relación cómputo-comunicación entre tareas sino también en la optimización de código local que se ejecuta en cada procesador de una máquina paralela de memoria compartida.

Palabras Clave: Cómputo Paralelo, Rendimiento, Supercomputadoras de Memoria Compartida, Optimización de Código, Granularidad, Speedup.

1.- Introducción

Los problemas del ámbito científico-numérico se han resuelto tradicionalmente con una combinación de procesadores muy potentes más la posibilidad de realizar cómputo simultáneamente en más de un procesador a la vez. Con muy pocas excepciones que no han perdurado en el tiempo, el diseño de las máquinas paralelas ha intentado incluir como elementos de cálculo los procesadores con la mayor potencia disponible. Dentro de esta gama, se pueden identificar múltiples ejemplos de computadoras paralelas MIMD (Multiple Instruction, Multiple Data stream) con memoria compartida o distribuida.

[1] Laboratorio de Investigación y Desarrollo, 50 y 115, Fac. de Informática, UNLP, 1900, La Plata.
[2] Centro de Técnicas Analógico-Digitales, 48 y 116, Dto. De Electrotecnia, Fac. de Ingeniería, UNLP, 1900, La Plata.

Desde el ámbito de las aplicaciones de cómputo intensivo, tradicionalmente se ha intentado aprovechar al máximo todas las características de diseño de las computadoras paralelas en las cuales se calculan los resultados numéricos esperados. Esta optimización incluye desde el *mejor* algoritmo paralelo hasta la *mejor* forma de hacer los cálculos en cada uno de los procesadores disponibles en la máquina paralela. La cuantificación de esta tarea combinada de paralelización ha sido y es muy difícil, y por lo tanto también lo es la comparación de las alternativas de solución de un problema en particular. Por esta razón, hasta ahora se han impuesto los métodos de evaluación de las soluciones paralelas por medio de la experimentación sobre las propias computadoras paralelas y comparación directa del rendimiento obtenido por cada una de las alternativas propuestas.

Dentro de las aplicaciones de cómputo intensivo, el área de las aplicaciones de álgebra lineal ha sido y es de gran actividad de investigación dada la gran cantidad de problemas sobre las que se utilizan, y tal es así que la librería LAPACK (Linear Algebra PACKage) [1] constituye un estándar de facto. En esta librería se definen un conjunto bastante reducido de operaciones que son utilizadas en la mayoría (sino en *todas*) las aplicaciones de álgebra lineal. A los afectos de reconocer lo más rápidamente posible las operaciones con mayor incidencia en el rendimiento de un programa que resuelve completamente un problema, se identifican dentro de LAPACK las operaciones más básicas: BLAS (Basic Lineal Algebra Subroutines) en tres niveles de operadores. A su vez, se identifica el nivel con mayor requerimiento de cómputo como BLAS Level 3, donde todos los operadores son de la forma: $Mat_1 = Mat_2$ op Mat_3, es decir operaciones entre matrices que dan como resultado otra matriz. Se puede considerar a la operación de multiplicación de matrices como la más representativa de estas operaciones y por lo tanto es la elegida para la discusión y experimentación en este artículo.

2.- Multiplicación de Matrices en Paralelo

Una gran cantidad de algoritmos han sido propuestos y analizados para la multiplicación de matrices sobre diferentes computadoras paralelas [13]. Por razones de simplicidad, los algoritmos se describen normalmente en términos de C = A x B, donde A, B y C son matrices consideradas densas y cuadradas de orden *N*. En el contexto de cómputo paralelo, la multiplicación de matrices misma se considera inherentemente buena para su paralelización, dado el patrón de acceso a los datos de las matrices y su consecuente falta de dependencia en los cálculos. Para el cálculo de C = A x B, las matrices A y B se acceden sólo para lectura y la matriz C es la única sobre la que habría que coordinar el acceso de escritura en un ambiente de memoria compartida.

Dadas las características del cálculo de la multiplicación de matrices, la gran mayoría de los algoritmos siguen el modelo de cómputo paralelo SPMD (Single Program - Multiple Data), donde todos los procesadores ejecutan el mismo programa. Esto también simplifica el diseño mismo del algoritmo paralelo así como su depuración y muchas veces su (trans)portabilidad.

Algunos de los algoritmos paralelos propuestos, tales como el Divide-and-Conquer, el Recursivo y las paralelizaciones del método de Strassen [10] son considerados especialmente orientados a las computadoras paralelas de memoria compartida. En general, estos algoritmos también se han considerado limitados a las computadoras de memoria físicamente compartida dado que la simulación

o el manejo de memoria compartida en las computadoras paralelas de memoria distribuida es extremadamente costoso en términos de rendimiento.

Puede considerarse que la mayoría de los algoritmos paralelos de producto de matrices son orientados a máquinas con arquitectura MIMD de memoria distribuida y, más específicamente, con interconexión de procesadores en forma de grilla o toro [9] [8] [3] [6]. En particular, el algoritmo de Fox [8] es uno de los más representativos al respecto, con varias publicaciones posteriores que hacen su adaptación a arquitecturas paralelas determinadas [4] [11] [5]. A pesar de que no es específicamente orientado a máquinas paralelas de memoria compartida, es el que se utilizará en la experimentación y con el que se logra speedup superlineal. Para los algoritmos orientados a las arquitecturas paralelas con memoria distribuida, también es bastante usual hacer una descripción bastante minuciosa de la distribución de los datos con respecto a los procesadores.

3.- Algoritmo de Fox

Tal como se detalló anteriormente, este algoritmo está orientado a las máquinas paralelas de memoria distribuida y se puede explicar en términos del modelo de cómputo paralelo SPMD junto con la distribución de los datos de las matrices.

3.1.- Distribución de los Datos

Para realizar la multiplicación se utilizan p tareas o procesos ubicados sobre una grilla de $m \times m$ elementos, siendo $p = m^2$. Cada tarea calculará un bloque de $n \times n$ elementos de la matriz resultado. El tamaño de las matrices A, B y C es $(n \times m) \times (n \times m)$, es decir que se opera con matrices cuadradas de orden $N = (n \times m)$. Las matrices A, B y C son almacenadas como bloques distribuidos en las m^2 tareas. Cada tarea t_{ij} (donde i es la fila y j es la columna de la grilla de procesadores) contiene los bloques A_{ij}, B_{ij} y C_{ij}. La Fig. 1 muestra una matriz dividida en bloques y su distribución entre las distintas tareas. Esta distribución es igual para las tres matrices (A, B y C).

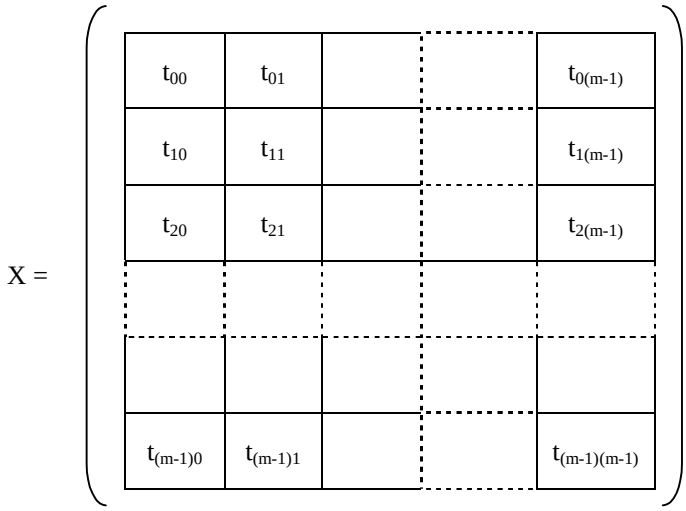

Figura 1: Distribución de los datos en procesos según el algoritmo de Fox.

3.2.- Descripción del Algoritmo

La Fig. 2 muestra en pseudo-código lo que ejecuta cada tarea t_{ij} para calcular su bloque correspondiente de la matriz resultado, siguiendo el modelo SPMD.

```
for k = 0 to m-1
{
  if (j == ((i + k) mod m))
      Broadcast A_ij a las tareas de la misma fila
  else
      Receive A_ip                    / * donde p = (i + k) mod m * /
  C_ij = A_ip * B_ij                  / * donde p = (i + k) mod m * /
  Send B_ij                           / * envío de B_ij a la tarea t_(i-1)j  * /
  Receive B_ij                        / * recepción del bloque de B de la tarea t_(i+1)j * /
}
```

Figura 2: Código que ejecuta cada tarea según el algoritmo de Fox.

En el caso de tener matrices de 12x12 elementos y una grilla de 3x3 procesos-tareas-procesadores, cada uno de éstos tiene bloques de 4x4 elementos, es decir que $p = 9$, $m = 3$, $n = 4$ y $N = n \times m = 12$. La Fig. 3 muestra la grilla de 3x3 tareas, donde cada tarea t_{ij} contiene los bloques A_{ij}, B_{ij} y C_{ij}, con $i, j = 0, 1, 2$.

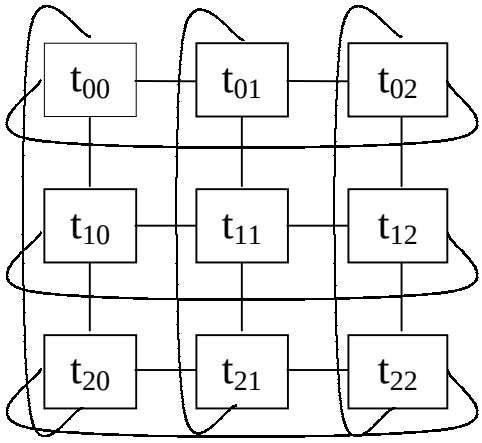

Figura 3: Grilla de 3x3 tareas para el algoritmo de Fox.

En lo que podría considerarse el "Primer Paso" del algoritmo de Fox, tal como se deriva del pseudo-código de la Fig. 2 es el envío broadcast por filas de cada bloque de la matriz A asignado a una tarea de la diagonal de la grilla hacia las demás tareas en la misma fila, tal como lo muestra la Fig. 4.

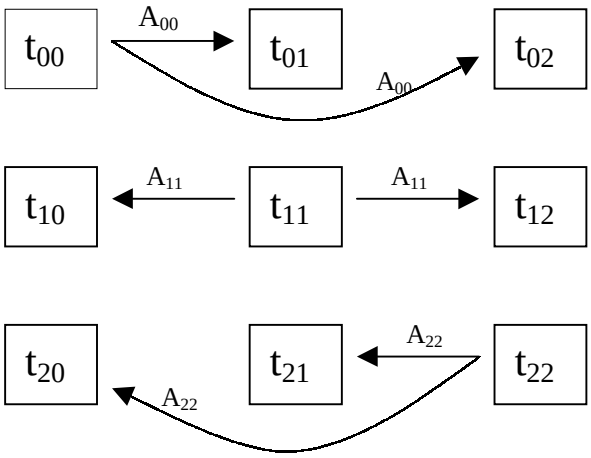

Figura 4: Primer broadcast por filas del algoritmo de Fox.

En lo que se podría considerar como el segundo paso, cada tarea t_{ij} realiza el primer cálculo intermedio de C_{ij}, es decir que cada tarea realiza la operación: $C_{ij} = C_{ij} + A_{ii} \times B_{ij}$. En el tercer paso, los bloques de la matriz B, B_{ij}, se rotan por columnas de la grilla de procesos tal como lo muestra la Fig. 5.

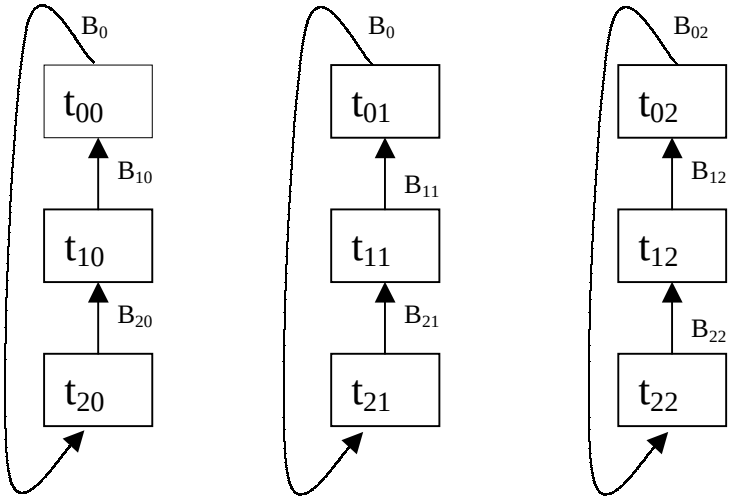

Figura 5: Rotación de los bloques de B en el algoritmo de Fox.

3.3.- Análisis del Algoritmo

Las principales carácterísticas del algoritmo de Fox son:

- En lo que se refiere a la arquitectura de cómputo, el algoritmo de Fox está claramente orientado a máquinas paralelas MIMD con memoria distribuida, dado básicamente por la distribución y posterior intercambio de los datos asignados a cada tarea-proceso-procesador.
- Por otro lado, el modelo de paralelización es el de pasaje de mensajes, donde cada proceso-tarea consiste en una secuencia de instrucciones entre las que se intercalan mensajes, o rutinas de comunicación de datos necesarios para la cooperación de las tareas que resuelven el problema.

- El modelo de ejecución claramente es SPMD, dado por la definición de un único código (a nivel de pseudo-código, para evitar detalles no necesarios para comprender el algoritmo) que todas las tareas ejecutan y con el cual se llega al resultado esperado.

4.- Experimentación

La experimentación se llevó a cabo en la computadora paralela conocida como Clementina 2, que es el modelo Origin 2000 de SGI, con las siguientes características:

- Elementos de procesamiento: procesadores MIPS R12000 de 300 MHz.
- Cantidad de procesadores: 40.
- Memoria compartida de 10 GB NUMA (Non Uniform Memory Access).
- Capacidad de disco: 360 GB.
- Sistema Operativo: IRIX 6.5, con capacidad de multiprocesamiento simétrico (SMP) de hasta 128 procesadores y soporte de 32 y 64 bits.

El código de multiplicación de matrices que se ejecuta en cada iteración del algoritmo es el más simple y directo, sin ningún tipo de optimización ni siquiera a nivel del compilador. Cada vez que se realiza el cálculo parcial del bloque de C, C_{ij}, en cada proceso se ejecutan las tres iteraciones anidadas clásicas para el cálculo de una multiplicación de matrices.

El manejo de procesos y pasaje de mensajes es realizado con la librería PVM (Parallel Virtual Machine) [7]. Tareas tales como creación, sincronización y eliminación de procesos, creación, envío, recepción y broadcast de mensajes es realizada con funciones de esta librería.

Las matrices son cuadradas, y su dimensión es de *alrededor* de 1700x1700 elementos (en los casos en que es posible). Si bien puede considerarse que el tamaño es relativamente pequeño con respecto a la memoria principal, los experimentos mostraron que tamaños mayores de matrices implican la utilización de espacio de memoria *swap*. La utilización de este espacio de memoria (*swap*) desvirtúa considerablemente los resultados en términos de rendimiento, dado que el tiempo de acceso tiene varios órdenes de magnitud de diferencia con respecto a la memoria principal y esto debe evitarse. La principal (y quizás la *única*) razón por la cual tamaños mayores implican utilización del espacio de *swap* es el acceso compartido que se tiene a la máquina paralela y por lo tanto la consiguiente distribución de la memoria entre *todos* los trabajos en ejecución al mismo tiempo de la experimentación con la multiplicación de matrices en paralelo.

Las distintas pruebas varían en la cantidad de procesos usados para realizar el producto ($p = m \times m$) y en el tamaño del bloque (n) calculado por cada uno (estos dos valores son parámetros del programa). Tal como lo define el algoritmo, las matrices son cuadradas y su dimensión es de $(m \times n) \times (m \times n)$. Los valores de m y n de las distintas pruebas fueron elegidos para que las matrices sean de 1700×1700 elementos. En los casos en que 1700 no es múltiplo de m, se multiplican matrices de dimensión igual al múltiplo de m menor más cercano a 1700.

Dado que se dispone de 40 procesadores, las pruebas se llevaron a cabo con 1, 4, 9, 16, 25 y 36 procesos. No interesa que más de un proceso se ejecute en un procesador porque esto implica la

serialización (ejecución secuencial) de la ejecución de dichos procesos. La Tabla 1 resume las características principales de la experimentación.

m	cantidad de procesos	n	tamaño de matrices
1	1	1700	1700
2	4	850	1700
3	9	566	1698
4	16	425	1700
5	25	340	1700
6	36	283	1698

Tabla 1: Características de la experimentación.

5.- Resultados Obtenidos

Los tiempos de ejecución obtenidos en las pruebas mencionadas en la sección anterior se ilustran en la Fig. 6. Estos tiempos son la suma de los tiempos de *user*, *system* y *pvmd*. Con estos datos se incluye el tiempo de procesamiento de cada proceso, más el tiempo que emplea PVM (las funciones de librería más el propio proceso de manejo de mensajes *pvmd*), para crear los procesos y para realizar el pasaje de mensajes.

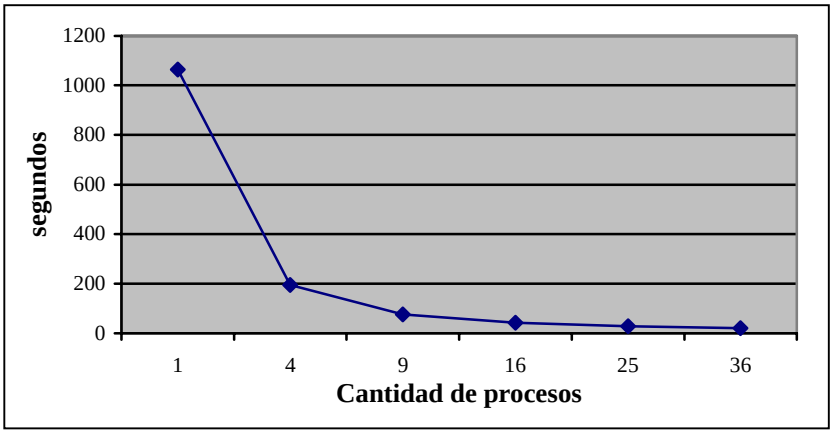

Figura 6: Tiempos de ejecución del algoritmo de Fox.

La gran diferencia de tiempo entre la ejecución con 1 solo proceso (1063.93 segundos) y el resto de las pruebas (menor a 200 segundos), más las diferencias relativas entre las cantidades de procesos utilizados (los puntos en el eje **x** se muestran a igual distancia entre sí) en las distintas pruebas hace que no se pueda apreciar con precisión la diferencia entre los tiempos de ejecución de las pruebas con más de 1 proceso.

Teniendo en cuenta la definición del índice de rendimiento conocido como speedup (o aceleración),

$$\textbf{\textit{Speedup}} = \frac{\text{tiempo secuencial}}{\text{tiempo paralelo}}$$

se pueden graficar los valores de speedup obtenidos para cada cantidad de procesadores utilizados. Como es conocido, el valor máximo del speedup en teoría es igual a la cantidad de procesadores, razón por la cual los resultados suelen compararse con la recta y = x. Como lo muestra la Fig. 7, el speedup obtenido es superior al máximo teórico, y los casos en los que esto ocurre suelen denominarse como speedup superlineal.

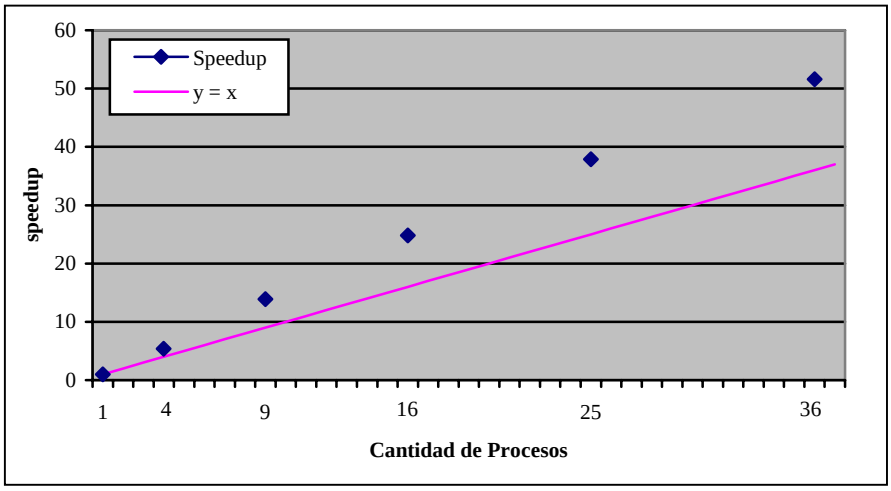

Figura 7: Valores de speedup del algoritmo de Fox.

Dado que el speedup superlineal puede considerarse casi como una anomalía, deben analizarse con mucho detalle las razones por las cuales ocurre.

6.- Análisis de la Obtención de Speedup Superlineal

A priori, el algoritmo de Fox sobre una arquitectura paralela tal como la utilizada no solamente no debería generar speedup superlineal sino que debería tender a cierta degradación por el traslado, en principio innecesario, de los datos desde un proceso hacia los demás. En este sentido, el algoritmo de Fox tiene varias comunicaciones punto a punto (todas las comunicaciones por columnas de los bloques de la matriz B) y también comunicaciones colectivas (todos los broadcasts por filas de los bloques de la matriz A). A su vez, dado que la librería PVM es la encargada de las comunicaciones, todos los mensajes que potencialmente se podrían resolver simultáneamente en una grilla de procesadores deben pasar por el proceso que maneja las transferencias de datos (*pvmd*) con la consiguiente serialización.

Por un lado, es evidente que PVM resuelve en forma muy satisfactoria (en términos de rendimiento) todas las transferencias de datos entre los procesos, ya que el tiempo de ejecución no se ve afectado por las tranferencias de datos entre los procesos. Para el algoritmo de Fox, estas transferencias de datos significan: una comunicación colectiva (broadcast por fila de los bloques de A) y una comunicación punto a punto (bloques de B) por cada cálculo intermedio de bloques de C.

Por otro lado, algo más debe suceder a nivel de cálculo para que la misma cantidad de operaciones que para la resolución con cómputo secuencial se resuelvan más rapidamente, y tal como lo muestra la

Fig. 7, se resuelvan aún más rápido a medida que se utilizan mayor cantidad de procesos. La respuesta a esta interrogante se encuentra en la localidad de referencia que tienen los procesos de la aplicación paralela a medida que la cantidad de procesos es mayor y en la optimización que eso implica para el cómputo en un procesador.

Manteniendo la cantidad de datos constante (dimensión de las matrices en este caso), al aumentar la cantidad de procesos-procesadores involucrados la cantidad de datos a los que tiene acceso cada procesador disminuye proporcionalmente. Por ejemplo: cuando se utilizan cuatro procesadores (grilla de 2x2 para el algoritmo de Fox) cada uno de ellos tiene 1/4 de datos con respecto a la aplicación secuencial. Esto necesariamente implica aumentar cuatro veces la localidad de referencia. ¿Qué relación tiene esto con la velocidad de procesamiento? Mucha, dado que todos los procesadores actualmente tienen integrados como mínimo dos niveles de memoria cache y por lo tanto aumentar la localidad de referencia implica aumentar la velocidad de acceso a los datos dado que se hace más probable encontrarlos en algún nivel de memoria cache en vez de tener que acceder *siempre* a memoria principal. Aumentando la localidad de referencia el acceso a memoria se resuelve más rápidamente, lo que implica que los cálculos en los cuales se utilizan esos datos se realizan más rápidamente y por lo tanto es como si se tuvieran procesadores con mayor velocidad que la que se obtiene al procesar todos los datos secuencialmente en un procesador.

La idea de aumentar la localidad de los datos a los que se accede para aumentar la velocidad de procesamiento por accesos a memoria cache no es nueva en el contexto del procesamiento secuencial [2] [9] [12]. Sin embargo, los resultados de todos los esfuerzos para mejorar la localidad de referencia de los cálculos secuenciales (a los que se denomina *procesamiento por bloques*) se tienen como efecto *colateral* de la paralelización de las aplicaciones. Es interesante notar que la paralelización por memoria compartida no necesariamente determina *per se* aumentar la localidad de referencia, a diferencia de la paralelización con procesos que se comunican y que por lo tanto tienen áreas de datos totalmente disjuntas.

7.- Conclusiones y Trabajo Futuro

Las dos conclusiones más importantes a partir de lo que se ha presentado son:

- La paralelización sobre arquitecturas con memoria compartida no necesariamente impide la utilización de los algoritmos orientados a máquinas paralelas con memoria distribuida. En el presente artículo se utilizó el algoritmo de Fox para paralelizar la multiplicación de matrices sobre una SGI Origin 2000.
- La paralelización utilizando código no optimizado puede dar como resultado la obtención de speedup superlineal en función de lo que significa un “sistema” paralelo (algoritmo más arquitectura de hardware). En el caso de la multiplicación de matrices quizás no sea tan complicado (costoso) utilizar código optimizado para las distintas arquitecturas de los procesadores/máquinas paralelas, pero quizás sí lo sea para otras aplicaciones del área de cálculo intensivo.

Existen varias posibilidades de estudio y análisis a partir del presente artículo y de los resultados que se reportan:

- Una de las primeras extensiones al presente trabajo debería ser la identificación automática de la

mejor cantidad de procesos (en términos de localidad de referencia y velocidad de procesamiento) dado un tamaño de problema.

- Utilización de código optimizado para la identificación de la ganancia obtenida por el aumento de la localidad de referencia (procesamiento por bloques) con relación a las demás optimizaciones posibles, tales como el aprovechamiento de las características superescalares de los procesadores actuales.
- Extensión del presente trabajo a todas las operaciones definidas en el nivel 3 de BLAS y en LAPACK.

Bibliografía

[1] Anderson E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK: A Portable Linear Algebra Library for High-Performance Computers, Proceedings of Supercomputing '90, pages 1-10, IEEE Press, 1990.

[2] Bilmes J., K. Asanovic, C. Chin, J. Demmel, Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology, Proc. Int. Conf. on Supercomputing, Vienna, Austria, July 1997, ACM SIGARC.

[3] Cannon L. E., A Cellular Computer to Implement the Kalman Filter Algorithm, Ph.D. Thesis, Montana State University, Bozman, Montana, 1969.

[4] Choi J., J. Dongarra, D. Walker, PUMMA: Parallel Universal Matrix Multiplication Algorithm on Distributed Memory Concurrent Computers, in Concurrency: Practice and Experience, 6:543-570, 1994.

[5] Choi J., “A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers”, Proc. of the High-Perf. Comp. on the Information Superhighway, IEEE, HPC-Asia '97.

[6] Dekel E., D. Nassimi, S. Sahni, Parallel matrix and graph algorithms, SIAM Journal on Computing, 10:657-673, 1981.

[7] Dongarra J., A. Geist, R. Manchek, V. Sunderam, Integrated pvm framework supports heterogeneous network computing, Computers in Physics, (7)2, pp. 166-175, April 1993.

[8] Fox G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Vol. I, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[9] Golub G. H., C. F. Van Loan, Matrix Computation, Second Edition, The John Hopkins University Press, Baltimore, Maryland, 1989.

[10] Strassen V., Gaussian Elimination Is Not Optimal, Numerische Mathematik, Vol. 13, 1969.

[11] van de Geijn R., J. Watts, SUMMA Scalable Universal Matrix Multiplication Algorithm, LAPACK Working Note 99, Technical Report CS-95-286, University of Tenesse, 1995.

[12] Whaley R., J. Dongarra, Automatically Tuned Linear Algebra Software, Proceedings of the SC98 Conference, Orlando, FL, IEEE Publications, November, 1998.

[13] Wilkinson B., Allen M., Parallel Programming: Techniques and Applications Using Networking Workstations, Prentice-Hall, Inc., 1999.