Análisis del impacto de distintas técnicas de optimización de rendimiento en multicore

Matías Dell'Oso[1], Juan Manuel Paniego[1], Martín Pi Puig[1], Marcelo Naiouf[1], Armando De Giusti[1,2]

[1] Instituto de Investigación en Informática LIDI (III-LIDI) – Facultad de Informática –UNLP
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Argentina
{mdelloso, jmpaniego, mpipuig, mnaiouf, degiusti}@lidi.info.unlp.edu.ar

Abstract. Este trabajo se enfoca en la comparación de distintas técnicas para reducir el tiempo de ejecución de un algoritmo. En primer lugar se analizan dos APIs para la programación multiproceso de memoria compartida, Pthreads y OpenMP realizando una breve comparación entre ellas y resaltando ventajas y desventajas de cada una. Luego, se demuestra la importancia del buen uso de la memoria caché y cómo impacta en la performance de programas tanto paralelos como secuenciales. Por último, se utilizan distintas optimizaciones brindadas por el compilador para aumentar la performance de los algoritmos. Al finalizar todas las pruebas se presenta una comparación entre las tres técnicas estudiadas, resaltando los escenarios en los cuales cada una de ellas presenta mejores resultados. El caso de estudio elegido es el problema clásico de multiplicación de matrices, utilizado para demostrar el impacto de la utilización óptima de la jerarquía de memoria existente en una arquitectura paralela.

Keywords*:* Arquitecturas paralelas, Multicore, Jerarquía de memoria, Pthreads, OpenMP.

1 Introducción

Tradicionalmente, la simulación numérica de sistemas complejos como por ejemplo dinámica de fluidos, clima, circuitos electrónicos, reacciones químicas, modelos ambientales, ha impulsado el desarrollo de computadores cada vez más potentes. Inicialmente, esto se alcanzaba aumentando la velocidad de los procesadores (según la conocida Ley de Moore [5]), incrementando los puntos de acceso a los datos (lo que disminuye los tiempos de disponibilidad de los mismos) e implementando algoritmos y técnicas optimizadas para resolver las tareas eficientemente.

Estas técnicas alcanzaron pronto sus límites, ya que los procesadores disipaban cada vez más calor generando fallas y eran más costosos en relación a sus prestaciones. Además, ya no era posible seguir optimizando los algoritmos para obtener mejores resultados en cuanto a tiempo. Para dar solución a estos problemas surgen los multiprocesadores, y con ellos la programación paralela. Actualmente, las arquitecturas dominantes son los *multicore*, donde en un mismo chip se incluyen varios núcleos de procesamiento [6][7].

El hecho de dividir el trabajo entre 2 o más procesadores permite bajar sus frecuencia de reloj (reduciendo sus costos) y obtener resultados superiores a los obtenidos utilizando un único procesador. Sin embargo, a pesar del buen rendimiento de los multiprocesadores, el desarrollo de hardware y software paralelo ha sido tradicionalmente dificultoso.

En ese sentido, surgieron nuevas problemáticas como la coherencia de los datos y el acceso compartido a los recursos. Para hacer frente a esto, se desarrollaron APIs y metodologías para asegurar la confiabilidad de las aplicaciones paralelas.

En el caso de los multicore, debe tenerse en cuenta la jerarquía de memorias involucradas, generalmente con diferentes niveles de caché, para obtener buenas prestaciones. En este trabajo se realiza una comparación entre la API Pthreads y la API OpenMP, resaltando las ventajas y desventajas de cada una de ellas, y se mostrará la importancia del buen uso de la memoria caché y cómo impacta en la performance de programas tanto paralelos como secuenciales.

Como caso de estudio se utiliza la multiplicación de matrices, un problema conocido y que suele ser parte de soluciones más generales. Para ello se realizan una serie de multiplicaciones de matrices utilizando ambas librerías y se comparará tanto el speedup alcanzado por cada una de ellas respecto del algoritmo secuencial, como también la simplicidad de cada código. Luego, se implementan distintas técnicas de optimización de la memoria caché, para demostrar que en muchos casos su buen uso puede arrojar mejores resultados que paralelizar la aplicación. Por último, se utilizaron comandos especiales del compilador GCC para compilar más "agresivamente" y así obtener mejores resultados.

El trabajo está organizado de la siguiente manera: la Sección 2 presenta el caso de estudio y el equipo utilizado. A continuación, en las Secciones 3, 4 y 5 se presentan los resultados obtenidos para las tres optimizaciones propuestas, así como los basamentos teóricos de cada una de ellas. El análisis de los resultados obtenidos así como la comparación entre las tres técnicas utilizadas se muestra en la Sección 6. Por último, en la Sección 7, se exponen las conclusiones.

2 Caso de estudio y equipo utilizado

Se eligió la multiplicación de matrices como caso de estudio, ya que representa un problema ampliamente tratado y que en muchos casos es parte de soluciones a problemas de mayor envergadura. Dadas dos matrices A de m x p y B de p x n elementos, la multiplicación de ambas consiste en obtener la matriz C de m x n elementos ($C = A$ x B), donde cada elemento se calcula por medio de la Ecuación 1, tal como muestra la Figura 3.

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} * B_{k,j} \quad (1)$$

Para encontrar el producto de las matrices existen diferentes algoritmos, entre otros el que utiliza 3 iteraciones y la solución por bloques [8][9].

En este trabajo la experimentación se realizó sobre un equipo con procesador Intel Xeon [13] y 8 GB de memoria RAM. Todos los algoritmos fueron compilados con el GNU Compiler Collection (GCC) corriendo sobre Fedora release 14 (Laughlin). Las especificaciones del equipo son las siguientes: Procesador: Intel Xeon X3430; Cores: 4; Threads: 4; Frecuencia de reloj: 2400 MHz; Intel Turbo Boost: 2800 MHz (deshabilitado); Caché de datos L1: 32 KB por core; Caché L2: 256 KB por core; Caché L3: 8 MB compartida.

3 Pthreads vs OpenMP

Históricamente, los vendedores de hardware implementaban sus propias versiones de threads. Estas implementaciones diferían completamente unas de otras, haciendo dificultosa la programación de aplicaciones portables. Para hacer frente a este problema y poder explotar todo el potencial de los threads, fue necesario crear un estándar. Para los sistemas UNIX, la IEEE especificó el estándar 1003.1c-1995, también conocido como Pthreads [11], el cual es soportado por la mayoría de los vendedores de hardware [1]. Si bien la API Pthreads ha ido evolucionando, su uso continúa siendo restringido a un sector acotado de programadores. Esto se debe a que su modelo de programación se basa en primitivas de bajo nivel y puede resultar complejo para los programadores afines a utilizar lenguajes de alto nivel.

Por ello surgió OpenMP [10] (Open Multi-Processing), una API que provee un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas. A su vez, las directivas de OpenMP proveen soporte para concurrencia, sincronización y manejo de datos, obviando el uso explícito de semáforos, variables condición, alcance de los datos e inicialización de threads [2].

Para realizar la comparación entre las dos APIs, se utiliza el tiempo que tardan en ejecutarse tres algoritmos de multiplicación de matrices cuadradas (dimensión *N*x*N*) y que almacenan datos en punto flotante de doble precisión. La primera multiplicación se realiza utilizando un algoritmo secuencial, la segunda utilizando Pthreads y la última utilizando OpenMP, variando la dimensión de las matrices. En la Tabla 1 se pueden observar los resultados obtenidos.

Tabla 1. Tiempos obtenidos para cada algoritmo (en segundos).

N	Secuencial	Pthread (4 hilos)			OMP (4 hilos)		
	Tiempo	Tiempo	Speedup	Eficiencia	Tiempo	Speedup	Eficiencia
512	1,6643	0,4732	3,5171	0,8792	0,6394	2,6029	0,6507
1024	24,3859	6,2967	3,8727	0,9681	8,0689	3,0221	0,7555
2048	284,3292	81,2401	3,4998	0,8749	112,0459	2,5376	0,6344
4096	2524,209	693,5912	3,6393	0,9098	934,7879	2,7003	0,6751
8192	21854,622	5941,7058	3,6781	0,9195	8379,496	2,6081	0,6525
16384	205989,06	54546,853	3,7763	0,944	71807,286	2,8686	0,7171

Como se observa en la tabla anterior, al utilizar la API Pthreads se obtiene un mayor speedup y eficiencia que usando OpenMP. Esto se debe a que con Pthreads se tiene un mejor control sobre las tareas a realizar, es decir, el programador se encarga de la concurrencia y de la creación de los threads. Por otra parte, programar aplicaciones paralelas utilizando OpenMP es muy sencillo, por lo que simplifica de manera notable la creación de dichas aplicaciones.

Vale la pena destacar que si la programación se realiza de manera correcta, resulta muy difícil obtener un mayor speedup utilizando OpenMP que mediante el uso de Pthreads. De todas maneras, si el programador no está acostumbrado a trabajar con primitivas de bajo nivel o si el programa a resolver es muy complejo, es probable que se obtengan mejores resultados utilizando OpenMP. Es por esto que resulta una herramienta sumamente útil para programadores menos experimentados en programación paralela.

4 Optimización de la memoria caché

Los algoritmos con los que se realizaron los cálculos anteriores no presentan optimización alguna. Ambas matrices A y B se inicializan por filas y luego se realiza la multiplicación accediendo a cada una de ellas del mismo modo. Al inicializar las matrices A y B de la misma forma, a la hora de multiplicarlas, los fallos de caché son muchos, lo que provoca un incremento considerable en el tiempo de ejecución del algoritmo. Como se observa en la Figura 1, para calcular el valor C[1,1] se necesitan todos los valores de la fila 1 de la matriz A y todos los valores de la columna 1 de la matriz B.

Cuando un dato es copiado desde la memoria RAM hacia la memoria caché, también son copiados los datos que están contiguos en dicha memoria (por principio de localidad espacial [1]). Al estar cargadas ambas matrices por filas, una porción de una fila de cada matriz es copiada a la memoria caché para su rápida utilización. Para la matriz A, esto no presenta problemas, ya que toda la fila es necesaria para realizar el cálculo. La matriz B, por otro lado, al necesitar una columna entera, produce un fallo de caché en cada iteración de la multiplicación (si consideramos matrices de dimensiones grandes). Esto genera que todos los datos de B deban ser buscados en la memoria principal, lo que agrega un overhead de comunicaciones sumamente alto.

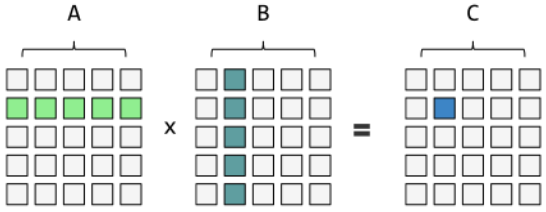

Fig.1. Multiplicación de matrices.

Para hacer frente a este problema se proponen una serie de modificaciones a los algoritmos buscando optimizar el acceso a los datos:

a) Optimizar la carga

Como se mencionó anteriormente, la matriz B presenta problemas debido a que está inicializada por filas. Al modificar el algoritmo para que B sea inicializada por columnas, se reducirán significativamente los fallos de caché (ver Tabla 2). De esta manera, una porción de la fila de la matriz A y una porción de la columna de la matriz B son cargadas en caché, lo que reduce notablemente el acceso a la memoria principal [4].

Tabla 2. Tiempos obtenidos para cada algoritmo optimizado (en segundos).

N	Secuencial	Pthread (4 hilos)			OMP (4 hilos)		
	Tiempo	Tiempo	Speedup	Eficiencia	Tiempo	Speedup	Eficiencia
512	1,1375	0,3166	3,5921	0,898	0,4266	2,6658	0,6665
1024	9,1716	2,5337	3,6197	0,9049	3,4052	2,6933	0,6733
2048	74,0484	20,459	3,6193	0,9048	27,4889	2,6937	0,6734
4096	594,9148	164,6297	3,6136	0,9034	221,2108	2,6893	0,6723
8192	4759,606	1324,293	3,59407	0,8985	1767,229	2,6932	0,6733
16384	38061,901	10626,83	3,58167	0,8954	14146,057	2,6906	0,6726

Se observa que la relación entre los tiempos obtenidos con los tres algoritmos (secuencial, Pthreads y OMP) se asemeja a la obtenida sin la optimización, pero ahora todos los tiempos se redujeron notablemente.

b) Optimización de caché L1 - Multiplicación por bloques

Para optimizar aún más el acceso a los datos, se utiliza un algoritmo de multiplicación por bloques [3]. La idea es dividir las matrices A y B en sub-matrices más pequeñas que quepan en la caché L1 con el fin de realizar la multiplicación con un número menor de fallos de caché. La clave del éxito de este algoritmo es elegir el tamaño óptimo del bloque el cual puede ser calculado según la Ecuación 2:

$$2 * (tamañoDeBloque)^2 * tamañoPalabra = tamañoCacheL1 \quad (2)$$

En este caso, *tamañoPalabra* es 8 bytes debido a que las matrices son del tipo double. Nótese que se multiplica por 2, debido a que es necesario que quepan 2 bloques. Sustituyendo los valores, se calcula que el tamaño de bloque óptimo es de 45. Sin embargo, se encontró que el tamaño que arroja mejores resultados (utilizando potencias de 2) es 32. Esto puede deberse a que las pruebas no se realizaron sobre un sistema aislado por lo que otro proceso en segundo plano pudo estar ocupando parte de la memoria caché. En la Tabla 3 se muestra los resultados obtenidos.

Tabla 3. Tiempos obtenidos para cada algoritmo optimizado por bloques (en segundos).

N	Secuencial	Pthread (4 hilos)			OMP (4 hilos)		
	Tiempo	Tiempo	Speedup	Eficiencia	Tiempo	Speedup	Eficiencia
512	1,0817	0,2987	3,6208	0,9052	0,3930	2,7519	0,6879
1024	8,6606	2,3916	3,6211	0,9052	3,1478	2,7512	0,6878
2048	69,2909	19,1524	3,6178	0,9044	25,1873	2,7510	0,6877
4096	554,0851	153,1872	3,6170	0,9042	201,6417	2,7478	0,6869
8192	4433,513	1228,393	3,6091	0,9022	1612,661	2,7491	0,6872
16384	35482,77	9837,132	3,6070	0,9017	12903,712	2,7498	0,6874

Si bien la mejora no es sustancial respecto de la anterior optimización, utilizando el algoritmo por bloques se consigue disminuir nuevamente el tiempo de ejecución de las tareas. Esto se debe a que los dos bloques a multiplicar entran por completo en la memoria caché L1, lo que reduce aún más los fallos de dicha memoria.

5 Optimización del compilador

GCC provee varias opciones de compilación para mejorar el rendimiento de una aplicación. A continuación se observan las opciones de compilación utilizadas para realizar las pruebas:

Secuencial	gcc –O3 -o nombreEjecutable nombreArchivo.c
Pthreads	gcc –O3 -o nombreEjecutable nombreArchivo.c -lpthread
OpenMP	gcc –O3 -o nombreEjecutable nombreArchivo.c -fopenmp

No es el objetivo de este trabajo entrar en detalle sobre los comandos existentes y su funcionamiento. Es posible visualizar una lista detallada de todas las opciones de compilación en la página web de GCC [12] o utilizando el comando “man” en sistemas Unix.

La optimización se realiza sobre el algoritmo de multiplicación por bloques. En la Tabla 4 se encuentran los resultados obtenidos.

Tabla 4. Algoritmo optimizado por compilador (tiempos en segundos).

N	Secuencial	Pthreads (4 hilos)			OMP (4 hilos)		
	Tiempo	Tiempo	Speedup	Eficiencia	Tiempo	Speedup	Eficiencia
512	0,1564	0,0504	3,1030	0,7757	0,0453	3,4509	0,8627
1024	1,2442	0,3940	3,1575	0,7893	0,3622	3,4345	0,8586
2048	9,9620	3,1570	3,1554	0,7888	2,8960	3,4398	0,8599
4096	79,6851	25,3469	3,1437	0,7859	23,2905	3,4213	0,8553
8192	638,314	205,845	3,1009	0,7752	187,6127	3,4022	0,8505
16384	5124,94	1664,209	3,0795	0,7698	1526,994	3,3562	0,8390

En la tabla 4 se puede observar una notable reducción de los tiempos de ejecución de todos los algoritmos. Cabe destacar que en esta ocasión la API OpenMP arrojó mejores resultados que la API Pthreads. Dado que el programador desconoce el funcionamiento interno de cada librería, no es posible predecir su comportamiento al combinarlas con optimizaciones del compilador. Es posible que utilizando otros compiladores los resultados varíen.

6 Análisis de resultados obtenidos

A lo largo de este trabajo se implementaron distintas técnicas para mejorar el tiempo de ejecución de una aplicación. Cada una de ellas aportó una mejora sobre la anterior llegando a obtener resultados muy interesantes. En la Tabla 5 se puede observar el speedup obtenido respecto del algoritmo secuencial sin optimizar, utilizando los tres tipos de optimizaciones vistos: paralelismo, optimización de caché y optimización del compilador.

Tabla 5. Tiempos obtenidos utilizando todas las optimizaciones (en segundos)

N	Algoritmo sin optimizar	Algoritmo optimizado	
	Tiempo	Tiempo	Speedup
512	1,6643	0,0453	36,72143
1024	24,3853	0,3622	67,3110
2048	284,3292	2,8960	98,1784
4096	2524,2094	23,2905	108,3797
8192	21854,6228	187,6127	116,4879
16384	205989,063	1526,9949	134,8983

Para la optimización por paralelismo se utilizó la API OpenMP (4 hilos) y para la optimización de acceso a los datos se utilizó el algoritmo de multiplicación por bloques. La Figura 2 muestra el aporte de cada una de las optimizaciones para las seis dimensiones de matrices analizadas.

En los gráficos se puede observar que, a medida que crece la dimensión de las matrices, la optimización de acceso a los datos juega un papel cada vez más relevante. Esto indica la importancia de hacer un buen uso de la memoria caché al manejar grandes volúmenes de información y de cómo impactan los fallos de dicha memoria en diferentes aplicaciones.

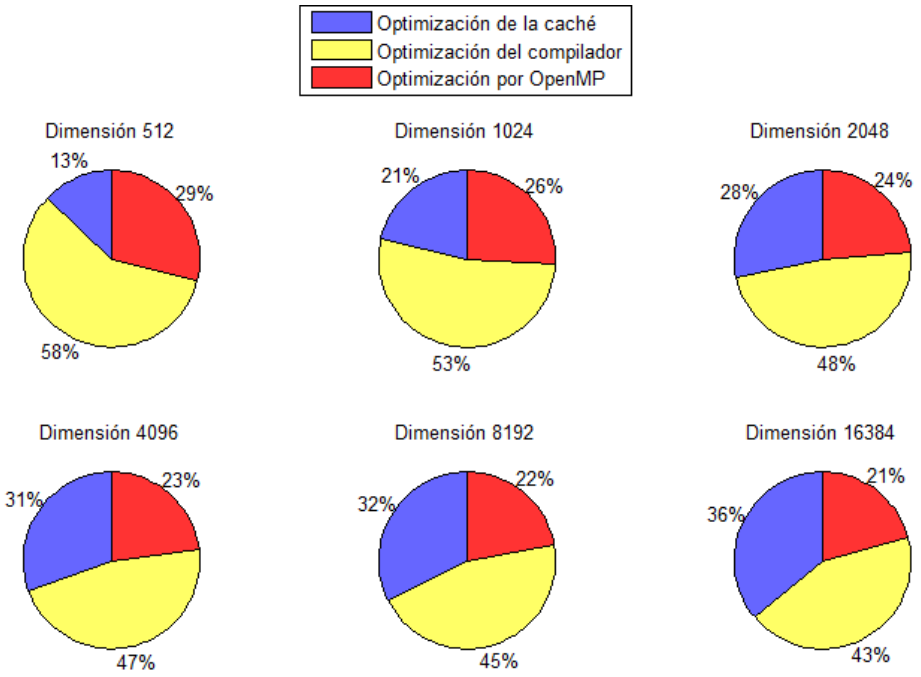

Fig. 2. Aporte de cada optimización.

A su vez, se observa que la optimización del compilador aporta una gran parte del speedup final obtenido, con la ventaja de que no implica esfuerzo adicional por parte del programador. Si bien el aporte de la librería OpenMP es el menor de los tres, no siempre es posible realizar optimizaciones por software sobre el algoritmo, ya sea por su complejidad o porque ya se alcanzó el nivel máximo de optimización. En estas situaciones, la programación multicore juega un papel fundamental a la hora de aumentar la performance de un algoritmo. Como en todas las aplicaciones la clave es saber elegir la técnica que mejor se aplique al problema y que otorgue el mayor beneficio al menor costo.

7 Conclusiones

A lo largo del trabajo se analizaron distintas técnicas utilizadas para optimizar una aplicación obteniendo resultados muy satisfactorios. Se demostró que aumentar el hardware paralelo no es la única opción para reducir su tiempo de ejecución. Mediante el uso de técnicas como reescribir el algoritmo para optimizar el acceso a los datos o compilar la aplicación utilizando comandos especiales, se pueden obtener mejoras que se asemejen a las que se obtendrían incrementando el número de procesadores o incluso mayores.

De todas maneras, si el algoritmo es muy complejo o simplemente ya no es posible optimizarlo por software, es aquí donde las técnicas de paralelización entran en juego.

Para ello puede utilizarse la API Pthreads, si se quiere el máximo speedup posible en detrimento de agregar complejidad al código, u OpenMP, si lo que se busca es disminuir el tiempo de desarrollo a expensas de un menor control sobre las tareas a realizar.

8 Bibliografía

1. Grama A., Gupta A., Karpyis G., Kumar V. "Introduction to Parallel Computing". Pearson – Addison Wesley 2003. ISBN: 0201648652. Segunda Edición (Capítulo 3 y 7).
2. Andrews G., Wesley A. "Foundations of Multithreaded, Parallel, and Distributed Programming". Disponible en Internet en: www.cs.arizona.edu/people/greg/mpdbook
3. Parello D., Temam O., Verdun JM. "On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance – Matrix-Multiply revisited", Supercomputing, 2002.
4. Lam M., Rothberg E., Wolf M. "The Cache Performance and Optimizations of Blocked Algorithms", ASPLOS IV, 1991.
5. Hennessy J., Patterson D. "Computer Architecture: A Quantitaive Approach". Quinta Edición (Capítulo 1).
6. Chapman B., "The Multicore Programming Challenge, Advanced Parallel Processing Technologies"; 7th International Symposium, (7th APPT'07), Lecture Notes in Computer Science (LNCS), Vol. 4847, p. 3, Springer-Verlag (New York), November 2007.
7. Kumar V., Gupta A., "Analyzing Scalability of Parallel Algorithms and Architectures". Journal of Parallel and Distributed Computing. Vol 22, nro 1.Pags 60-79. 1994.
8. Pardines I. "Técnicas paralelas aplicadas a optimización no lineal en sistemas de memoria distribuída".
9. Guerequeta R., Vallecillo A. "Técnicas de Diseño de Algoritmos". Servicio de Publicaciones de la Universidad de Málaga. Segunda Edición (Capítulo 3).
10. OpenMP Tutorial. Disponible en Internet en: https://computing.llnl.gov/tutorials/openMP/
11. POSIX Threads Programming Tutorial. Disponible en Internet en: https://computing.llnl.gov/tutorials/pthreads/
12. GCC online documentation. Disponible en Internet en: https://gcc.gnu.org/onlinedocs/gcc-5.1.0/gcc/
13. Intel Xeon X3430. Disponible en Internet en: http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20X3430%20-%20BV80605001914AG%20(BX80605X3430).html