Evolución del diseño de soluciones paralelas sobre clústers de multicore a fin de maximizar la performance. Aprovechamiento de la jerarquía de memoria.

Tesista: Lic. Fabiana Yael Leibovich
Director: Dr. Ricardo Marcelo Naiouf
Co-Director: Dra. Laura C. De Giusti

Trabajo Final presentado para obtener el grado de Especialista en Cómputo de Altas Prestaciones y Tecnología Grid

Facultad de Informática - Universidad Nacional de La Plata

Septiembre de 2013

Índice

1. Introducción

Dentro de las evoluciones recientes en las arquitecturas paralelas se encuentran los *clusters* de *multicore* [Cha12]. El mismo consiste en un conjunto de procesadores de múltiples núcleos interconectados mediante una red, en la que trabajan cooperativamente como un único recurso de cómputo. Es similar a un *cluster* tradicional pero con la diferencia de que cada nodo posee al menos un procesador *multicore* en lugar de un monoprocesador. Esto permite combinar las características más distintivas de ambas arquitecturas (memoria distribuida y compartida), dando origen a los sistemas híbridos [Don2002].

Al diseñar un algoritmo paralelo es muy importante considerar la jerarquía de memoria con la que se cuenta, ya que es uno de los factores que incidirá directamente en la performance alcanzable del mismo. Los *clusters* de *multicore* introducen un nivel más en la jerarquía de memoria si se lo compara con los *multicore*: la memoria distribuida accesible vía red; que permite la interconexión de los diferentes procesadores que conforman el *cluster*. Si se enumera la jerarquía de memoria para *multicores* del tipo Intel Xeón e5405, la misma queda conformada de la siguiente manera: niveles de registros y *cache* L1 propio de cada núcleo, *cache* compartida de a pares de núcleos (L2), memoria compartida entre los *cores* de un procesador *multicore* y finalmente memoria distribuida vía red [Bur2010], tal como puede verse en la Figura 1.

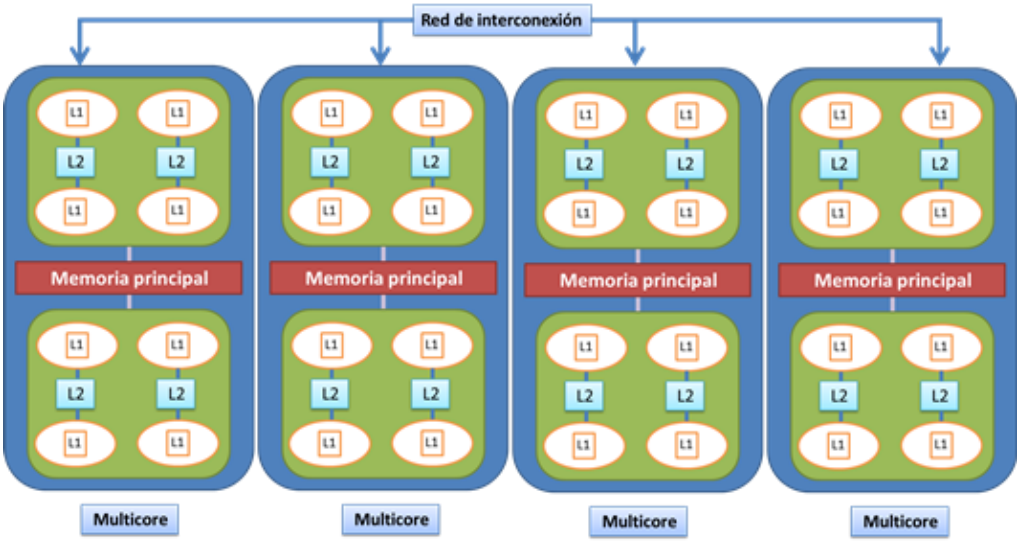

Fig. 1. *Cluster* de *multicore*

Es necesario aclarar que esta jerarquía puede ampliarse según los modelos de procesadores utilizados. Actualmente es frecuente encontrar procesadores *multicore* que utilizan un tercer nivel de *cache* (L3) [Int2012].

En este trabajo se ha seleccionado como caso de estudio la multiplicación de matrices, una de las aplicaciones más tradicionales y estudiada en el cómputo paralelo (dado que en muchos casos constituye la base de otras). Los motivos principales por los cuales se utiliza esta aplicación (ampliamente evaluada por los autores [Lei2011a] [Lei2012b] y los profesionales especializados en el área de interés de este trabajo [Cha2007] [Bis2008]) son que la misma permite utilizar y explotar el paralelismo de datos, así como la facilidad que presenta para escalar el problema al aumentar el tamaño de las matrices [And2000]. De esta forma, pueden compararse soluciones que utilizan diferentes paradigmas de programación paralela, por ejemplo, pasaje de mensajes e híbridas que aprovechan de manera diferente la jerarquía de memoria disponible en la arquitectura utilizada en la experimentación [Lei2011b].

2. Objetivos

El objetivo principal de este trabajo es estudiar la mejora en la performance de las soluciones paralelas por el aprovechamiento de la jerarquía de memoria, a partir de un análisis evolutivo de tres soluciones que resuelven el caso de estudio planteado utilizando diferentes paradigmas de resolución de problemas, así como diferentes grados de acoplamiento con la arquitectura. Partiendo de una solución típica al problema planteado, se llega a implementaciones más específicas desde el punto de vista de la arquitectura de prueba, particularmente de la jerarquía de memoria subyacente.

El análisis mencionado se realiza en base al tiempo de ejecución y eficiencia de las soluciones al escalar el tamaño del problema y la cantidad de núcleos utilizados.

Se llevaron a cabo 4 soluciones distintas, una en la que se utiliza pasaje de mensajes y otras tres híbridas (combinación de pasaje de mensajes con memoria compartida), tal como muestra la Figura 2.

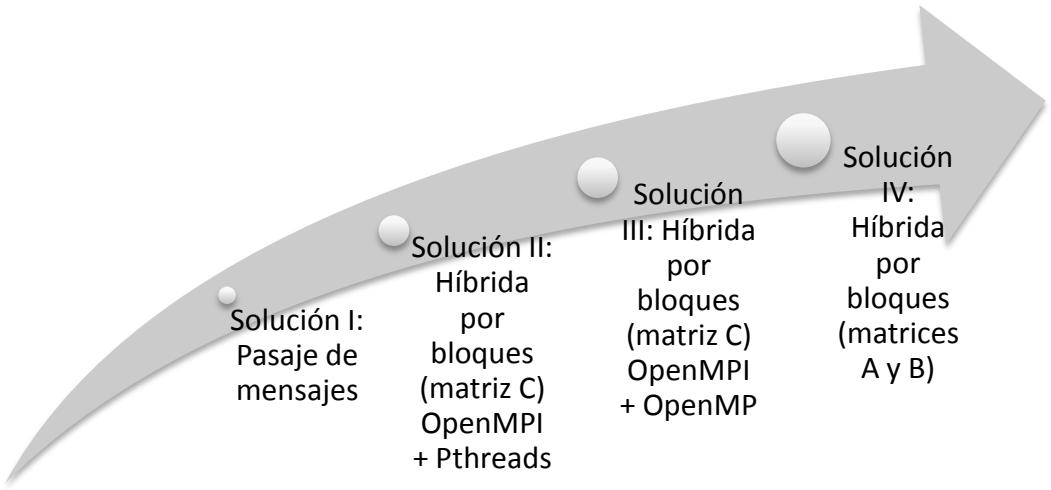

Fig. 2. Soluciones implementadas

3. Modelos de programación paralela

La elección del modelo de programación que se utiliza afecta la decisión del lenguaje de programación y de la librería a utilizar. Tradicionalmente el procesamiento paralelo se ha dividido en dos grandes modelos: memoria compartida y pasaje de mensajes [Don2002].

Memoria compartida: todos los datos accedidos por la aplicación se encuentran en una memoria global accesible por todos los procesadores paralelos. Esto significa que cada procesador puede buscar y almacenar datos de cualquier posición de memoria independientemente. Este modelo se caracteriza por la necesidad de la sincronización para preservar la integridad de las estructuras de datos compartidas.

Pasaje de mensajes: los datos son vistos como si estuvieran asociados a un procesador particular. De esta manera, se necesita de la comunicación entre ellos para acceder a un dato remoto. Generalmente, para acceder a un dato que se encuentra en una memoria remota, el procesador dueño de ese dato debe enviar el dato y el procesador que lo requiere debe recibirlo. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización.

Debido al avance de las arquitecturas paralelas y especialmente a la aparición de la arquitectura *multicore*, surge el modelo de programación híbrido en el cual se combinan las estrategias recientemente expuestas. La programación híbrida combina la memoria compartida con el pasaje de

mensajes [Don2002][Gra2003], aprovechando sus potencialidades. La comunicación entre procesos que pertenecen al mismo procesador físico puede realizarse utilizando memoria compartida (nivel micro), mientras que la comunicación entre procesadores físicos (nivel macro) se suele llevar a cabo por medio de pasaje de mensajes (memoria distribuida).

El objetivo de utilizar el modelo híbrido es aprovechar y aplicar las potencialidades de cada una de las estrategias que el modelo brinda, de acuerdo a la necesidad de la aplicación y a la arquitectura de prueba.

4. Jerarquía de memoria y programación híbrida

La performance de la jerarquía de memoria está determinada por dos parámetros de hardware. Latencia: tiempo entre que un dato es requerido y está disponible; ancho de banda: velocidad con la que los datos son enviados de la memoria al procesador. Tal como muestra la Figura 3, a medida que aumenta la capacidad se ve decrementada la velocidad pero asimismo disminuye el costo por bit.

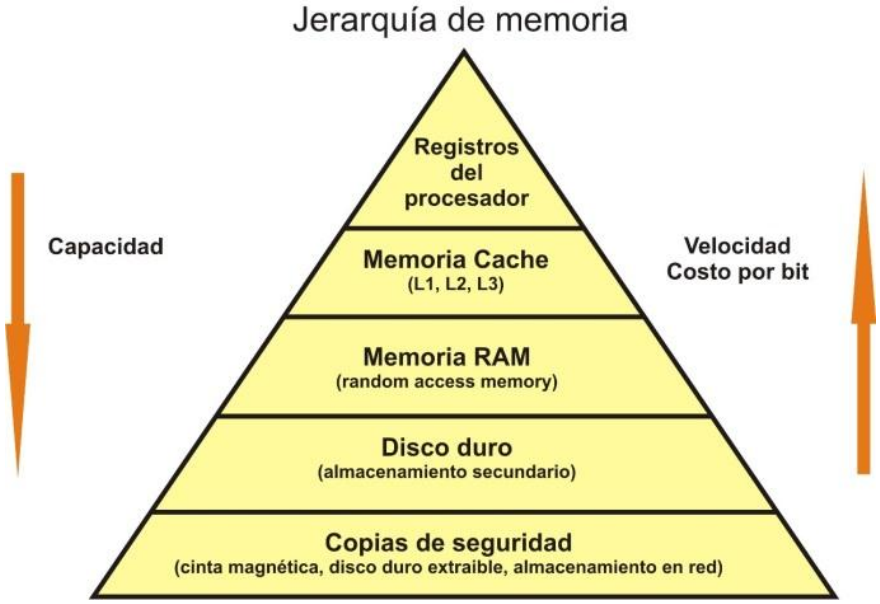

Fig. 3. Jerarquía de memoria tradicional

A continuación se describen los conceptos de localidad espacial y temporal de la *cache*. Luego se describe la programación híbrida como método de aprovechamiento de la arquitectura.

4.1. Localidad temporal y espacial de la *cache*

El concepto de localidad temporal hace referencia a que la *cache* mantiene los datos recientemente accedidos. Es decir, que si se aprovecha el acceso a los datos teniendo en cuenta este factor, la latencia en el acceso a los mismos disminuirá.

Por otro lado, el concepto de localidad espacial hace referencia a que cada vez que un dato es llevado a *cache*, se obtiene una línea de la misma. De esta forma, si se accede a datos contiguos, también se disminuye la latencia en el acceso a los datos (en una sola lectura a memoria, los datos contiguos estarán ya en memoria *cache*) [Gra2003].

4.2. Programación híbrida

La programación híbrida combina la memoria compartida con el pasaje de mensajes [Don2002][Gra2003], aprovechando las características propias de cada modelo de programación paralela. La comunicación entre procesos que pertenecen al mismo procesador físico puede realizarse utilizando memoria compartida (nivel micro), mientras que la comunicación entre procesadores físicos (nivel macro) se suele realizar por medio de pasaje de mensajes (memoria distribuida).

En el primer caso, los datos accedidos por la aplicación se encuentran en una memoria global accesible por los procesadores paralelos. Esto significa que cada procesador puede buscar y almacenar datos de cualquier posición de memoria independientemente uno de otro. Se caracteriza por la necesidad de la sincronización para preservar la integridad de las estructuras de datos compartidas.

En el pasaje de mensajes, los datos son vistos como asociados a un proceso particular. De esta manera, se necesita de la comunicación por mensajes entre ellos para acceder a un dato remoto. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización.

El objetivo de utilizar el modelo híbrido es aprovechar y aplicar las potencialidades de cada una de las estrategias que el mismo brinda, de acuerdo a la necesidad de la aplicación. Esta es un área de investigación de

interés actual, y entre los lenguajes que se utilizan para programación híbrida aparecen OpenMP [Omp2013] para memoria compartida y MPI [Mpi2013] para pasaje de mensajes.

5. Caso de estudio

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 4.

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

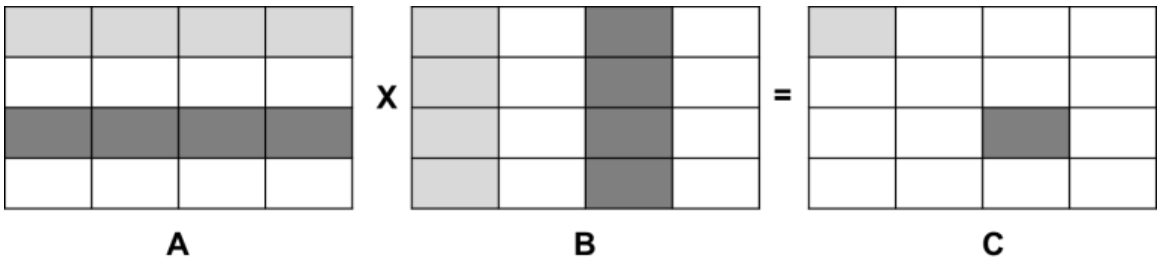

Fig. 4. Multiplicación de matrices

5.1. Soluciones Implementadas y Arquitectura Utilizada

Las soluciones implementadas fueron realizadas en dos grandes fases, como se resume en la Tabla 1.

Fase I	Fase II
a) Solución secuencial tradicional	a) Solución secuencial (matriz A y B por bloques)
b) Solución pasaje de mensajes (matriz C por bloques)	b) Solución híbrida II OpenMP (matriz C por bloques)
c) Solución híbrida I Pthreads (matriz C por bloques)	c) Solución híbrida III OpenMP (matriz A y B por bloques)

Tabla 1. Soluciones implementadas

Es importante destacar que en todas las soluciones implementadas se ha llevado a cabo un mapping manual de procesos y/o hilos a procesadores/núcleos. De esta manera, cada unidad de procesamiento tendrá

asignado un hilo/proceso según corresponda, teniendo en cuenta la localidad de los datos y la forma de interacción de los hilos/procesos de la aplicación para que por ejemplo los hilos de un mismo proceso en el caso híbrido, sean asignados a núcleos dentro del mismo procesador físico. En el anexo I se especifica la metodología mediante la cual se lleva a cabo este procedimiento.

5.2. Arquitectura utilizada

En ambas fases, el hardware utilizado para llevar a cabo las pruebas es un Blade de 16 servidores (hojas). Cada hoja posee 2 procesadores quad *core* Intel Xeón e5405 de 2.0 GHz; 14 hojas poseen 2 Gb de memoria RAM mientras que las dos restantes poseen 10GB. En todos los casos las características de la misma son las siguientes: memoria RAM compartida entre ambos procesadores; *cache* L2 de 2 X 6Mb compartida entre cada par de *cores* por procesador y *cache* L1 de datos de 32 KB propia de cada núcleo. El sistema operativo utilizado es Debian 6 de 64 bits [HP2011a][HP2011b].

5.3. Fase I

Los estudios experimentales fueron realizados en base a la implementación del algoritmo de multiplicación de matrices tradicional para el caso secuencial. Para las soluciones paralelas, se llevaron a cabo soluciones en las que los resultados (matriz C) se calculan por bloques. Las mismas se implementaron utilizando modelos de pasaje de mensajes e híbrido.

Tanto la solución secuencial como las paralelas fueron desarrolladas utilizando el lenguaje C. La solución paralela que utiliza pasaje de mensajes como mecanismo de comunicación entre procesos, usa para ello la librería OpenMPI [Mpi2013]. La solución híbrida utiliza la librería Pthreads [Pth2013] para memoria compartida junto a OpenMPI para el pasaje de mensajes.

A continuación se describen las soluciones implementadas. En todos los casos la multiplicación de matrices se realiza almacenando la matriz A por filas y la matriz B por columnas de manera de poder aprovechar la localidad espacial y temporal de la memoria *cache* en el acceso a los datos.

5.3.1. Solución Secuencial

Se resuelve secuencialmente el valor de cada posición de la matriz C según la Ecuación 1. Para mejorar el acceso a los datos se resuelve por filas de izquierda a derecha, y la matriz C se almacena por filas.

5.3.2. Solución con pasaje de mensajes

En este caso, el cálculo de la matriz C se realiza por bloques. Para ello cada proceso recibe las filas de A y las columnas de B necesarias para calcular el bloque de la matriz C que le fue asignado. La cantidad de bloques en que se divide la matriz C es divisible por la cantidad de procesos.

El algoritmo utiliza una interacción de tipo master/worker, donde el master trabaja tanto de coordinador como de worker. Se divide la matriz C en bloques a procesar y posteriormente se generan fases de procesamiento. Dado que todos los procesadores tienen la misma potencia de cómputo y que todos los bloques a procesar son del mismo tamaño, todos procesarán (aproximadamente) a la misma velocidad. De esta manera, en cada fase de procesamiento, el master reparte las filas de la matriz A y las columnas de la matriz B según el bloque correspondiente a cada worker, incluyendo un bloque para él; luego procesa su bloque y recibe de todos los demás los resultados para poder así pasar a la siguiente fase de procesamiento. La cantidad de fases se calcula de la siguiente manera: si b es la cantidad de bloques que se deben procesar y w la cantidad de workers (incluyendo al master que funciona como worker también), la cantidad de fases es b/w.

Es importante tomar en cuenta que cada proceso necesita almacenar las filas de la matriz A que va a procesar, las columnas de la matriz B y el bloque de la matriz C que genera como resultado.

Se puede resumir el algoritmo en la Figura 5:

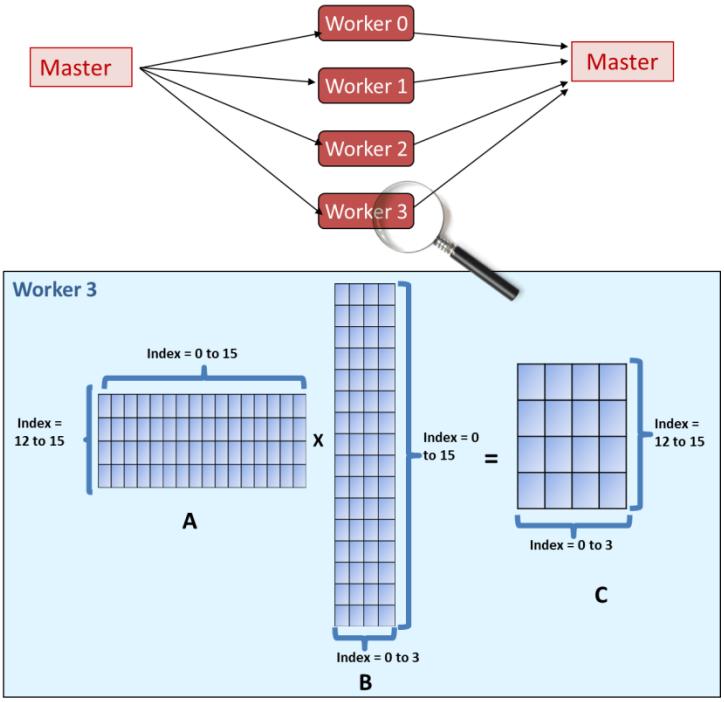

Fig. 5. Solución pasaje de mensajes

5.3.3. Solución híbrida (I) Pthreads

En esta solución existe un proceso por hoja que internamente genera 7 hilos para hacer su procesamiento. De esta manera, cada hoja trabaja con 8 hilos (los 7 generados más el proceso en sí). Se utiliza una estructura master/worker en la que uno de los procesos actúa como master, dividiendo la matriz C en bloques a ser procesados. Al igual que en el caso de pasaje de mensajes, el proceso master también actúa como worker y se generan las fases de procesamiento ya explicadas. En cada fase, una vez que reparte los bloques a los workers, genera los hilos correspondientes para procesar su propio bloque, dividiendo el mismo en filas para que cada hilo procese un subconjunto de las mismas. Los demás procesos worker actúan de la misma manera recibiendo datos y enviando sus resultados al proceso master.

El algoritmo se ilustra en la Figura 6:

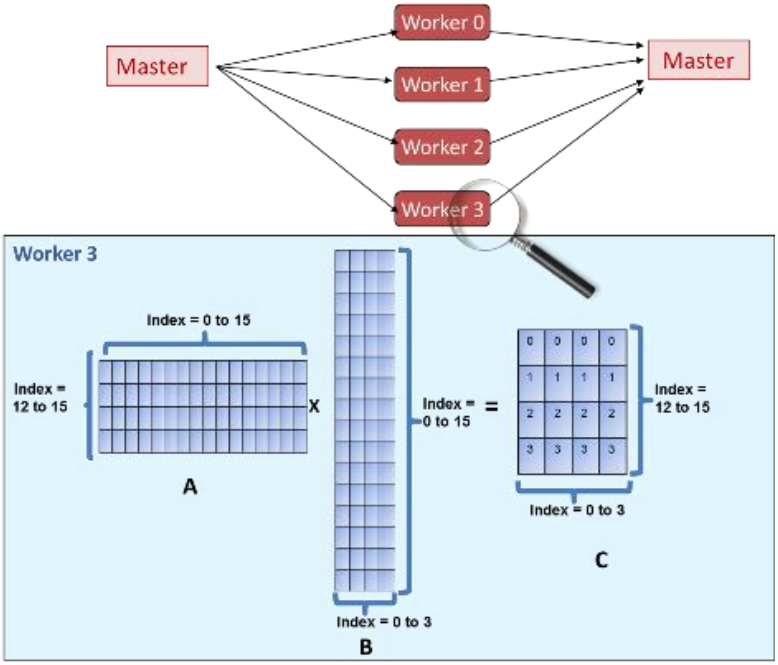

Fig. 6. Solución híbrida (matriz C por bloques)

5.3.4. Resultados

A continuación, pueden observarse los resultados obtenidos por las soluciones recién planteadas. Los resultados que se muestran se enfocan a analizar las soluciones en dos sentidos:

- El comportamiento al incrementar el tamaño del problema y la cantidad de núcleos (escalabilidad) [Kum1994][Leo2001]. En este caso, se procesaron matrices cuadradas de 1024, 2048, 4096, 8192 y 16384 filas y columnas.
- Comparar los tiempos de ejecución y eficiencia alcanzados con la solución híbrida con los obtenidos en la solución de pasaje de mensajes.

A continuación se muestran los resultados obtenidos en los experimentos realizados. Es necesario tener en cuenta que en los dos algoritmos utilizados, el proceso master es el que más memoria consume, por ello el mismo es mapeado para ser ejecutado en una hoja con 10GB de memoria RAM. Los tamaños de prueba de las matrices (cuadradas) así como de los bloques (cuadrados) se eligieron de manera de no generar swapping a disco en ninguna de las soluciones. Por otro lado, se minimizó la cantidad de bloques necesarios, de manera que sean divisibles por la cantidad de workers, minimizando la cantidad de fases de procesamiento.

En la Tabla 2 se muestran los tiempos de ejecución de la solución secuencial (Sec.). En la Tabla 3, los tiempos obtenidos por el algoritmo de pasaje de mensajes utilizando 16 y 32 núcleos (PM 16 y PM 32), el tamaño de los bloques (TBPM16 y TBPM32), los obtenidos por la solución híbrida con 16 y 32 núcleos (H(I)16 y H(II)32) y el tamaño de los bloques (TBH(I)16 y TBH(I)32). En todos los casos los tiempos de ejecución están expresados en segundos. En las pruebas se escala la dimensión de la matriz, así como también la cantidad de núcleos.

Tam. Matriz	Secuencial (seg.)
1024 * 1024	7,68
2048 * 2048	62,44
4096 * 4096	498,43
8192 * 8192	4019,53
16384 * 16384	32138,45

Tabla 2. Tiempos de ejecución secuencial

Tam.	TBPM 16	PM 16	TBPM 32	PM 32	TBH(I)16	H(I) 16	TBH(I)32	H(I) 32
1024	256	0,87	128	1,42	512	0,79	512	0,49
2048	512	5,58	256	4,01	1024	5,30	1024	2,54
4096	1024	39,28	512	23,96	2048	38,38	2048	18,98
8192	2048	306,72	1024	159,11	4096	302,38	4096	148,36
16384	1024	2558,96	1024	1290,92	8192	2410,28	8192	1186,82

Tabla 3. Tiempos de ejecución soluciones paralelas

En función de los resultados obtenidos, se observa que en la solución híbrida los tiempos de ejecución son en todos los casos mejores que los obtenidos por la solución con pasaje de mensajes. Asimismo, a medida que aumenta el tamaño del problema, la diferencia de tiempo entre ambas soluciones también aumenta, en favor de la solución híbrida.

5.3.5. Resultados Comparados

En la Tabla 4 se muestra la eficiencia alcanzada por las diferentes alternativas de prueba, mientras que en la Figura 7 puede verse un gráfico comparativo que muestra dicha información.

Tam.	PM16	H(I)16	PM32	H(I)32
1024	0,54	0,60	0,16	0,48
2048	0,69	0,73	0,48	0,76
4096	0,79	0,81	0,64	0,82
8192	0,81	0,83	0,78	0,84
16384	0,78	0,83	0,77	0,84

Tabla 4. Eficiencia

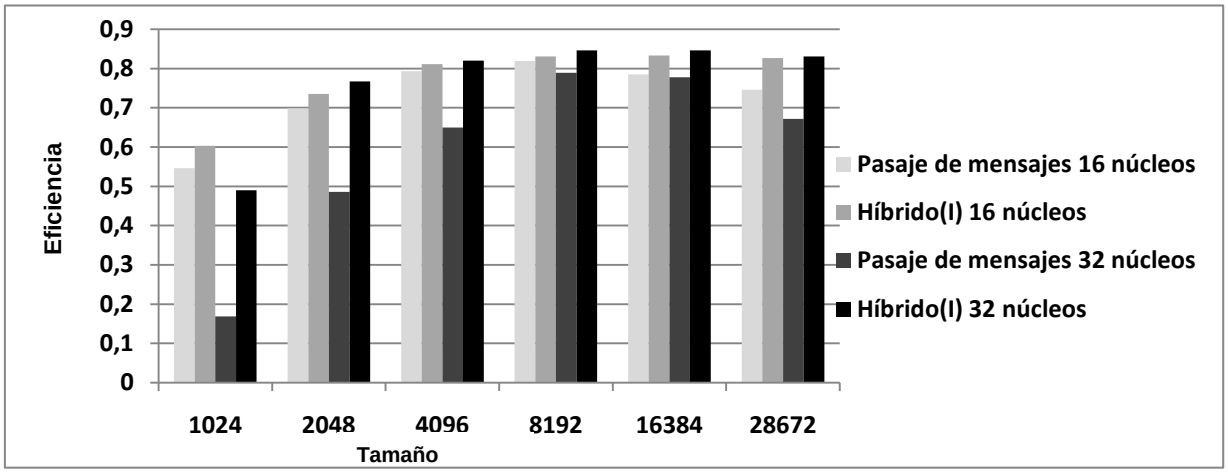

Fig. 7. Eficiencia

Los resultados permiten analizar por un lado, que en todos los casos la eficiencia alcanzada por la solución híbrida es mayor que la alcanzada por la solución que utiliza pasaje de mensajes, y que a medida que aumenta el tamaño del problema (para la misma cantidad de unidades de procesamiento), la eficiencia también se incrementa.

En el caso de pasaje de mensajes, al aumentar la cantidad de unidades de procesamiento a 32, para tamaños entre 1024 y 8192, se debe aumentar la cantidad de bloques a procesar para que cada worker pueda procesar al menos un bloque. Esto disminuye la eficiencia debido al aumento de comunicación y sincronización entre los procesos. Sin embargo, esto no sucede en el caso del algoritmo híbrido, ya que no es necesario aumentar la cantidad de bloques a procesar cuando se pasa de 16 a 32 unidades de procesamiento. Entonces al tener más unidades de cómputo para la misma cantidad de bloques, se disminuye, en lugar de aumentar como en el caso de mensajes, la cantidad de comunicación y sincronización; alcanzando una mejor eficiencia. Pero es necesario marcar que para el caso de matrices de 1024 *

1024 elementos, la eficiencia alcanzada por la solución que utiliza 32 núcleos es menor que la que utiliza 16. Esto se debe a que como el volumen de datos a procesar es pequeño, los costos de comunicación y sincronización para 32 núcleos frente al tiempo de cómputo son más altos que para 16 núcleos. No hay que olvidar que en el caso híbrido, para 16 núcleos, hay 2 procesos que internamente mediante la generación de hilos, utilizan 8 núcleos cada uno y que para 32, hay 4 procesos con las mismas características que los anteriores.

Por otro lado, se debe destacar que la eficiencia alcanzada por la solución de pasaje de mensajes para 16384 * 16384 elementos con 16 y 32 núcleos, se ve degradada frente a los demás tamaños para la misma cantidad de núcleos. El motivo son las limitaciones en la memoria principal disponible en cada hoja, que provoca que sea necesario aumentar la cantidad de bloques de procesamiento pues es necesario que los mismos sean más chicos para evitar el swapping. De esta manera se generan más fases de procesamiento y sincronización que atentan contra la eficiencia alcanzable por el algoritmo. Este impacto es más notorio para 16 núcleos que para 32, dado que para la misma cantidad de bloques, con 32 núcleos habrá menos fases de sincronización que para 16.

En referencia a la escalabilidad, los resultados obtenidos muestran que la solución híbrida es escalable y que el aumento del tamaño del problema incrementa la eficiencia lograda por el algoritmo.

Por otro lado, la comparación entre la solución de pasaje de mensajes con respecto a la híbrida permite ver que esta última es la que da como resultado mejores tiempos de ejecución.

En este sentido, se observa la mejora introducida por la solución híbrida que aprovecha las características del problema y la arquitectura utilizada.

La posibilidad de aprovechar la memoria compartida evita la replicación de datos en cada hoja. Para el caso particular del problema elegido como caso de estudio, evita replicar las filas de la matriz A y las columnas de la matriz B que le corresponden según el bloque que debe procesar en cada uno de los workers. Esto no ocurre en la solución que utiliza pasaje de mensajes, ya que cada uno de ellos maneja su propio espacio de memoria y por lo tanto debe tener una copia de las filas de la matriz A y de las columnas de la matriz B que

le corresponde, según el bloque que procesa. Esto se refleja en los tamaños de bloques necesarios para evitar el swapping.

En la solución que utiliza pasaje de mensajes, para tamaños grandes de matrices, se tiene mayor cantidad de fases de sincronización que en la solución híbrida, lo que provoca que el tiempo de ejecución y la eficiencia se vean degradados notablemente ya que se necesitan más fases de comunicación y sincronización que en el caso híbrido, por lo que se genera más overhead.

En función de las conclusiones recientemente expuestas, en que la solución híbrida arroja mejores resultados en todos los frentes de análisis realizados frente a la solución que utiliza pasaje de mensajes, es que se continuó la investigación sobre la solución híbrida de manera que permita explotar aún más las características de la jerarquía de memoria disponible en la arquitectura de prueba; haciendo una solución con alto grado de acoplamiento a la arquitectura en pos de obtener un mejor aprovechamiento de la misma. A continuación se presentan los algoritmos implementados fruto de la experimentación llevada a cabo.

5.4. Fase II

En función de los resultados obtenidos en la Fase I, la investigación continuó destinada a la optimización del algoritmo híbrido. Para ello, se buscó optimizar desde el algoritmo, las librerías de programación así como desde el compilador.

Inicialmente, se implementó la misma solución híbrida de la fase anterior pero utilizando la librería de memoria compartida OpenMP en lugar de Pthreads. La diferencia de tiempos obtenidos por las dos implementaciones, no es significativa, en ninguno de los casos supera el 10%; pero en la mayor parte de los casos, Pthreads arroja levemente mejores resultados que OpenMP.

Según un informe presentado por Intel, OpenMP tiene un overhead inicial en sus primitivas como el caso de *#pragma omp parallel for* [Lin2006]. Este overhead será significativo o no, dependiendo de cuanto representa (en porcentaje) respecto del tiempo total de ejecución del algoritmo. Lo que es

importante notar es que este *overhead* es inicial y no está relacionado a las operaciones que después se ejecuten.

En algoritmos que aprovechen eficientemente operaciones optimizadas de OpenMP, la diferencia de ejecutar el mismo algoritmo con Pthreads y con OpenMP será probablemente significativa.

Desde el punto de vista del esfuerzo de la programación, OpenMP simplifica notablemente la implementación del algoritmo dadas las directivas que provee. La misma permite paralelizar una solución secuencial en muy pocos pasos, dado su alto nivel de abstracción.

Las tendencias actuales en el ámbito del cómputo de altas prestaciones muestran que la librería más utilizada para la programación híbrida es OpenMP, dado que tiene un manejo muy eficiente de los hilos, de manera que a partir de que se define la cantidad de hilos a ser usados, *OpenMP* genera un pool de hilos, que va a ir reutilizando a medida que se necesitan. Se evita así la creación y destrucción de los mismos cada vez que se entra/sale de una zona paralela. Esto es ampliamente aprovechado cuando el algoritmo posee varias zonas paralelas.

En el caso de Pthreads habría que crear y destruir los hilos en cada una de ellas, o crear y mantener un pool de threads desde el algoritmo. Además, si el algoritmo necesitara de la sincronización entre los hilos o de paralelizaciones más complejas, como por ejemplo operaciones de reducción de variables compartidas, combinación de secciones críticas con zonas que no lo son, etc, estimativamente OpenMP arrojaría mejores resultados que Pthreads dado que la misma librería nos provee de mecanismos especializados y optimizados para ello; mientras que en Pthreads habría que implementarlo manualmente utilizando semáforos o variables condition.

Teniendo en cuenta que el objetivo principal es la optimización del algoritmo de multiplicación de matrices, los desarrollos experimentales se realizaron implementando la versión secuencial pero modificando la manera tradicional de resolverlo, tal como se explica en la Sección 5.4.1.

Para las soluciones paralelas, se implementaron dos variantes. En una de ellas los resultados (matriz C) se calculan por bloques al igual que en la fase anterior pero utilizando la librería OpenMP en lugar de Pthreads para memoria compartida [Lei2012a], mientras que en el segundo algoritmo paralelo, el

cálculo de la matriz C se realiza subdividiendo las matrices A y B en bloques para ser procesados. En ambas soluciones se utilizó el modelo de programación híbrida, combinando las librerías OpenMP y OpenMPI.

5.4.1. Solución Secuencial

Se resuelve secuencialmente el valor de cada posición de la matriz C según la Ecuación 1. La diferencia con la solución secuencial tradicional se basa en que la matriz A es subdividida en filas y la B en columnas de bloques pequeños para aprovechar la localidad de la *cache* L1.

5.4.2. Solución híbrida (II) OpenMP

El algoritmo utiliza una interacción de tipo master/worker, donde el master trabaja tanto de coordinador como de worker. El mismo divide la matriz C en bloques a procesar y posteriormente genera fases de procesamiento. Dado que todos los procesadores tienen la misma potencia de cómputo y que todos los bloques a procesar son del mismo tamaño, todos procesarán (aproximadamente) a la misma velocidad. De esta manera, en cada fase de procesamiento, el master reparte las filas de la matriz A y las columnas de la matriz B según el bloque correspondiente a cada worker, incluyendo un bloque para él. Procesa su bloque y luego recibe de todos los demás los resultados para poder así pasar a la siguiente fase de procesamiento. La cantidad de fases se calcula de la siguiente manera: si b es la cantidad de bloques que se deben procesar y w la cantidad de workers (incluyendo al master que funciona como worker también), la cantidad de fases es b/w. En esta solución existe un proceso por hoja que internamente genera 8 hilos para hacer su procesamiento.

Es importante tomar en cuenta que cada proceso necesita almacenar las filas de la matriz A que va a procesar, las columnas de la matriz B y el bloque de la matriz C que genera como resultado.

En cada fase, una vez que reparte los bloques a los workers, genera los hilos correspondientes para procesar su propio bloque dividiendo el mismo en filas para que cada hilo procese un subconjunto de las mismas. Los demás

procesos worker actúan de la misma manera recibiendo datos y enviando sus resultados al master.

Esta implementación, a nivel macro, es la misma que la solución híbrida (I) Pthreads con la diferencia que en lugar de utilizar la librería mencionada anteriormente, se utiliza OpenMP, teniendo que cambiar levemente la generación de hilos debido a las características de la última.

5.4.3. Solución híbrida (III)

Al igual que en la solución híbrida II, el algoritmo utiliza una interacción de tipo master/worker, donde el master trabaja tanto de coordinador como de worker. El mismo, divide la matriz A en filas de bloques y la B en columnas de bloques. Dado que todos los procesadores tienen la misma potencia de cómputo y que todos los bloques a procesar son del mismo tamaño, todos procesarán aproximadamente a la misma velocidad. De esta manera, el master reparte las filas de bloques de la matriz A (según el subconjunto de filas de C que debe calcular cada uno) y todos los bloques de la matriz B a cada worker, incluyéndose a sí mismo. Procesa sus filas de bloques de A y luego recibe de todos los demás los resultados. Es importante tener en cuenta que cada proceso necesita almacenar las filas de bloques de la matriz A que va a procesar, todas las columnas de bloques de la matriz B y las filas de la matriz C que genera como resultado.

Una vez que reparte los datos a los workers, se generan los hilos correspondientes para procesar por cada fila de bloques todas las columnas de bloques que la misma posee. Los demás procesos worker actúan de la misma manera recibiendo datos y enviando sus resultados al master.

Esta solución está directamente relacionada al concepto de localidad temporal, que hace referencia a que la *cache* mantiene los datos recientemente accedidos. Es decir que si se aprovecha el acceso a los datos teniendo en cuenta este factor, la latencia en el acceso a los mismos disminuirá. Por otro lado, el concepto de localidad espacial hace referencia a que cada vez que se trae un dato a *cache*, se obtiene una línea de la misma; de esta forma, si los datos accedidos son contiguos, también se disminuye la latencia en el acceso a los datos (en una sola lectura a memoria, los datos contiguos estarán ya en

memoria *cache*). Esta solución justamente aprovecha la localidad espacial y temporal de la *cache* L1.

Se puede resumir el algoritmo de la siguiente manera:

Proceso *master*:
1. Divide la matriz A y B en bloques.
2. Comunica las filas de bloques correspondientes de la matriz A y todas las columnas de bloques de la matriz B a los procesos *worker*.
3. Genera los hilos y procesan la información.
4. Recibe los resultados de los procesos *worker*.

Procesos *worker*
1. Reciben los datos a procesar
2. Generan los hilos y procesan los datos.
3. Comunican los resultados al proceso *master*.

La Figura 8 muestra un gráfico representativo de la solución híbrida (II) con tamaño de bloque 2 *2

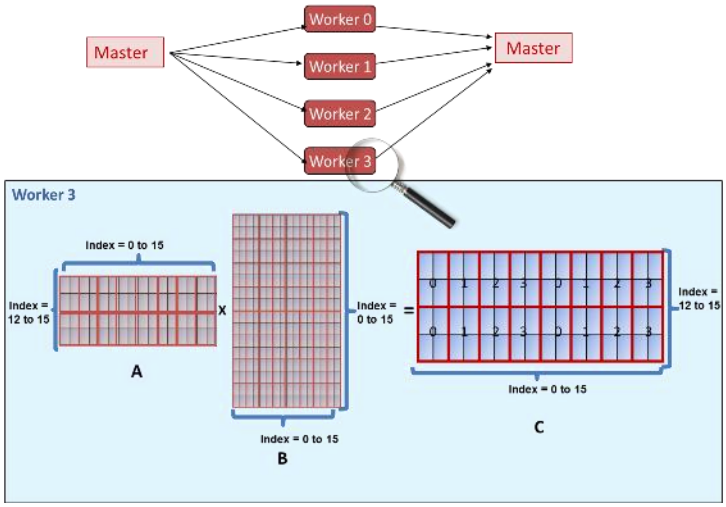

Fig. 8. Solución híbrida (III)

5.4.4. Resultados

En la Tabla 5 se muestran los tiempos de ejecución de la solución secuencial (Sec.). En la Tabla 6, los tiempos obtenidos por la solución híbrida (II) utilizando 16 y 32 núcleos (H(II) 16 y H(II) 32) y el tamaño de los bloques (TBH(II)16 y TBH(II)32) y los obtenidos por la solución híbrida (III) con 16 y 32 núcleos (H(III)16 y H(III)32) y el tamaño de los bloques (TBH(III)16 y TBH(III)32). En todos los casos los tiempos de ejecución están expresados en segundos. En las pruebas se escala tanto la dimensión de la matriz, como la cantidad de núcleos. En la Figura 9 se muestra la eficiencia lograda por ambas soluciones. Es importante tomar en cuenta que a diferencia de la fase anterior,

las pruebas se realizaron utilizando también optimización del compilador (nivel –O3) como método complementario para obtener resultados optimizados.

Tam. Matriz	Secuencial (seg.)
1024 * 1024	1,62
2048 *2048	12,99
4096 * 4096	103,92
8192 * 8192	831,32
16384 * 16384	6652,68

Tabla 5. Tiempos de ejecución secuenciales

Tam.	TBH(II)16	H(II)16	TBH(II)32	H(II)32	TBH(III)16	H(III)16	TBH(III)32	H(III)32
1024	512	0,24	512	0,23	64	0,25	64	0,25
2048	1024	1,62	1024	1,20	64	1,69	64	1,21
4096	2048	16,85	2048	7,96	64	10,24	64	6,09
8192	1024	127,29	4096	82,90	64	70,12	64	39,22
16384	1024	1212,67	8192	740,87	64	525,44	64	279,97

Tabla 6. Tiempos de ejecución paralelos

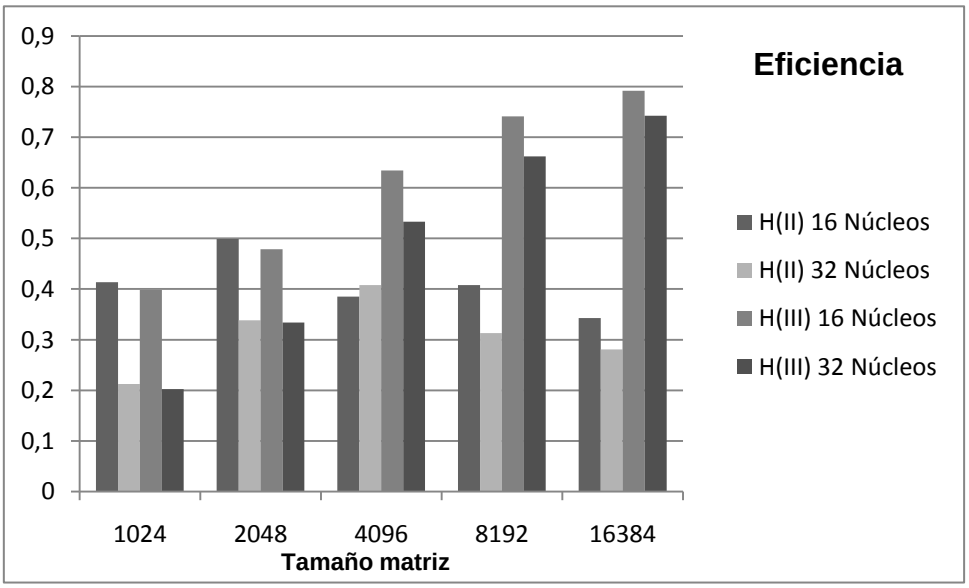

Fig. 9. Eficiencia

En función de los resultados obtenidos, se observa que en la solución híbrida (II) los tiempos de ejecución obtenidos para tamaños de matriz menores que 4096 son en todos los casos mejores que los obtenidos por la solución híbrida (III). Mientras que a partir de 4096, los resultados son exactamente inversos. Esto se debe a que para tamaños menores a 4096, en ambas soluciones los datos necesarios entran aproximadamente en *Cache* L1 y la forma de resolución del algoritmo II, al requerir menos operaciones que la solución III, arroja mejores resultados. Mientras que para tamaños mayores, los

datos entrarán en *Cache* L1 para la solución híbrida (III) mientras que no para el algoritmo (II), y el impacto de ello, provoca que los fallos de *Cache* L1 del algoritmo (II) retrasen notablemente los tiempos de ejecución.

Los resultados obtenidos permiten analizar que la solución híbrida (III) alcanza una eficiencia mucho mayor al escalar el tamaño del problema del que alcanza el algoritmo (II), ya que se aprovecha la jerarquía de memoria existente en el sistema. Esto se aplica también cuando se escala en la cantidad de núcleos de procesamiento utilizados. Esta diferencia entre un algoritmo y el otro se incrementa al aumentar el tamaño del problema dado que cuando los datos necesarios para ser procesados dejan de entrar en *cache* L1 en el algoritmo (II), sí lo hacen en el (III), verificando la idea original de la que parte este trabajo que es justamente el impacto del aprovechamiento de la arquitectura en la performance alcanzable por un algoritmo.

Como conclusión se puede indicar que aprovechar la localidad espacial y temporal de los datos en la *cache* L1, mejora notablemente la eficiencia obtenida a medida que crece el tamaño de los datos (independientemente de la cantidad de núcleos utilizados) a procesar, si se lo compara con una solución que no lo aprovecha. Para poder comprobar empíricamente las conclusiones alcanzadas, se realizó un análisis más profundo acerca del aprovechamiento de la jerarquía de memoria utilizando para ello contadores de hardware. En la siguiente sección se explica la metodología utilizada para el acceso a la información junto a los resultados obtenidos.

6. Profiling y contadores de hardware

El profiling es una técnica que se utiliza para obtener información acumulada de diferentes eventos que el hardware es capaz de contabilizar. Entre los eventos posibles pueden enumerarse los relacionados a la CPU (número de ciclos, cantidad de instrucciones, etc.); eventos relacionados a la *cache* (*cache* misses, *cache loads*, *cache stores*, etc.); y eventos relacionados a la Translation Lookaside Buffer o TLB, entre otros.

Existen diferentes herramientas que permiten realizar *profiling*. La que se utiliza en este trabajo es Perf [Per2013], que se detalla a continuación.

6.1. Perf

Perf (*performance counter subsystem* para sistemas LINUX), es una librería dedicada a realizar *profiling* en función de los contadores de hardware disponibles en la arquitectura.

Los contadores de hardware son registros del procesador que cuentan eventos de hardware, como por ejemplo cantidad de instrucciones ejecutadas y cache-misses.

Una de las principales ventajas que ofrece Perf frente a otras librerías dedicadas al profiling es que no se debe instrumentar, de manera que no es necesario modificar el código de la aplicación para obtener la información que se busca. Incluso en aplicaciones donde sólo se cuenta con el binario de la misma, se podrá acceder a la información buscada sin necesidad de tener el código fuente.

Sin embargo, perf posee una limitación importante, dado que no es una herramienta diseñada para aplicaciones distribuidas. De esta manera, pensando en un *cluster* de *multicore*, la librería solo toma en cuenta los contadores de hardware del procesador donde se lanza la ejecución. En este trabajo, en el que se utiliza la multiplicación de matrices cuadradas, esto no representa una limitación a la hora de contabilizar los eventos porque al ser una aplicación regular, ejecutándose en una arquitectura homogénea con balance de carga, todos los procesadores deben tener aproximadamente, la misma información.

Para poder utilizar perf, será necesario indicar en la línea de comandos junto al ejecutable de la aplicación la lista de eventos que se quieren monitorizar, de la siguiente manera:

perf stat -e lista de eventos separados por comas nombreDelBinario [parámetros (dependiendo de la aplicación)]

Hay diferentes opciones en cuanto a los eventos. Pueden utilizarse directamente los nombres de los que ya están definidos en perf o los que el fabricante del hardware utiliza para eventos específicos de cada arquitectura. Además, por ejemplo, los eventos pueden medirse a nivel de usuario, a nivel de kernel, etc… [Per2013].

6.2. Resultados obtenidos

Para poder comprobar que las diferencias de tiempos obtenidos por las soluciones híbridas (II) y (III) están relacionadas particularmente al aprovechamiento de la *cache* L1 es que se han realizado pruebas experimentales en las cuales mediante la librería perf se accedió a la información almacenada en los contadores de hardware.

La librería provee acceso a diferentes contadores relacionados a la *cache*, tales como cache-misses, L1-dcache-loads, L1-dcache-load-misses, LLC-loads, entre otros. Debe tenerse en cuenta que los resultados obtenidos, como ya se mencionó, corresponden al procesador donde se lanzó la ejecución de la aplicación.

Dadas las características de la aplicación, en la que la mayor parte del tiempo se acceden a datos de la *cache* que no son modificados (datos de las matrices A y B), los contadores de hardware más relevantes son cache-misses (cantidad de accesos a memoria que no pudieron ser servidos por ninguno de los niveles de *cache*) y L1-dcache-load-misses (cantidad de veces que se intentó leer un dato de *cache* L1 y este no se encontraba en la misma).

En las Tablas 7 y 8 se muestran los resultados obtenidos utilizando 16 núcleos de procesamiento, mientras que en las Tablas 9 y 10 los obtenidos con 32 núcleos. En ambos casos los datos que se detallan son: tamaño de la matriz (n x n), tamaño del bloque de datos utilizado, tiempo de ejecución y valores arrojados por los dos contadores de hardware. El porcentaje de diferencia entre ambas soluciones ((Max-Min)/Min)*100 se muestra en la Figura 10 mientras que el porcentaje de diferencia obtenidas por ambos contadores, calculados de la misma manera que en el caso anterior para 16 y 32 núcleos se muestra en la Tabla 11.

Tam.	Tam Bloque H(II)	Tiempo ejecución H(II)	Cache-misses H(II)	L1-dcache-load-misses H(II)
1024	512	0,169787	90833	39215937
2048	1024	1,590069	7007919	1144669820
4096	2048	14,549366	52331548	11710962997
8192	1024	107,45902	353382411	150944601494
16384	1024	999,836686	3319932856	1235070343976

Tabla 7. Solución híbrida (II) 16 Núcleos

Tam.	Tam Bloque H(III)	Tiempo ejecución H(III)	Cache-misses H(III)	L1-dcache-load-misses H(III)
1024	64	0,200437	111712	2780171
2048	64	1,248707	622774	12055241
4096	64	8,581281	2107495	84902512
8192	64	63,357448	9620608	1422336083
16384	64	486,539785	75615320	5153111864

Tabla 8. Solución híbrida (III) 16 Núcleos

Tam.	Tam Bloque H(II)	Tiempo ejecución H(II)	Cache-misses H(II)	L1-dcache-load-misses H(II)
1024	512	0,178317	204275	9676612
2048	1024	1,13594	5363830	571470855
4096	2048	8,93149	35368586	6018801855
8192	1024	61,209796	247342621	76685071654
16384	1024	544,72132	1984860405	625413462057

Tabla 9. Solución híbrida (II) 32 Núcleos

Tam.	Tam Bloque H(III)	Tiempo ejecución H(III)	Cache-misses H(III)	L1-dcache-load-misses H(III)
1024	64	0,178167	115983	2410420
2048	64	0,876612	331756	6447472
4096	64	5,259647	949626	90814423
8192	64	35,410192	5289732	351520078
16384	64	257,835239	43687045	5850873036

Tabla 10. Solución híbrida (III) 32 Núcleos

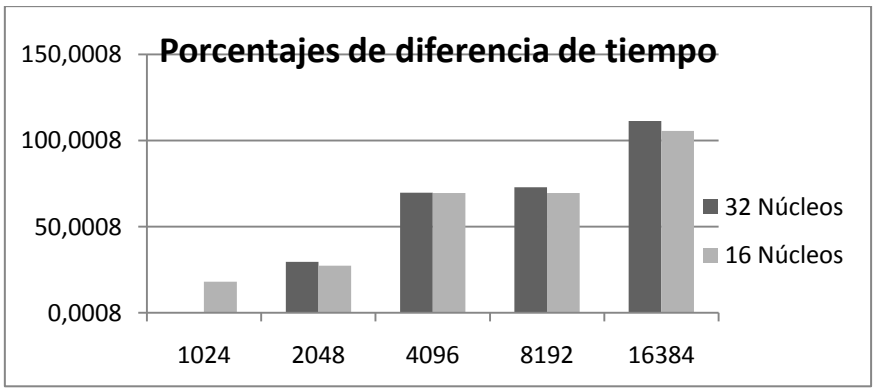

Fig 10. Porcentajes de diferencia de tiempo

Tam.	Cache-misses 16 núcleos	L1-dcache-load-misses 16 núcleos	Cache-misses 32 núcleos	L1-dcache-load-misses 32 núcleos
1024	22,98	1310,55	76,12	301,44
2048	1025,27	9395,20	1516,79	8763,48
4096	2383,11	13693,42	3624,47	6527,58
8192	3573,18	10512,44	4575,90	21715,27
16384	4290,55	23867,46	4443,36	10589,23

Tabla 11. Porcentaje de diferencia contadores de hardware

En función de los resultados obtenidos, se observa que en la solución híbrida (II) los tiempos de ejecución obtenidos para tamaños de matriz 1024, utilizando 16 núcleos, son mejores que los obtenidos por la solución híbrida (III). Para todas las demás pruebas, los resultados son exactamente inversos. Esto se debe a que para tamaños menores de 1024, en ambas soluciones los datos necesarios entran aproximadamente en *Cache* L1 y la forma de resolución del algoritmo II, al requerir menos operaciones que la solución III, arroja mejores resultados. En cambio para tamaños mayores, los datos entrarán en *Cache* L1 para la solución híbrida (III) mientras que no para el algoritmo (II), y el impacto de ello, provoca que los fallos de *Cache* L1 del algoritmo (II) retrasen notablemente los tiempos de ejecución. Esto es verificado en la comparación de la información arrojada por los contadores de hardware.

6.3. Análisis de resultados

Los resultados obtenidos permiten analizar por un lado, que la solución híbrida (II) alcanza tiempos de ejecución menores que el algoritmo (I), tal como se muestra en el gráfico de porcentaje de diferencia de tiempos, ya que se aprovecha la jerarquía de memoria existente en el sistema. Esto se aplica también cuando se escala en la cantidad de núcleos de procesamiento utilizados. Esta diferencia entre un algoritmo y el otro se incrementa al aumentar el tamaño del problema dado que cuando los datos necesarios para ser procesados dejan de entrar en *cache* L1 en el algoritmo (I), sí lo hacen en el (II), verificando la idea original de la que parte este trabajo que es justamente

el impacto del aprovechamiento de la arquitectura en la performance alcanzable por un algoritmo. Esto es verificado si se analizan los porcentajes de diferencia de cache-misses y L1-dcache-load-misses. En todos los casos, estos valores permiten ver que el algoritmo híbrido (I) genera siempre más de un 20% de fallos de *cache* que el algoritmo (II), llegando a porcentajes de diferencia de más del 100% a medida que aumenta el tamaño de las matrices.

Finalmente se podrá concluir que el algoritmo híbrido (II) efectivamente aprovecha la localidad espacial y temporal de los datos en la *cache* L1, mejorando notablemente los tiempos de ejecución obtenidos a medida que crece el tamaño de los datos (independientemente de la cantidad de núcleos utilizados) a procesar si se lo compara con una solución que no lo aprovecha, tal como lo reflejan los contadores de hardware a los que se puede acceder mediante la librería perf, permitiendo verificar la hipótesis de la que parte este trabajo. Sin embargo, hay que destacar que la mejora reflejada en los contadores no tiene una relación lineal con la mejora de tiempos obtenidos. Esto permite concluir que la optimización por el camino de la utilización de jerarquía de memoria en principio estaría alcanzada (considerando desde la *cache* L1 hacia la memoria compartida). Para profundizar la optimización deberían analizarse otro tipo de optimizaciones tales como unrolling loops, entre otros. Es importante considerar que los algoritmos fueron compilados con el nivel máximo de optimización dado por el compilador (-O3).

Las líneas de investigación futuras incluyen el análisis de la eficiencia energética de los algoritmos que aprovechan la localidad de los datos y de los que no lo hacen, a fin de estudiar la influencia sobre dicho parámetro, y la adaptación de los algoritmos y su posterior análisis sobre *clusters* heterogéneos [21][22], incluyendo la combinación de *multicore* con GPU.

7. Conclusiones finales

A lo largo del trabajo de investigación se desarrollaron soluciones que permiten aprovechar la jerarquía de memoria de la arquitectura de experimentación.

Desde el punto de vista de la performance alcanzada, a medida que se implementan soluciones a más bajo nivel, los resultados obtenidos muestran

que el aprovechamiento de la misma permite sacar más provecho de la arquitectura, acercándose al óptimo. Sin embargo, hay que tener en cuenta que las mismas aumentan el nivel de acoplamiento con la arquitectura y los costos de implementación desde la perspectiva del esfuerzo de programación.

Dependiendo del tipo de aplicación que sea necesaria desarrollar y del compromiso entre performance a obtener y costos de implementación, las soluciones aprovecharán en mayor o menor medida la arquitectura.

En el área del cómputo de altas prestaciones, la mejora de la performance alcanzable por un algoritmo es el objetivo principal, dado el tipo de algoritmos que se deben ejecutar y las exigencias de tiempos de respuesta de las aplicaciones. Es por ello que todas aquellas optimizaciones posibles sobre los algoritmos son necesarias y fundamentales para cumplir su objetivo, independientemente de los costos de implementación. Si se piensa, por ejemplo, en la simulación climática que mediante parámetros permite predecir el clima futuro, estamos ante la presencia de una aplicación que requiere ser ejecutada con la mayor optimización posible dado que de nada sirve simular para predecir el clima de las próximas 24 horas si la simulación toma más de 24 horas para obtener resultados.

Para finalizar, puede concluirse que el aprovechamiento de la jerarquía de memoria de las arquitecturas *cluster* de *multicore* impacta directamente en la performance alcanzable por las aplicaciones paralelas.

8. Anexo I: Mapping

Los sistemas operativos tradicionales realizan una asignación (mapping) de hilos y procesos a procesadores y núcleos utilizando para ello diferentes técnicas de scheduling.

Sin embargo, hay mecanismo que permiten al programador de las aplicaciones realizar el mapping manualmente, tanto para la asignación de hilos como de procesos, con la diferencia de que en el primer caso se debe modificar el código de la aplicación; mientras que para la asignación de procesos, el mapping puede realizarse en el momento de la ejecución utilizando directamente el binario.

En función de la arquitectura que se utiliza en este trabajo, es decir, *cluster* de *multicore*, se utilizaron ambas alternativas de mapeo combinándolas entre sí para obtener una mejor performance global del sistema. Esto es aplicable ya que los algoritmos estudiados utilizan un modelo hibrido, combinando memoria compartida con pasaje de mensajes.

8.1. Mapping de procesos

Open MPI es un proyecto de código abierto que implementa el standard de MPI y que como funcionalidad extra al mismo, provee directivas para la asignación explícita de procesos a núcleos de procesamiento. Para ello requiere de dos archivos llamados rankfile y hostfile.

En el archivo hostfile se define el número de núcleos disponibles en el sistema y el nombre de la máquina dentro de la red. El formato del archivo es el siguiente:

hostNameX slots = nro de núcleos

hostNameY slots = nro de núcleos

En el archivo rankfile se define una entrada por cada proceso de la siguiente manera:

rank N = hostNameX slot = nro de cpu

rank M = hostNameY slot = nro de cpu:nroNúcleo

La Figura 11 muestra un esquema en el que se especifica qué representan los parámetros nro de cpu, y nroNúcleo.

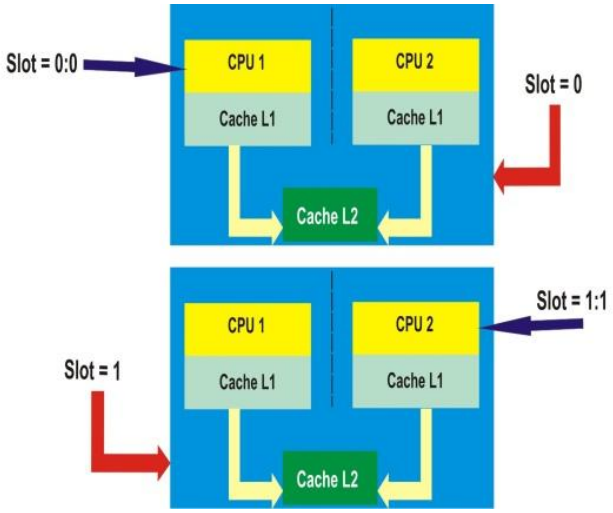

Fig. 11. Mapping manual

Estos dos archivos deben ser pasados como parámetro en el momento de la ejecución.

La ventaja de esta alternativa es que el código fuente queda intacto, no es necesario modificarlo para agregar esta funcionalidad y como consecuencia el mapping no consume tiempo de ejecución. La desventaja es que la asignación es estática, ya que como es pasada como parámetro en el momento de la ejecución, no puede modificarse dinámicamente.

8.2. Mapping de hilos

En los sistemas de memoria compartida no existe una función específica para asignar un hilo a un determinado núcleo de procesamiento. En lugar de ello, existe una función que permite definir la afinidad de cada tarea.

La afinidad permite especificar a cuáles de todos los núcleos existentes en un sistema se puede asignar un determinado proceso o hilo. Para poder asignar los mismos a un núcleo particular es necesario establecer que el único procesador/*core* planificable es al que se desea asignarlo.

Los sistemas operativos basados en Unix proveen una llamada al sistema que permite definir la afinidad explícitamente. Ésta debe ser utilizada en ambientes con memoria compartida ya que puede administrar los núcleos y procesadores de la máquina sobre la cual está corriendo el sistema operativo. El encabezado de esta función es el siguiente:

sched_setaffinity(pid_t pid, unsigned long cpusetsize, cpu_set_t *mask)

Dónde:

- pid: es el identificador del proceso al que se le define la afinidad. Si el valor es 0, se asume que es el proceso en ejecución. Es importante tomar en cuenta que los hilos que pertenecen a un proceso tienen el mismo PID, pero sin embargo, la afinidad pertenece a cada hilo en forma independiente y es almacenada en el contexto del mismo.
- cpusetsize: es la longitud (en bytes) de los datos apuntados por el parámetro mask.
- mask: representa la máscara de afinidad. Consiste en una máscara de bits en la cual cada uno de ellos representa un procesador (lógico) en el sistema. El orden se establece desde el bit menos significativo que

corresponde al primer procesador lógico hasta el bit más significativo que corresponde al último procesador lógico del sistema.

Si bit = 0 ese procesador no será planificable

Si bit = 1 ese procesador será planificable

Por defecto cuando un hilo/proceso se crea es planificable a todos los núcleos existentes en el sistema. Para poder realizar el mapping de un proceso o hilo es necesario establecer que el único procesador planificable es al que queremos asignarlo.

Una de las ventajas que presenta esta técnica de planificación es la posibilidad de modificar dinámicamente (durante la ejecución) la planificación de un determinado proceso. Esto es posible debido a que la llamada del sistema se puede incluir en el código de la aplicación.

9. Anexo II: Librerías de programación paralela – comparación

A continuación se describen las librerías de programación paralela utilizadas en este trabajo para finalmente realizar una comparación de las mismas.

9.1. Pthreads

A lo largo del tiempo, los fabricantes de hardware han implementado sus propias versiones para el manejo y administración de hilos. Estas versiones difieren mucho unas de otras, dificultando a los programadores desarrollar aplicaciones multihilos que sean portables. Por este motivo, en el año 1995 la IEEE estableció el standard POSIX (Portable Operating System Interface). La última versión que existe es la IEEE Std 1003.1, 2004 Edition [Pth2013].

Pthreads es una librería que implementa el standard POSIX y está compuesto por un conjunto de tipos y llamadas a procedimientos en el lenguaje de programación C que incluye un header file y una librería de hilos que forma parte por ejemplo de la librería libc, entre otras. Se utiliza para la programación de aplicaciones paralelas que utilizan memoria compartida.

Las subrutinas que conforman la API de Pthreads pueden ser clasificadas en cuatro grandes grupos:

- Manejo de hilos: rutinas que trabajan directamente con los threads - crear, separar, unir, etc. También incluyen funciones para establecer/consultar atributos del hilo (acoplables, scheduling, etc.).
- Mutex: rutinas que se ocupan de la sincronización y exclusión mutua.
- Mutex (semáforos) proporciona funciones para crear, destruir, bloquear y desbloquear semáforos. Estos se complementan con las funciones de atributos de semáforos que establecen o modifican los atributos asociados con los mismos.
- Variables condición: rutinas para el manejo de la sincronización por condición entre hilos que comparten semáforos. Provee funciones para crear, destruir, wait y signal sobre las variables. También provee funciones para establecer/consultar atributos de las mismas.
- Sincronización: rutinas que manejan locks y barreras.

9.2. OpenMP

OpenMP es una interface de programación (API) definida por un conjunto de fabricantes de hardware y software entre los que se encuentran: Sun Microsystems, IBM, Intel, AMD, entre otros [Omp2013].

Provee de una interface de programación portable y escalable para desarrolladores de aplicaciones paralelas que utilizan memoria compartida.

Esta API soporta C/C++ y Fortran en múltiples arquitecturas, incluyendo LINUX y Windows NT. Provee varios constructores y directivas para especificar regiones paralelas, trabajo compartido, sincronización y variables de entorno.

Está constituido por los siguientes tres componentes:

- Directivas de compilación
- Biblioteca de rutinas en tiempo de ejecución
- Variables de entorno

9.3. OpenMPI

MPI es una interface de pasaje de mensajes creada para resolver el inconveniente que surgió luego de que cada fabricante de hardware definiera su propia interfase de comunicación, generalmente incompatible con las

demás. El objetivo de MPI es solucionar este problema definiendo para ello un standard [Mpi2013].

MPI es una librería que puede ser utilizada para desarrollar programas que utilizan pasaje de mensajes (memoria distribuida) utilizando para ello los lenguajes de programación C o Fortran. El standard MPI define tanto la sintaxis como la semántica del conjunto de rutinas que pueden ser utilizadas en la implementación de programas que utilicen pasaje de mensajes. Brinda más de 125 rutinas, pero posee unas pocas claves conceptuales que facilitan notablemente su utilización.

Una de las implementaciones de este standard es OpenMPI que es la que se utiliza en este trabajo, dado que provee de mecanismos para realizar el mapping (asignación) de procesos a núcleos para ser ejecutados manualmente.

10. Bibliografía

- [Cha12] Chai L., Gao Q., Panda D. K., “Understanding the impact of multi-core architecture in cluster computing: A case study with Intel Dual-Core System”. IEEE International Symposium on Cluster Computing and the Grid 2007 (CCGRID 2007), pp. 471-478. 2007.
- [Don2002] Dongarra J., Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. “Sourcebook of Parallel computing”. Morgan Kaufmann Publishers 2002. ISBN 1558608710 (Capítulo 3).
- [Bur2010]Burger T. “Intel Multi-Core Processors: Quick Reference Guide ”http:// cachewww.intel.com/cd/00/00/23/19/231912_231912.pdf. (2010).
- [Int2012]http://www.intel.com/support/sp/processors/xeon/sb/cs-007758.htm (2012).
- [Lei2011a]Leibovich F., Gallo S., De Giusti L., Chichizola F., Naiouf M., De Giusti A. “Comparación de paradigmas de programación paralela en cluster de multicores: Pasaje de mensajes e híbrido. Un caso de estudio”. Proceedings del XVII Congreso Argentino de Ciencias de la Computación (CACIC 2011). Págs. 241-250. ISBN 978-950-34-0756-1.

- [Lei2012a]Leibovich, F., Naiouf, M., De Giusti L., Tinetti, F. G., De Giusti E. “Hybrid Algorithms for Matrix Multiplication on Multicore Clusters”. Julio 2012. WorldComp’12.
- [And2000]Andrews G. “Foundations of Multithreaded, Parallel and Distributed Programming”. Addison Wesley Higher Education 2000. ISBN-13: 9780201357523.
- [Lei2011b]Leibovich, F., De Giusti, L., Naiouf, M. “Parallel Algorithms on Clusters of Multicores: Comparing Message Passing vs Hybrid Programming”. Julio 2011. WorldComp’11.
- [Gra2003]Grama A., Gupta A., Karpyis G., Kumar V. “Introduction to Parallel Computing”. Pearson – Addison Wesley 2003. ISBN: 0201648652. Segunda Edición (Capítulo 3).
- [HP2011a]HP, "HP BladeSystem". http://h18004.www1.hp.com/products/blades/components/c-class.html. (2011).
- [HP2011b]HP, "HP BladeSystem c-Class architecture". http://h20000. www2.hp.com/bc/docs/support/SupportManual/c00810839/c00810839.pdf (2011).
- [Omp2013]https://computing.llnl.gov/tutorials/openMP (2013).
- [Mpi2013]http://www.open-mpi.org (2013).
- [Pth013]https://computing.llnl.gov/tutorials/pthreads (2013)
- [Kum1994]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.
- [Leo2001]Leopold C., “Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches”. Wiley, 2001. ISBN: 0471358312(Capítulos 1, 2 y 3).
- [Per2013]https://perf.wiki.kernel.org/index.php/Main_Page (2013).
- [Lin2006]Lindberg P., “Basic OpenMP Threading Overhead”. Intel, 2006.
- [Cha2007]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.
- [Bis2008]Bischof C., Bucker M., Gibbon P., Joubert G., Lippert T., Mohr B., Peters F. (eds.), Parallel Computing: Architectures, Algorithms and

Applications, Advances in Parallel Computing, Vol. 15, IOS Press, February 2008.