Programación híbrida en arquitecturas *cluster* de *multicore*. Escalabilidad y comparación con memoria compartida y pasaje de mensajes.

Fabiana Leibovich, Armando De Giusti, Marcelo Naiouf, Laura De Giusti, Franco Chichizola

Instituto de Investigación en Informática LIDI (III-LIDI), Facultad de Informática, Universidad Nacional de La Plata, 50 y 120 2do piso, La Plata, Argentina.
{fleibovich, degiusti, mnaiouf,ldgiusti,francoch}@lidi.info.unlp.edu.ar

Abstract. En este trabajo se analiza la programación híbrida en arquitecturas de *cluster* de *multicore*, en donde se combinan los modelos de memoria compartida y pasaje de mensajes. Para ello se utiliza una aplicación de procesamiento de imágenes como caso de estudio, que permite escalar fácilmente el volumen de datos a procesar.
Se estudia la mejora introducida por el uso de una estrategia híbrida en dos sentidos: por una parte, al crecer el tamaño del problema (escalabilidad), y por la otra comparando la solución con otras puras de memoria compartida o de pasaje de mensajes.
En el estudio experimental se analiza *speedup* y tiempo de ejecución,.utilizando una arquitectura *cluster* de *multicore* (en particular un blade).

Keywords: programación híbrida, *cluster*, *multicore*.

1 Introducción

Las arquitecturas paralelas han evolucionado en pos de obtener mejores tiempos de respuesta para las aplicaciones. En esta evolución pueden mencionarse los *clusters*, luego los *multicores*, y actualmente el nacimiento de las arquitecturas de *clusters* de *multicores*. Estas últimas consisten básicamente en una colección de procesadores *multicore* interconectados mediante una red.

Los *clusters* de *multicore* permiten combinar las características más distintivas de los *clusters* (la utilización de pasaje de mensajes en memoria distribuida) y de los *multicores* (como el uso de memoria compartida). Además los *clusters* de *multicores* introducen modificaciones en la jerarquía de memoria e incrementan aun más la capacidad y el poder de los sistemas computacionales.

Teniendo en cuenta el auge de esta arquitectura, es importante el estudio de nuevas técnicas para la programación de algoritmos paralelos que aprovechen eficientemente la potencia de la misma, considerando los sistemas híbridos en los que se combina memoria compartida y distribuida.

Como se mencionó anteriormente, un *cluster* de *multicore* consiste en una colección de procesadores *multicore* interconectados mediante una red, en la que trabajan cooperativamente como un único recurso de cómputo. Es decir, es similar a un *cluster* tradicional pero con la diferencia de que cada nodo posee un procesador *multicore* en lugar de un monoprocesador.

A la hora de implementar un algoritmo paralelo es muy importante considerar la jerarquía de memoria con la que se cuenta, ya que ello incidirá directamente en la *performance* alcanzable del mismo.

La *performance* de la jerarquía de memoria está determinada por dos parámetros de hardware: latencia de la memoria (tiempo entre que un dato es requerido y está disponible) y el ancho de banda de la misma (la velocidad con la que los datos son enviados de la memoria al procesador).

En la Fig. 1 se muestra un esquema de la jerarquía de memoria en las diferentes arquitecturas.

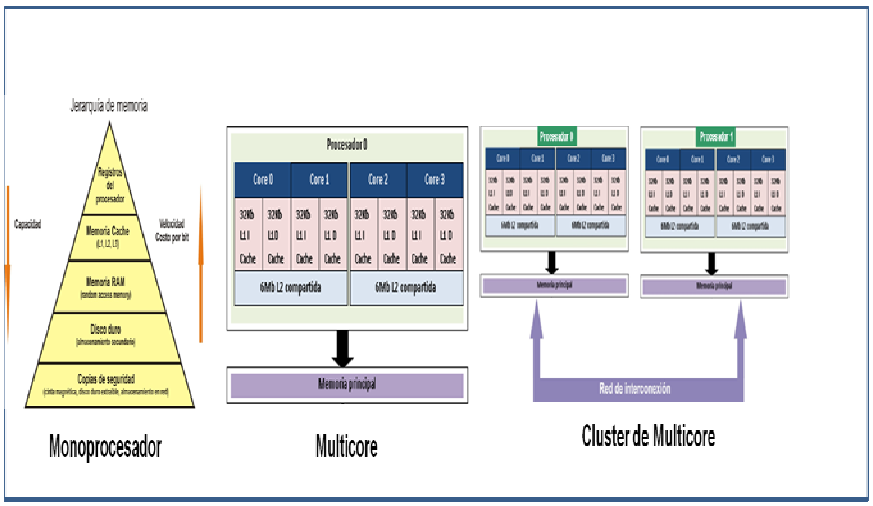

Fig. 1. Jerarquía de memoria en monoprocesador, *multicore* y *cluster* de *multicore*

Si se piensa en una arquitectura *multicore* existen, además de los niveles de registros y L1 propio de cada núcleo, dos niveles de memoria: la cache compartida de a pares de núcleos (L2) y la memoria compartida entre los *cores* de un procesador *multicore* [1].

En el caso de *clusters* tradicionales (homogéneos y heterogéneos) existen los niveles propios de cada procesador (registros del procesador y nivel L1 y L2 de cache) pero además se incluye un nuevo nivel: memoria distribuida a través de la red.

Particularmente, los *cluster* de *multicore* introducen un nivel más en la jerarquía de memoria tradicional. Además de la cache compartida entre pares de núcleos, y la memoria compartida entre todos los núcleos de un mismo procesador físico, se agrega la memoria distribuida accesible vía red.

Existe un conjunto de aplicaciones de naturaleza paralela y que por lo tanto pueden ejecutarse sobre esta arquitectura. Muchas de las mismas se encuentran dentro del

área del procesamiento digital de imágenes, incluyendo por ejemplo el tratamiento de imágenes médicas, satelitales o astronómicas. Un caso particular, utilizado en este trabajo, es BASIZ (*Bright and Saturated Image Zones*). Se trata de una aplicación de procesamiento de imágenes para la identificación y detección de las zonas con más brillo e intensidad de color de una imagen. Dada una imagen de entrada, el algoritmo lleva a cabo una serie de fases de procesamiento obteniendo como resultado la imagen original en la cual están marcadas las zonas más perceptibles al ojo humano, que son las zonas más brillantes [2][3].

Este artículo está organizado de la siguiente manera. En la Sección 2 se detalla la contribución del trabajo, mientras que la Sección 3 describe las características de la programación híbrida. En la Sección 4 se detalla el caso de estudio, mientras que en la Sección 5 se comentan las soluciones implementadas y la arquitectura utilizada para generar los resultados mostrados en la Sección 6. Por último, en la Sección 7 se exponen las conclusiones y líneas futuras

2 Contribución

La contribución principal es analizar la mejora en *performance* alcanzable en una arquitectura *cluster* de *multicore*, contrastando los modelos tradicionales de programación paralela (memoria compartida y distribuida) con la programación híbrida.

El análisis se realiza en base a tiempo de ejecución y *speedup* de la solución híbrida al crecer el tamaño del problema y en comparación con soluciones puramente de memoria compartida o de pasaje de mensajes.

3 Programación híbrida

Tradicionalmente el procesamiento paralelo se ha dividido en dos grandes modelos, el de memoria compartida y el de pasaje de mensajes [4].

- Memoria compartida: 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. 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 asociados a un procesador 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.

Relacionado con la aparición de la arquitectura *cluster* de *multicore* surge el modelo de programación híbrido, en el cual se combinan las estrategias recientemente expuestas. Esto implica una combinación de los modelos de programación paralela memoria compartida y pasaje de mensajes. 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) puede pensarse utilizando pasaje de mensajes.

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 gran interés actual.

Actualmente, entre los lenguajes que se utilizan para programación híbrida aparecen OpenMP para memoria compartida y MPI para pasaje de mensajes.

OpenMP es una interfase 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. Provee de una interfase 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 [5].

Por otro lado, MPI es una interfase de pasaje de mensajes creada para brindar portabilidad. 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 [6].

4 Caso de Estudio

La aplicación BASIZ está compuesta por siete pasos lógicos que pueden ser divididos en subtareas. Estos 7 pasos son [7]:

1. Separación en los tres colores básicos (rojo, verde y azul).
2. Difuminado de cada uno de los tres canales de color básico (*bluring*).
3. Suma de las imágenes difuminadas para cada canal de color.
4. Unión de los tres canales de color difuminados.
5. Conversión de formato de representación de la imagen difuminada de RGB a HSV (matiz, saturación y brillo).
6. Umbralización: obtener el umbral a partir del cual se determina la inclusión de los píxeles a un segmento sensitivo.
7. Detección y marcado de las zonas más sensitivas al ojo humano.

Para poder llevar a cabo los experimentos que permitan explotar la programación híbrida, se tomó sólo una parte de la aplicación BASIZ que posibilita la explotación del paralelismo de datos. Para ello se seleccionó el paso 2 (difuminado) para un canal de color.

El difuminado es un tipo de filtrado espacial, que es una operación que altera el valor de un píxel en función de los valores de los píxels que le rodean (procesamiento basado en la vecindad u operación de vecindad). El filtrado espacial se basa en aplicar la siguiente ecuación:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x+s, y+t)$$

donde f(x+s,y+t): valor de los pixels del bloque seleccionado, y w(s,t): coeficientes que se aplican al bloque (máscara), siendo la matriz del bloque: m(filas) = 2a + 1 y n(columnas) = 2b + 1.

5 Soluciones implementadas y Arquitectura utilizada

Los estudios experimentales fueron realizados en base a la implementación de la Fase 2 del algoritmo BASIZ, en forma secuencial y utilizando diferentes modelos de programación paralela: memoria compartida, pasaje de mensajes, e híbrido (combinación de los dos anteriores).

Tanto la solución secuencial como las paralelas, fueron desarrolladas utilizando el lenguaje C sin librerías destinadas al procesamiento de imágenes para poder tener pleno acceso al código que maneja las imágenes. Las soluciones paralelas que utilizan MPI como mecanismo de comunicación entre procesos, usan para ello la librería OpenMPI [6]. Las soluciones que utilizan memoria compartida para la programación paralela utilizan la librería OpenMP[5].

En esta primera fase de investigación, se realiza un análisis experimental del comportamiento de una aplicación híbrida, desde el punto de vista de los modelos de programación, en una arquitectura *cluster* de *multicore*.

Los resultados que se muestran se enfocan a analizar la solución híbrida en dos sentidos:

- El comportamiento al crecer el tamaño del problema (escalabilidad). En este caso, se procesaron secuencias de imágenes (entre 16 y 30), cada una de tamaño 4995 x 5113, 7104 x 7272 y 9472 x 9696 pixels.
- Comparar los tiempos de ejecución y *speedup* con los obtenidos en soluciones estrictamente de memoria compartida y de pasaje de mensajes. Aquí se utilizaron secuencias para tamaños de imágenes de 9472 x 9696 pixels.

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; 2 Gb de memoria RAM (compartida entre ambos procesadores); cache L2 de 2 X 6Mb compartida entre cada par de *cores* por procesador. El sistema operativo utilizado es Fedora 12 de 64 bits.

A continuación se describen los pseudocódigos de las soluciones implementadas.

5.1 Solución secuencial

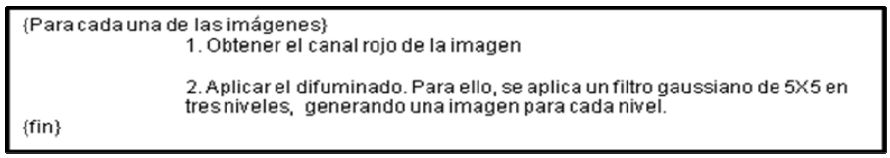

Fig. 2. Solución secuencial

5.2 Solución en memoria compartida

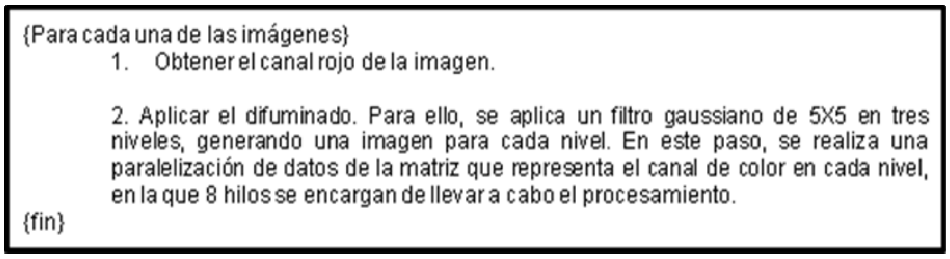

Fig. 3. Solución con memoria compartida

5.3 Solución con pasaje de mensajes

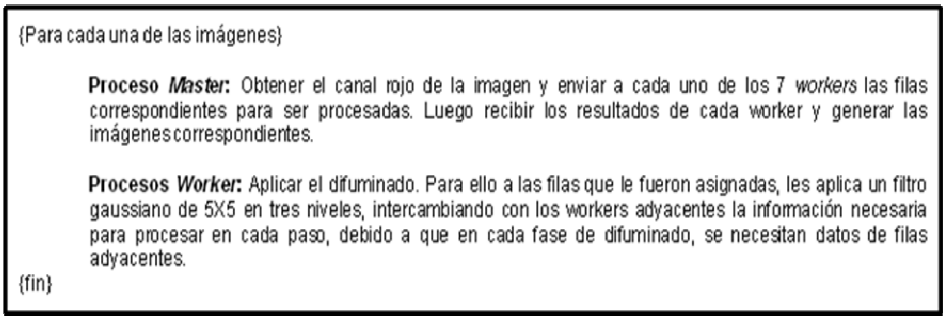

Fig. 4. Solución con pasaje de mensajes

5.4 Solución híbrida

{Para cada una de las imágenes}

Proceso 0: Obtener el canal rojo de la imagen y enviar al Proceso 1 la mitad de las filas para ser procesadas. Luego procesar la otra mitad de las filas, aplicando el filtro gaussiano, utilizando para ello 8 hilos. Finalmente recibir los resultados del Proceso 1 y junto a los generados localmente, generar las imágenes correspondientes.

Proceso 1: Aplicar el difuminado a las filas recibidas utilizando 8 hilos, intercambiando con el Proceso 0 la información necesaria para procesar en cada paso.

{fin}

Fig. 5. Solución híbrida

6 Resultados obtenidos

A continuación se muestran los resultados obtenidos en las pruebas experimentales realizadas. En la Tabla 1 se puede ver los tiempos de ejecución de la solución híbrida para diferente tamaños de la secuencia de imágenes (de 16 a 30) donde en cada secuencia el tamaño de las imágenes también escala.

N° de imágenes	4996x5113 (a)	7104x7272 (b)	9472x9696 (c)
16	440,27	860,29	1429,58
18	498,70	928,86	1579,89
20	548,13	1077,99	1755,82
22	603,28	1187,32	1930,21
24	658,00	1335,33	2111,51
26	725,38	1400,33	2280,48
28	775,75	1526,13	2451,83
30	821,67	1619,35	2634,55

Tabla 1. Tiempos de ejecución (en segundos) .

Puede observarse que al crecer el tamaño del problema, el tiempo de ejecución en esta solución híbrida crece en proporción menor o igual que el mismo. A modo de ejemplo, para secuencias de 20 imágenes el problema crece 2.02 veces de (b) a (a) y el tiempo crece 1.96 veces, y de (c) a (b) el tamaño crece 1.77 veces mientras el tiempo crece 1,62 veces.

En la Tabla 2 se muestran los tiempos de ejecución de cada una de las soluciones implementadas (memoria compartida, pasaje de mensajes e híbrida) para una secuencia de imagenes que varia entre 16 y 30 donde cada imagen es de 9472x9696 pixels.

N° de imágenes	Secuencial	Memoria compartida	Pasaje de mensajes	Híbrida
16	1946,15	1455,48	6455,81	1429,58
18	2186,46	1637,99	12823,26	1579,89
20	2432,10	1812,44	8931,06	1755,82
22	2676,64	1993,78	14327,46	1930,21
24	2920,19	2175,62	10260,59	2111,51
26	3160,48	2355,63	13362,11	2280,48
28	3405,58	2535,32	10113,91	2451,83
30	3657,90	2719,22	11123,98	2634,55

Tabla 2. Tiempos de ejecución (en segundos) .

La Fig. 6 muestra un gráfico comparativo del *speedup* para los resultados mostrados en la Tabla 2.

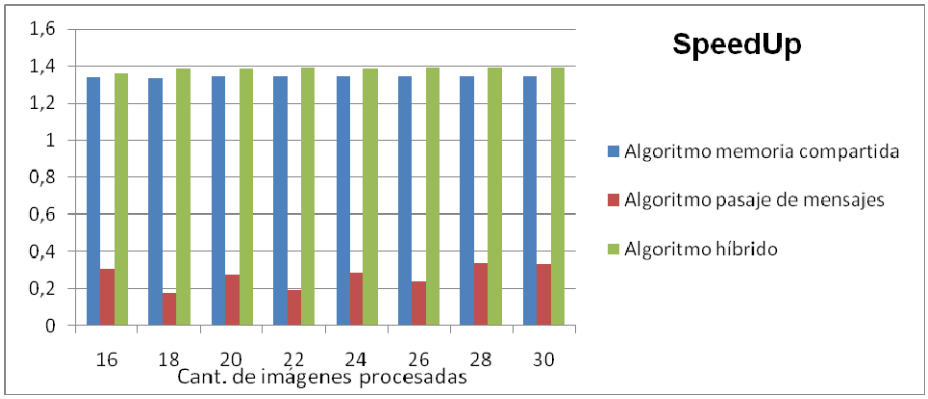

Fig. 6. Comparación de *speedup*

7 Conclusiones y Trabajos Futuros

En referencia a la escalabilidad, los resultados obtenidos muestran que la solución híbrida es escalable y que el aumento del tamaño del problema lleva a tiempos de ejecución que en porcentaje son menores o iguales a dicho incremento.

Por otro lado, la comparación entre las soluciones de memoria compartida y pasaje de mensajes con respecto a la híbrida permite ver que esta última es la que da como resultado mejores tiempos de ejecución. Esto se refleja además en el *speedup* obtenido.

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

Como trabajos futuros, se estudiará el comportamiento para tamaños aún más grandes de imágenes y de otras estrategias de paralelización (por ejemplo la distribución dinámica del volumen de trabajo apuntando al uso de *cores* heterogéneos).

8 Referencias

1. Burger T. “Intel Multi-Core Processors: Quick Reference Guide ”http://cachewww.intel.com/cd/00/00/23/19/231912_231912.pdf.
2. Roig C., Ripoll A., Borrás J., Luque E. “Efficient Mapping for Message-Passing.Applications Using the TTIG Model: A Case Study in Image Processing.”(2001).
3. Roig C. “Algoritmos de asignación basados en un nuevo modelo de representación de programas paralelos” Tesis Doctoral – 2005 Unidad Autónoma de Barcelona – España.
4. Dongarra J. , Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. “Sourcebook of Parallel computing”. Morgan Kaufmann Publishers – ISBN 1 55860 871 0 (Capítulo 3).
5. https://computing.llnl.gov/tutorials/openMP (2010)
6. http://www.open-mpi.org (2010)
7. Roig C., Ripoll A., Senar M. A., Guirado F., Luque E. “A New Model for Static Mapping of Parallel Applications with Task and Data Parallelism”.(2002)