Comparación de paradigmas de programación paralela en *cluster* de *multicores*: Pasaje de mensajes e híbrido. Un caso de estudio.

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

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, sgallo, ldgiusti, francoch, mnaiouf, degiusti}@lidi.info.unlp.edu.ar

Abstract. La programación híbrida ha alcanzado importancia a raíz de la aparición de las arquitecturas *cluster* de *multicores*, fruto del avance tecnológico de los procesadores y las limitaciones físicas que imponían las arquitecturas tradicionales. Este nuevo paradigma de programación permite aprovechar la nueva jerarquía de memoria que la arquitectura provee.
El objetivo de este trabajo es realizar un análisis comparativo de dos paradigmas de programación paralela, el paradigma de pasaje de mensajes, (utilizado en los *clusters* tradicionales) y la programación híbrida (donde se combina pasaje de mensajes y memoria compartida). La experimentación se realiza sobre el problema clásico de multiplicación de matrices.
La arquitectura de prueba utilizada en el análisis experimental es un *cluster* de *multicores*.

Keywords: arquitecturas paralelas, programación híbrida, cluster, multicore, pasaje de mensajes, memoria compartida.

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 las arquitecturas de *clusters* de *multicores* [1].

Un *cluster* de *multicores* consiste en un conjunto 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 cada nodo posee al menos un procesador con múltiples núcleos en lugar de un monoprocesador. Esto permite combinar las características más distintivas de la arquitectura de memoria distribuida de los *cluster* con la memoria compartida de los *multicores*.

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 [2].

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

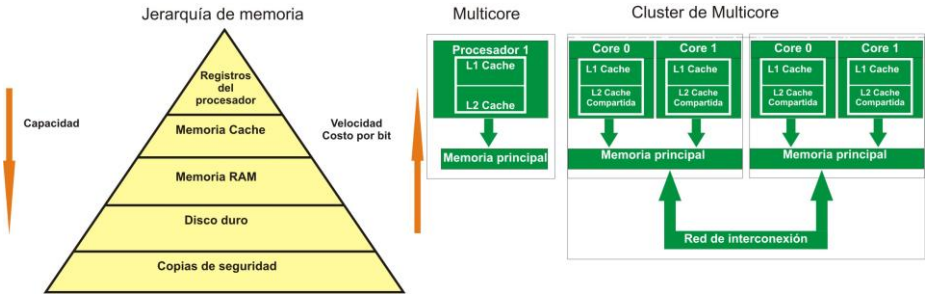

Fig. 1. Jerarquía de memoria

En el caso de *clusters* tradicionales (homogéneos y heterogéneos) existen los niveles de memoria 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.

Si se piensa en una arquitectura *multicore* además de los niveles de registros y L1 propio de cada núcleo, existen dos niveles de memoria: la *cache* compartida de a pares de núcleos (L2) y la memoria compartida entre los *cores* de un procesador *multicore* [3].

En el caso de los *clusters* de *multicores*, se introduce un nivel más en la jerarquía de memoria al agregar la memoria distribuida accesible vía red.

Existe un gran número de aplicaciones paralelas en diferentes áreas. Una de las más tradicionales y ampliamente estudiada en el cómputo paralelo (dado que constituye la base de otras) es la multiplicación de matrices, la cual se utiliza en este trabajo. El motivo por el cual se utiliza esta aplicación (ampliamente probada y evaluada) es que la misma permite utilizar y explotar el paralelismo de datos, y por otro lado la facilidad del mismo para escalar el problema al aumentar el tamaño de las matrices [4]. Este problema es de los conocidos como embarazosamente paralelos. En esta aplicación los problemas de sincronización son mínimos y nos permiten enfocar directamente en el consumo de memoria, que es una limitación importante del problema elegido. De esta forma, pueden compararse las soluciones que utilizan pasaje de mensajes con aquellas que utilizan memoria compartida y pasaje de mensajes (híbrida). Este articulo es un avance de una investigación previa, en la que se utilizó este mismo caso de estudio comparando la programación híbrida con la de pasaje de mensajes pero con una estrategia de implementación en la que el consumo

de memoria en ambas soluciones no permitía escalar el problema para tamaños de matrices mayores a 8192 X 8192 [5].

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 muestran 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 de investigación futuras.

2 Contribución

El objetivo de la investigación es realizar un análisis comparativo contrastando la *performance* alcanzable con la programación híbrida en una arquitectura *cluster* de *multicores*, con la alcanzable en un modelo tradicional de programación paralela (memoria distribuida), analizando también cómo impacta en un algoritmo el aprovechamiento de las jerarquías de memoria existentes en la arquitectura donde se ejecutan.

El análisis se realiza en base al tiempo de ejecución y eficiencia de la solución híbrida al escalar el tamaño del problema y la cantidad de núcleos utilizados y en comparación con soluciones que utilizan sólo pasaje de mensajes.

3 Programación Híbrida

Tradicionalmente el procesamiento paralelo se ha dividido en dos grandes modelos, el de memoria compartida y pasaje de mensajes [2][6].

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.

La arquitectura *cluster* de *multicores* permite introducir un nuevo modelo de programación paralela como es la programación híbrida. En la misma se combinan las estrategias recientemente expuestas. 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.

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 *Pthreads* [7] para memoria compartida y *MPI* [8] para pasaje de mensajes.

4 Caso de estudio

Dadas dos matrices A de *n x p* y B de *p x n* elementos, la multiplicación de ambas matrices 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.

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

5 Soluciones Implementadas y Arquitectura Utilizada

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 diferentes modelos de programación paralela: 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* [8]. La solución híbrida utiliza la librería *Pthreads* [7] para memoria compartida junto a *OpenMPI* para el pasaje de mensajes.

En la investigación realizada en este artículo, 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 *multicores* [9][10][11].

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

1. El comportamiento al incrementar el tamaño del problema y la cantidad de núcleos (escalabilidad) [9][10]. En este caso, se procesaron matrices cuadradas de 1024, 2048, 4096, 8192, 16384 y 28672 filas y columnas.
2. Comparar los tiempos de ejecución y eficiencia con los obtenidos en la solución de pasaje de mensajes.

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. El sistema operativo utilizado es *Fedora* 12 de 64 bits [12][13].

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.

5.1 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.

5.2 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*. 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 de procesamiento 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.

5.3 Solución híbrida

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*.

Podemos resumir el algoritmo de la siguiente manera:

```
Proceso master:
  Divide la matriz C en bloques.
  Para cada fase:
    Comunica las filas correspondientes de la matriz A
    y las columnas correspondientes de la matriz B a
    los procesos worker, según el bloque que va a
    procesar cada uno.
    Genera los hilos y procesa el bloque que le
    corresponde.
    Recibe los resultados de los procesos worker.
Procesos worker
  Para cada fase:
    Reciben los datos a procesar
    Generan los hilos y procesan los datos.
    Comunican los resultados al proceso master.
```

6 Resultados Obtenidos

A continuación se muestran los resultados obtenidos en las pruebas experimentales realizadas. 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 1 se muestran los tiempos de ejecución de la solución secuencial (Sec.). En la Tabla 2, los tiempos obtenidos por el algoritmo de pasaje de mensajes utilizando 16 y 32 núcleos (PM 16 y PM 32) y el tamaño de los bloques (TBPM16 y TBPM32) y los obtenidos por la solución híbrida con 16 y 32 núcleos (H16 y H32) y el tamaño de los bloques (TBH16 y TBH32). 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. En la Figura 2 se muestran los *speedups* obtenidos para las pruebas mencionadas.

Tabla 1. Tiempos de ejecución secuencial

Tam. Matriz	Secuencial (seg.)
1024 * 1024	7,68
2048 *2048	62,44
4096 * 4096	498,43
8192 * 8192	4019,53
16384 * 16384	32138,45
28672 * 28672	172343,27

Tabla 2. Tiempos de ejecución soluciones paralelas

Tam.	TBPM 16	PM 16	TBPM 32	PM 32	TBH16	H 16	TBH32	H 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
28672	512	14433,93	512	8019,19	2048	13026,13	2048	6480,14

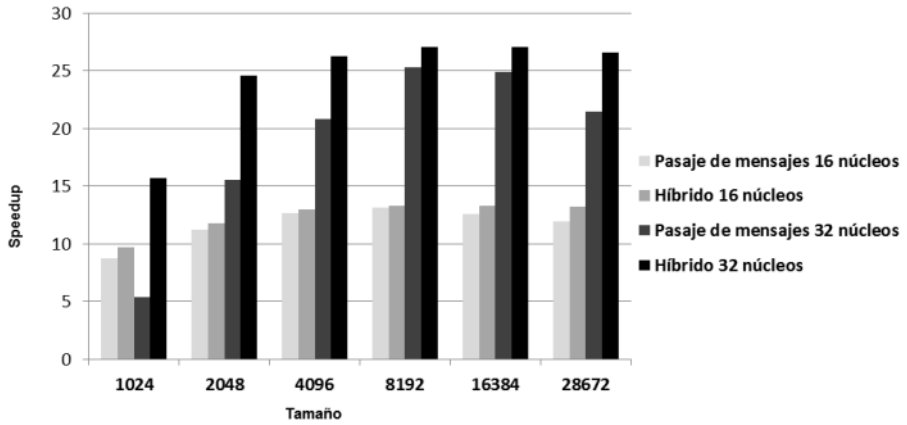

Fig. 2. *Speedup*

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.

6.1 Resultados Comparados

En la Tabla 3 se muestra la eficiencia alcanzada por las diferentes alternativas de prueba, mientras que en la Figura 3 puede verse un gráfico comparativo que muestra dicha información.

Tabla 3. Eficiencia

Tam.	PM16	H16	PM32	H32
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
28672	0,74	0,82	0,67	0,83

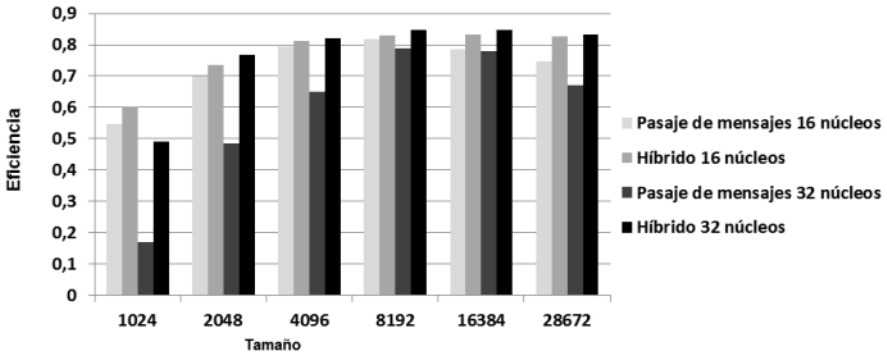

Fig. 3. Eficiencia

Los resultados obtenidos 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, debemos 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. Esto se debe a que no es necesario aumentar la cantidad de bloques a procesar cuando pasamos de 16 a 32 unidades de procesamiento. Entonces al tener más unidades de cómputo para la misma cantidad de bloques, disminuimos, 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, tenemos 2 procesos que internamente mediante la generación de hilos, utilizan 8 núcleos cada uno y que para 32, tenemos 4 procesos con las mismas características que los anteriores.

Por otro lado, hay que destacar que la eficiencia alcanzada por la solución de pasaje de mensajes para 16384 * 16384 y para 28672 * 28672 elementos con 16 y 32 núcleos, se ve degradada frente a los demás tamaños para la misma cantidad de núcleos. Esto se debe a que las limitaciones en la memoria principal disponible en cada hoja, provoca que sea necesario aumentar la cantidad de bloques de procesamiento, dado que es necesario que los mismo 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. Lo mismo ocurre para el caso híbrido con 28672 * 28672 elementos.

7 Conclusiones y Líneas de Investigación Futuras

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, debemos tener mayor cantidad de fases de sincronización que en la solución híbrida, esto provoca que el tiempo de ejecución y la eficiencia se ven degradados notablemente, ya que se necesitan más fases de comunicación y sincronización que en el caso híbrido, por lo que se genera más *overhead*.

Las líneas de investigación futuras incluyen el análisis de la eficiencia energética de los algoritmos que utilizan el paradigma de programación híbrida y el estudio de optimización de algoritmos híbridos sobre *clusters* heterogéneos [14][15].

8 Referencias

1. 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.

2. 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).
3. Burger T. “Intel Multi-Core Processors: Quick Reference Guide ”http:// cachewww.intel.com/cd/00/00/23/19/231912_231912.pdf. (2010).
4. Andrews G. “Foundations of Multithreaded, Parallel and Distributed Programming”. Addison Wesley Higher Education 2000. ISBN-13: 9780201357523 .
5. Leibovich, F., De Giusti, L., Naiouf, M. “Parallel Algorithms on Clusters of Multicores: Comparing Message Passing vs Hybrid Programming”. Julio 2011. WorldComp’11.
6. Grama A., Gupta A., Karpyis G., Kumar V. “Introduction to Parallel Computing”. Pearson – Addison Wesley 2003. ISBN: 0201648652. Segunda Edición (Capítulo 3).
7. https://computing.llnl.gov/tutorials/pthreads (2010)
8. http://www.open-mpi.org (2010)
9. 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.
10. Leopold C., “Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches”. Wiley, 2001. ISBN: 0471358312(Capítulos 1, 2 y 3).
11. 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.
12. HP, "HP BladeSystem". http://h18004.www1.hp.com/products/blades/components/c-class.html. (2011).
13. HP, "HP BladeSystem c-Class architecture". http://h20000. www2.hp.com/bc/docs/support/SupportManual/c00810839/c00810839.pdf (2011).
14. Feng, W.C., “The importance of being low power in high-performance computing”. Cyberinfrastructure Technology Watch Quarterly (CTWatch Quarterly). 2005.
15. Balladini J., Grosclaude E., Hanzich M., Suppi R., Rexachs D., Luque E., “Incidencia de los modelos de programación paralela y escalado de frecuencia de CPUs en el consumo energético de los sistemas de HPC”. XVI Congreso Argentino de Ciencias de la Computación, pp. 172-181. 2010.