Análisis de soluciones paralelas puras e híbridas en un problema de simulación.

Silvana Lis Gallo[1,2], Franco Chichizola[1], Laura De Giusti[1], Marcelo Naiouf[1]

[1] Instituto de Investigación en Informática LIDI (III-LIDI),
Facultad de Informática,
Universidad Nacional de La Plata,
50 y 120 2do piso, La Plata, Argentina.
[2] Becaria CONICET.

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

Resumen. Más allá de las mejoras constantes en las arquitecturas físicas, uno de los mayores desafíos se centra en cómo aprovechar al máximo su potencia. En los últimos años, uno de los cambios de mayor impacto ha sido el uso de manera masiva de procesadores con más de un núcleo (multinúcleo o *multicore*). La combinación de varios de estos procesadores ha producido plataformas híbridas (con memoria compartida y distribuida), llevando a la necesidad de desarrollar sistemas operativos, lenguajes y algoritmos que las usen adecuadamente. En el presente artículo el objetivo se centra en realizar un análisis comparativo de los diferentes modelos de programación paralela existentes, aplicados a problemas de simulaciones discretas. Se presenta como caso de estudio y experimentación la solución al problema Sharks and Fishes, analizando eficiencia y escalabilidad de los algoritmos implementados.

Palabras Clave: Algoritmos paralelos y distribuidos. Multicore. Cluster de multicore. Evaluación de performance. Simulación.

1 Introducción

Reducir el tiempo de ejecución de aplicaciones con grandes requerimientos de procesamiento es el objetivo principal del procesamiento paralelo. Esto se debe al constante incremento del volumen de procesamiento y las limitaciones que impone el cómputo secuencial en cuanto a tiempos de respuesta, acceso a datos distribuidos y manejo de la concurrencia implícita en los problemas del mundo real [1][2].

El procesamiento paralelo implica la existencia de múltiples procesadores y el tipo y organización de los mismos dentro de las arquitecturas de soporte, influye en el diseño de las aplicaciones para cumplir con el objetivo propuesto.

A raíz del crecimiento de los límites de generación de calor y consumo de energía alcanzados por los procesadores, surge como alternativa la integración de dos o más núcleos computaciones dentro de un mismo chip, lo cual se conoce como procesador

multicore o multinúcleo. Los procesadores multicore pueden mejorar el rendimiento de una aplicación si se distribuye el trabajo entre los núcleos disponibles [3].

La aparición de la tecnología multicore ha impactado sobre los clusters (colección de computadoras individuales interconectadas que trabajan en conjunto como un único recurso integrado de computación), introduciéndolos en una nueva etapa. Un cluster de multicores es similar a un cluster tradicional, sólo que en lugar de monoprocesadores está formado por procesadores multicore, agregando un nivel más de comunicación [4] [5]. Además de la caché compartida entre pares de núcleos y la memoria compartida entre todos los núcleos de un mismo procesador, se cuenta con la memoria distribuida accesible vía red (Fig. 1).

A la hora de escribir un algoritmo paralelo en esta arquitectura, no se puede obviar la jerarquía de memoria presente, ya que incide sobre el rendimiento alcanzable por el mismo. Por el contrario, se deben estudiar nuevas técnicas de programación de algoritmos paralelos que permitan aprovechar de manera eficiente la potencia de estas arquitecturas [6].

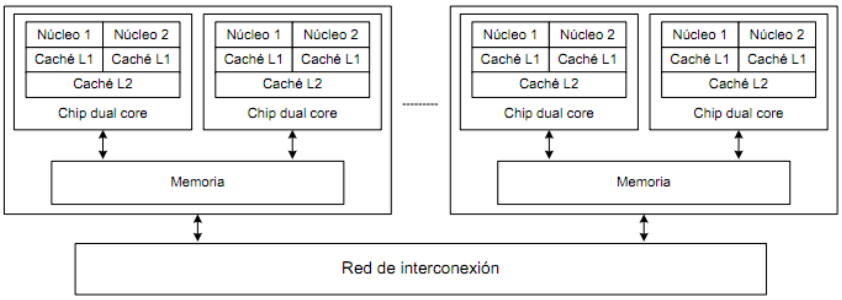

Fig. 1. Cluster de multicores

Diversos lenguajes de programación y librerías han sido desarrollados para la programación paralela explícita. Estas difieren principalmente en la manera en que el usuario ve al espacio de direcciones. Los modelos se dividen básicamente en los que proveen un espacio de direcciones compartido o uno distribuido, aunque también existen modelos híbridos que combinan las características de ambos. A continuación se detallan los diferentes modelos [7].

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. Se caracteriza por la necesidad de la sincronización para preservar la integridad de las estructuras de datos compartidas [8].

Pasaje de mensajes: los datos son vistos como 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 enviarlo y el procesador que lo requiere debe recibirlo, y el dato viaja a través de un canal. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización [8]. Las variantes en el envío y recepción (sincrónica o asincrónica), y las características

de los canales de comunicación (uni o bidireccionales), dan lugar a los diferentes mecanismos de comunicación.

Debido al avance de las arquitecturas paralelas y especialmente a la aparición de la arquitectura de múltiples núcleos (multicore), surge el modelo de programación **híbrido** en el cual se combinan las estrategias recientemente expuestas. Por ejemplo, en un cluster de multicores, puede utilizarse el modelo de memoria compartida para los procesadores lógicos dentro de cada procesador y el de pasaje de mensajes para la comunicación entre los procesadores físicos.

Este trabajo presenta diferentes soluciones paralelas (puras e híbrida) a un problema de simulación, analizando el comportamiento en términos de eficiencia y escalabilidad.

2 Descripción del problema de aplicación

El problema seleccionado para la experimentación es Sharks and Fishes. El mismo consiste en la simulación de la evolución de una población de tiburones y peces dentro del océano, siguiendo diferentes reglas. Este problema es representativo de una clase de sistemas biológicos que permite estudiar el comportamiento de diferentes poblaciones evolucionando en el tiempo.

En este problema, el océano se encuentra dividido en una grilla de tamaño NxN, donde el lado derecho de la misma se encuentra conectado con el izquierdo, y a su vez el lado superior con el inferior (es un espacio toroidal). Las diferentes celdas que constituyen la grilla se encuentran vacías u ocupadas por un pez, un tiburón o por plancton. La población evoluciona con pasos de tiempo discretos de acuerdo a reglas establecidas para los individuos.

2.1 Reglas para la población de peces

En cada paso de tiempo, el pez intenta moverse a una celda vecina que se encuentre con plancton (para alimentarse), en caso de hallarlo, pasa a ocupar esa celda y su energía aumenta según un parámetro de la simulación. Si sus celdas vecinas se encuentran ocupadas (por otros peces o tiburones), se queda en la posición anterior.

Un pez no puede reproducirse si no puede moverse. Si un pez alcanza la edad de reproducción ante un nuevo paso de tiempo, se reproduce dejando una cría con edad 0 en la celda anterior del movimiento y con la mitad de la energía que tenía el pez inicial. Los peces mueren cuando ya no tienen energía suficiente (la energía se reduce con el paso del tiempo).

2.2 Reglas para la población de tiburones

En cada paso, si en alguna de las celdas vecinas se encuentra un pez, el tiburón se mueve para poder comerlo, pasando a ocupar la celda donde se encontraba el pez y ganado una cantidad determinada de energía. Si no hay ningún pez a su alrededor, pero se encuentra alguna celda vacía (agua), el tiburón se mueve hacia ella. En caso

contrario, no se mueve. Los tiburones únicamente comen peces. Cuando ya no tiene energía suficiente, el tiburón muere (la energía se reduce al pasar el tiempo).

3 Descripción de la solución secuencial implementada

Para la resolución del problema en forma tradicional o secuencial, se utilizó un autómata celular. Se representó al océano mediante una matriz de estructuras o registros que mantiene el estado de cada posición de la misma.

Los pasos que sigue el algoritmo son simples. En primer lugar se inicializa la matriz, con las cantidades de tiburones y peces necesarias y de acuerdo a las constantes definidas para la simulación. Luego, se realiza el recorrido de la matriz por cada paso de tiempo, invocando a una función que analiza y elige aleatoriamente el comportamiento a seguir por el pez o tiburón, dependiendo del estado de las celdas adyacentes y de la acción que se desee realizar. Dicha función verifica las ocho celdas que rodean a la posición actual, y de todos los movimientos posibles para la acción a realizar, elige uno aleatoriamente.

4 Descripción de las soluciones paralelas implementadas

4.1 Solución de memoria compartida

En esta sección, se especifica el mecanismo utilizado en la implementación para el modelo de memoria compartida por medio de la herramienta OpenMP.

Para llevar a cabo la solución, se realizó una división de la matriz en strips (o grupos de filas) horizontales de igual tamaño para cada thread. Como consecuencia de este tipo de división, pueden suceder conflictos en los límites de los fragmentos, ya que se tendrían dos hilos trabajando en celdas contiguas y el funcionamiento original del algoritmo en una celda en particular necesita interactuar con las ocho celdas adyacentes.

Para evitar dichos conflictos, se realizó la paralelización de los recorridos de los diferentes fragmentos de matriz de la siguiente manera. Se distinguen tres zonas en cada fragmento de matriz.

Por un lado, la parte superior, constituida por los N elementos que son "compartidos" con el thread anterior al actual. Esto significa que si el identificador del thread actual es i, el thread con el cual se comparte dicha zona es el hilo i-1 ó, si la cantidad de threads es th y se trata del primer thread (al cual corresponde el identificador 0), se compartirán los datos con el thread identificado con el valor th-1. Luego, se tiene una parte central en la cual se encuentran las celdas o elementos que son propias del thread. Estas serán únicamente modificadas por el hilo al cual fueron asignadas. Por último, la parte inferior del fragmento contiene, al igual que la parte superior, celdas compartidas, pero con el thread siguiente al actual (el hilo con identificador i+1 o 0 si se trata del thread th-1).

Por lo tanto, se tendrán tantos fragmentos como threads se creen y se irá alternando el trabajo de los mismos en las diferentes zonas. Para un paso de tiempo dado de la

simulación se producirán tres instancias de procesamiento. Inicialmente, cada thread trabaja con la parte superior de su zona, reflejando, en caso de ser necesario, modificaciones en la parte inferior del hilo anterior. Más tarde, en una segunda fase, se procesa la parte central de cada submatriz por el thread correspondiente. Finalmente, se procesa la parte inferior teniendo la posibilidad de alterar la parte superior del thread siguiente. Este mecanismo se repite por cada paso de tiempo de la simulación permitiendo la ejecución a lo largo de toda la matriz. De manera que, en cada fragmento de la grilla y en un mismo instante de tiempo, un único hilo puede estar realizando modificaciones.

4.2 Solución de Pasaje de Mensajes

Esta sección introduce una alternativa de implementación que plantea el uso de pasaje de mensajes para la resolución del problema. En este caso, la solución implementada divide la matriz en strips horizontales (al igual que las soluciones de memoria compartida), teniendo en cuenta el modo de almacenamiento de la misma. Se aprovecha el espacio de almacenamiento horizontal contiguo para no tener demoras en el armado de estructuras de datos para el envío.

En cuanto a la asignación de fragmentos a procesos, se determinó que corresponder uno a cada proceso sería una buena alternativa, ya que si se opera con un único fragmento a lo largo de toda la simulación solo se necesita intercambiar mensajes para la actualización de los límites.

Como se mencionó anteriormente, existe la posibilidad de que sucedan conflictos en los límites de los segmentos de matriz entre procesos. Para su solución se utilizó la sincronización entre los mismos, realizando el envío de la información a través de pasaje de mensajes (librería MPI). La operatoria del algoritmo se lleva a cabo mediante el uso de un esquema de comunicación circular de procesos que comunica mensajes en forma bidireccional. Según la cantidad de procesadores disponibles, se divide la matriz de manera uniforme entre los mismos.

Se tiene 2 tipos de procesos; el primero o proceso 0 reparte la matriz inicial, ejecuta la simulación en su fragmento de matriz y recopila el resultado final (matriz obtenida al final del algoritmo). Por otro lado, los demás procesos reciben su fracción de la matriz y mediante mensajes se comunican con los vecinos, enviando los fragmentos necesarios para operar en los límites de las porciones.

En una primera etapa, se procesa el fragmento superior y el segmento independiente de la submatriz de cada proceso. Posteriormente, se realiza el envío y recepción de los fragmentos compartidos y se continúa con el procesamiento hasta finalizar con el recorrido de la última fracción compartida. Aquí se produce un nuevo intercambio de límites y se continúa con los sucesivos pasos de simulación.

4.3 Solución híbrida

Con el propósito de lograr un mejor aprovechamiento de la arquitectura y mejorar las prestaciones, se propone una solución que combina el modelo de memoria compartida con pasaje de mensajes. Esta solución sigue un esquema donde se combina MPI con OpenMP.

Para su implementación, se evaluó el modo de distribución de los datos y la comunicación necesaria entre los procesos e hilos. Se optó por realizar un particionamiento de los datos al igual que en el algoritmo MPI, por lo que a cada proceso le corresponde un bloque de filas de la matriz original. Además, para que los hilos dentro de cada uno de los procesos puedan cooperar para completar la simulación, se repitió el esquema de división de datos de la solución de memoria compartida correspondiendo a cada hilo una submatriz, sólo que en este caso se le asignará un fragmento correspondiente a la porción del proceso.

Cuando se trabaja con algoritmos híbridos, la comunicación se lleva a cabo entre los hilos de los diferentes nodos de la arquitectura. Como se observa en la Fig. 2., se tiene un grupo de procesos, donde cada uno de ellos se comunica con su antecesor y sucesor, y donde el proceso 0 comunica los datos iniciales mediante sentencias de comunicación colectivas, procesa y recibe el resultado final. La comunicación entre procesos se realiza para el intercambio de las filas limítrofes correspondientes, tal como sucede con el algoritmo MPI.

Dentro de cada proceso existe un conjunto de threads, los cuales resuelven los pasos de tiempo de la simulación para la submatriz del proceso que integran (ver Fig. 2.). Debido al modelo híbrido de comunicación, son necesarios dos tipos de threads para esta solución.

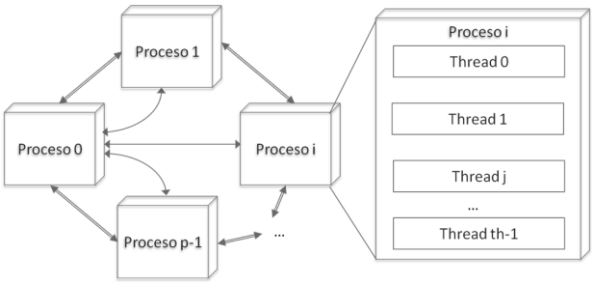

Fig. 2. Estructura de la solución híbrida

Por un lado, se encuentran los "threads de frontera" que permiten la comunicación con los procesos adyacentes al proceso al cual pertenecen. Entre ellos se encuentran los hilos iniciales, que se encargan de sincronizar el cómputo con los demás hilos de su mismo proceso pero, a su vez, se comunican mediante pasaje de mensajes con el último hilo del proceso anterior (en el caso del proceso 0, aquel con el cual se establece la comunicación es el proceso p-1). También se encuentran en la frontera los hilos finales, que interactúan con el hilo inicial del proceso siguiente para realizar el envío y recepción de los límites de la matriz (en el caso del proceso p-1, se interactuará con el proceso 0) y, además, alternan el procesamiento de su área con los demás hilos del mismo proceso.

Como segundo tipo, surgen los "threads intermedios", que únicamente interactúan con los hilos de su mismo proceso (en la Fig. 2, los threads 1 y j). De la misma manera que en la solución de memoria compartida llevan a cabo la simulación alternando el procesamiento en las zonas correspondientes.

En una primera etapa, el proceso 0 realiza la distribución de los datos iniciales y los demás procesos reciben dichos datos. Luego, cada proceso crea e inicializa las

estructuras necesarias para los hilos o threads que llevarán a cabo la simulación. En otra nueva fase, los threads trabajan para completar los pasos de tiempo comunicándose y sincronizando unos con otros. Por último, se destruyen los hilos generados y se recopilan los resultados de la simulación.

5 Pruebas realizadas

Para la experimentación se utilizó un Blade de 8 hojas, con 2 procesadores quad core Intel Xeón e5405 de 2.0 GHz en cada una de ellas. Cada hoja posee 2Gb de RAM (compartido entre ambos procesadores) y cache L2 de 2 x 6Mb entre par de núcleos.

Para analizar el comportamiento de cada solución paralela implementada en este trabajo, se midió tiempo de ejecución, speedup, y eficiencia de las mismas. También se analiza la escalabilidad desde tres puntos de vista: el primero variando el tamaño del océano (1024x1024; 2048x2048; 4096x4096; 8192x8192), el segundo modificando el lapso de tiempo de la simulación (1024; 2048; 4096; 8192 momentos), y el último utilizando diferentes cantidades de procesos y/o hilos (en total 4, 8, 16 y 32 cores). Las combinaciones mencionadas anteriormente se agrupan en 16 escenarios de prueba los cuales se encuentran detallados en la Tabla 1.

Tabla 1. Escenarios de prueba.

Escenario	Momento	Tamaño del océano
1	1024	1024
2	1024	2048
3	1024	4096
4	1024	8192
5	2048	1024
6	2048	2048
7	2048	4096
8	2048	8192
9	4096	1024
10	4096	2048
11	4096	4096
12	4096	8192
13	8192	1024
14	8192	2048
15	8192	4096
16	8192	8192

5.1 Soluciones puras

En esta Sección se realiza una comparación de la eficiencia obtenida por las soluciones detalladas en las Secciones 4.1 (OpenMP) y 4.2 (MPI). En la Fig. 3. se muestra las diferencia de eficiencia en los 16 escenarios y utilizando 4 hilos/procesos dependiendo el caso. A su vez en la Fig. 4. se presenta la misma comparación pero utilizando 8 hilos/procesos.

En el gráfico de la Fig. 3 se observa que cuando la cantidad de hilos/procesos no es grande, en general la eficiencia lograda por ambos algoritmos es similar, siendo levemente superior (en la mayoría de los casos) la de OpenMP.

Además, al crecer el problema (tamaño del océano y lapso de tiempo de la simulación) más se nota la aproximación de los resultados de MPI a los de OpenMP, y en algunos casos incluso la superioridad.

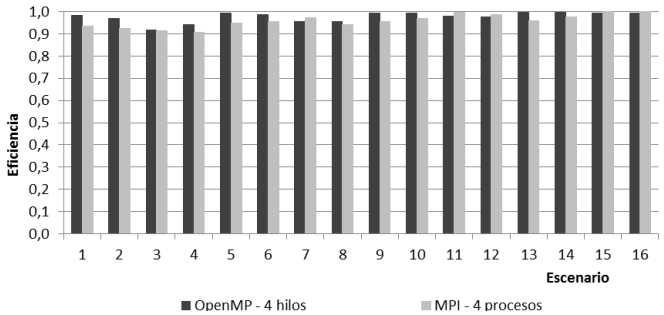

Fig. 3. Eficiencia obtenida en las diferentes soluciones puras para 4 threads/procesos.

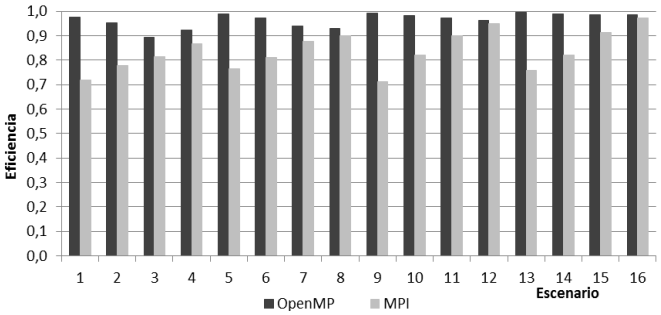

Fig. 4. Eficiencia obtenida en las diferentes soluciones puras para 8 threads/procesos.

Al trabajar con mayor cantidad de hilos/procesos, se puede notar que la diferencia de eficiencia lograda por los algoritmos es más notoria a favor de OpenMP, como se ve en la Fig. 4. En todos los escenarios este algoritmo se comporta mejor que el de MPI.

Teniendo en cuenta estos resultados (sobre todo al usar mayor cantidad de hilos/procesos) donde es más eficiente la solución con OpenMP, y la imposibilidad de poder ejecutar esta solución en más de una hoja de la arquitectura, cobra relevancia a la posibilidad de mejorar el rendimiento utilizando un algoritmo híbrido. De tal manera que aproveche la mejor eficiencia de la solución con OpenMP dentro de cada multicore, y la posibilidad de comunicación entre ellos por medio de MPI.

5.2 Soluciones para una arquitectura híbrida

En esta Sección se realiza una comparación de la eficiencia obtenida por las soluciones que pueden ser usadas en una arquitectura hibrida (cluster de multicores). En este caso la solución MPI (detalladas en las Secciones 4.2) y la híbrida que combina MPI+OpenMP (detallada en la sección 4.3).

En la Fig. 5. se muestran gráficamente las eficiencias logradas por ambos algoritmos en los 16 escenarios y utilizando 16 cores. A su vez en la Fig. 6. se presenta la misma comparación pero utilizando 32 cores.

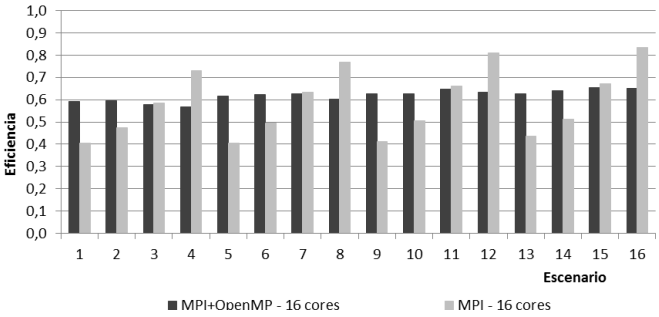

Fig. 5. Eficiencia obtenida en la solución hibrida y pasaje de mensajes para 16 cores.

Observando el gráfico de la Fig. 5., se puede observar que el algoritmo híbrido mantiene su eficiencia más estabilizada sin importar el tamaño del problema (tanto el del océano como el del lapso de tiempo de simulación), manteniéndose en general por encima del logrado por MPI. Por su parte, el algoritmo de Pasaje de Mensajes (MPI) incrementa su eficiencia al aumentar el tamaño del océano, logrando de esta manera superar la solución híbrida en las pruebas con el océano más grandes.

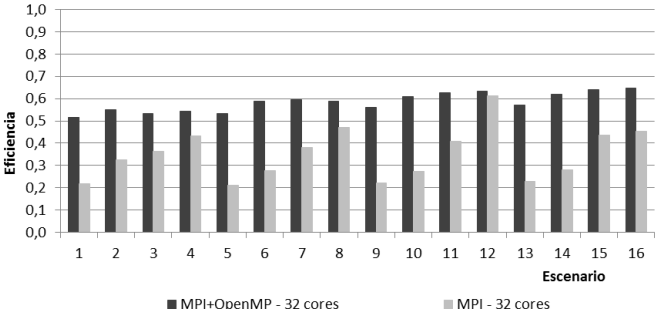

Fig. 6. Eficiencia obtenida en la solución hibrida y pasaje de mensajes para 32 cores.

Al trabajar con una arquitectura más grande, es más notoria la ventaja lograda por el algoritmo híbrido, como se puede observar en la Fig. 6. En este caso siempre consigue mejorar la eficiencia lograda por la solución de MPI.

6 Conclusiones y trabajos futuros

En este trabajo se realizó I/D de algoritmos paralelos sobre arquitecturas de soporte actuales de múltiples núcleos (multicores y cluster de multicores), con un ámbito de aplicación como los problemas de simulaciones con alta demanda computacional.

El caso de estudio elegido fue un problema conocido como lo es el de Sharks and Fishes, cuyo patrón algorítmico es representativo de otros problemas de cómputo científico.

Se plantearon diferentes soluciones paralelas utilizando distintos modelos de comunicación: memoria compartida (OpenMP), pasaje de mensajes (MPI) e híbrido combinando ambos esquemas (MPI+OpenMP). Asimismo, se evaluó la performance de las soluciones obtenidas, comparando por un lado las soluciones "puras" y por otro las que pueden trabajar en una arquitectura híbrida.

Si se comparan las soluciones "puras", la eficiencia lograda por el algoritmo de memoria compartida es (en general) superior al de pasaje de mensajes, siendo más notoria la diferencia al incrementar la cantidad de hilos/procesos. Esto se debe en parte a que se puede aprovechar eficientemente los niveles de memoria compartida de la arquitectura teniendo en cuenta que este tipo de problema requiere extensivo uso de memoria, evitando de esta manera las comunicaciones explícitas y sincronizaciones implícitas del pasaje de mensajes.

Si se compara la solución híbrida con las de Pasaje de Mensajes se puede ver que al aumentar la cantidad de cores utilizados, la eficiencia se reduce en ambos algoritmos. Sin embargo este decremento es más amplio en el algoritmo con MPI que en el híbrido. Con estos resultados se puede notar que la solución híbrida se comporta mejor que la de pasaje de mensajes.

Como trabajo futuro se plantea extender el estudio de escalabilidad para tamaños de problemas más grandes e incorporar heterogeneidad en la arquitectura para estudiar también el impacto del mapeo de procesos y datos a procesadores. Por último, se pretende extender las estrategias y resultados obtenidos a otros problemas de cómputo científico que presenten características similares.

Referencias

1. Barney B, "Introduction to parallel computing", Lawrence Livermore National Laboratory, 2010.
2. Wilkinson B, Allen M, "Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers", 2da Edición, Pearson Prentice Hall, 2005.
3. Mc Cool M, "Programming models for scalable multicore programming", 2007, http://www.hpcwire.com/features/17902939.html.
4. 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.
5. 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.
6. Rauber T, Rünger G, "Parallel programming for Multicore and Cluster Systems", Springer, 2010.
7. Grama A, Gupta A, Karypis G, Kumar V, "Introduction Parallel Computing", Pearson Addison Wesley, 2da Edición, 2003.
8. Dongarra J. , Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. "Sourcebook of Parallel computing". Morgan Kaufmann Publishers - ISBN 155860 871 0 (Capítulo 3).