Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

"PROGRAMACION PARALELA EN UN SISTEMA BEOWULF BAJO LINUX Y MPI: SIMULACION NUMERICA TRIDIMENSIONAL DE PROBLEMAS DE ELECTRODEPOSICION"

Autor:
Pablo Milano – LU 180/98
{pmilano@dc.uba.ar}

Director:
Dr. Guillermo Marshall
{Marshalg@mail.retina.ar}

Diciembre de 2003

Resumen

La computación paralela ocupa un lugar de primer nivel dentro de las áreas de informática que mayor desarrollo han tenido en los últimos años. La computación paralela nació por la necesidad imperiosa del desarrollo científico y tecnológico de acceder a un mayor poder de cómputo, necesidad que simplemente no podía ser satisfecha por arquitecturas con un único procesador. El objetivo de la computación paralela es entonces, resolver problemas más grandes en menos tiempo. Lamentablemente, no siempre es posible para las universidades contar con dicho equipamiento, motivo por el cual, las experimentaciones con simulaciones numéricas de sistemas complejos se ven limitadas en tiempo de proceso al poder de cálculo de una computadora personal, así como también se limitan en cuanto a su tamaño, a la cantidad de memoria disponible en dicha computadora. Es por ello que surge la idea de agrupar computadoras personales en clusters llamados "Beowulf", los cuales, permiten correr algoritmos paralelos diseñados especialmente para dichos clusters y obtener así, una performance similar a la de las grandes supercomputadoras con costos mucho menores, y por ende con mayor factibilidad para los investigadores.

En este trabajo, se introduce una implementación paralela sobre un sistema Beowulf de un modelo tridimensional secuencial para la simulación numérica de un problema de electrodeposicion en celdas planas. Asimismo, se introducen varias técnicas tendientes a la optimización del cálculo de dicha implementación cuyo objeto es mejorar la performance de la misma. Dichas técnicas consisten en un balance de carga semi-dinámico (para un mejor aprovechamiento de los recursos de un cluster heterogéneo, sin necesidad de conocer de antemano las características del mismo), alteración del orden lexicográfico de cálculo, y solapamiento de operaciones de cálculo con intercambio de mensajes para reducir los tiempos ociosos de comunicación y espera entre procesos. También se estudia el desempeño de las comunicaciones no bloqueantes en segundo plano en diferentes implementaciones de MPI, y se evalúa el comportamiento de las mismas con diferentes cantidades y tamaños de buffer. Todas estas técnicas permitieron mejoras en la performance del orden del 40% con respecto a una paralelización no optimizada. Por último, se estudiaron las diferencias obtenidas entre el algoritmo serial y el paralelo, y se propone un método para eliminarlas.

Si bien tanto el método de paralelización como el de optimización son lo suficientemente genéricos como para ser aplicados a cualquier algoritmo que resuelva problemas de cálculo numérico mediante métodos iterativos, se eligió para el desarrollo práctico un problema de electrodeposición en celdas delgadas (ECD) en 3 dimensiones.

Luego del desarrollo y optimización de la implementación, se lograron importantes reducciones en el tiempo de cálculo con respecto a la ejecución secuencial, lográndose en algunos casos, mediante la utilización de comunicaciones no bloqueantes, eficiencias cercanas a la ideal.

El software de paralelización desarrollado en este trabajo ha permitido estudiar problemas de ECD en la frontera del conocimiento, en una escala que antes estaba restringida a centros de supercomputación de países desarrollados

Abstract

Parallel computing is one of the major computer science areas which has evolved in the recent years. It was born due to the strong need among technicians and scientists of achieving bigger amounts of computer power. This need could not be satisfied using single-processor architectures. The main goal of parallel computing is the, to resolve bigger problems in less time. . Unfortunately, researchers and universities usually can not afford supercomputers, so, the simulations are limited to single computer capabilities. These limitations relate to processing time and size of the complex systems numerical simulations. In this environment, the need was detected to start grouping personal computers into "Beowulf" clusters which provide high performance levels similar to those achieved by supercomputers. By designing parallel programs specifically to be run in these clusters, it is possible to obtain results in reasonable times using the same research sizes than the ones run in supercomputers. The Beowulf clusters are much cheaper than the supercomputers which makes complex systems research much easier and affordable to universities and educational institutions.

This work provides the development of a parallel algorithm to a complex system numerical simulation for use in a Beowulf cluster. Optimization techniques were also used in order to increase the program performance. Some of these techniques are: dynamic load-balancing to better use up system resources in an heterogeneous clusters, changes in lexicographic calculus order and overlap between calculus and data transfer to reduce the amount of idle time inside each processor. Research was also done about performance of background non-blocking communications in MPI with different amounts of buffers and different buffer sizes. All these techniques led to a performance rise of about 40% in comparison with a non-optimized parallel algorithm. Finally, differences in results between serial and parallel algorithms were evaluated and a technique was proposed in order to save them.

Although the working method of this parallelization is general enough to be used with any relaxation algorithm, it was applied to a particular problem in electrochemical deposition in thin-layer cells (ECD) in three dimensions.

After development and optimization of the parallel algorithm, very high calculus time decreases were achieved, reaching in some cases performance levels close to the ideal

The parallel algorithm developed in this work allowed the study of ECD problems in the boundary of knowledge, using scales which used to be restricted to supercomputer centers in developed countries.

Agradecimientos

Quiero agradecer en primer lugar a mi director de Tesis, por su apoyo y por la conducción y supervisión de este trabajo.
A Esteban Mocskos, por su paciencia y esmero, al brindarme la introducción y la presentación del trabajo previo a esta tesis, así como sus respuestas a mi gran cantidad de dudas iniciales.
A Andrés Glavina y Darío Liberman, por su colaboración en el primer tramo de este trabajo.
A Carolina León Carri, por el invaluable soporte en lo referente a la administración y funcionamiento del cluster Beowulf "Speedy Gonzalez", que realizó en todo momento sin perder el buen humor.

Índice General

Capítulo 1 – Introducción

La computación paralela ocupa un lugar de primer nivel dentro de las áreas de informática que mayor desarrollo han tenido en los últimos años. La computación paralela nació por la necesidad imperiosa del desarrollo científico y tecnológico de acceder a un mayor poder de cómputo, necesidad que simplemente no podía ser satisfecha por arquitecturas con un único procesador. El objetivo de la computación paralela es entonces, resolver problemas más grandes en menos tiempo.

El problema central en computación paralela es la manera en que un procesador puede acceder a los resultados calculados por otro procesador. Esto requiere transferencia de datos. Los sistemas paralelos MIMD (multiple instruction multiple data) pueden dividirse en sistemas con arquitectura de memoria compartida (SMP), y en sistemas con arquitectura de memoria distribuida [1]. Una alternativa, el OpenMP, en donde el compilador inserta directivas en el programa, funciona exclusivamente en sistemas con memoria compartida. Otra alternativa, es el MPI (Message Passing Interface), donde existen una serie de rutinas de biblioteca estandarizadas que funcionan en sistemas con memoria distribuida, y también en sistemas SMP. La comparación entre MPI y SMP es compleja. En general, los algoritmos cuya escala de comunicación es pequeña o mediana son ideales para sistemas MPI mientras que algoritmos cuya escala de comunicación es muy grande funcionan mejor en OpenMP. En este trabajo se analiza el sistema MPI[1-9] y se muestran las razones de su elección.

Beowulf es un sistema para operar un conjunto de PC's conectadas entre sí, para formar una supercomputadora, permitiendo así realizar operaciones en el rango de los Gigaflops (billones de operaciones por segundo). En general se utiliza el sistema operativo Linux y la biblioteca de pasaje de mensajes (MPI), ambos de libre acceso. El sistema Beowulf surgió por la aparición en el mercado masivo de PC's, de computadoras personales con alta capacidad de cálculo (y con prestaciones similares a una supercomputadora de la década del 80). Hoy en día se logra una potencia de cálculo comparable a las de las supercomputadoras convencionales actuales, a un costo típicamente 100 veces menor. Un cluster de esta naturaleza, cuando su número de nodos es apreciable, constituye un sistema de computación de alto rendimiento (HPC, por sus siglas en Ingles: High Performance Computing). La inmensa mayoría de los sistemas Beowulf operan bajo Linux y la biblioteca de pasaje de mensajes MPI (Message Passing Interface) que se ha transformado de hecho en el standard de paralelización frente a su antecesor el sistema PVM (Parallel Virtual Machine). En este trabajo, siguiendo la tendencia general en los países desarrollados, se utiliza el sistema operativo Linux y la biblioteca de pasaje de mensajes MPI [1-9].

Algunas ventajas relativas del sistema Beowulf son: 1) Flexibilidad frente a los cambios tecnológicos dado por la modularidad de la arquitectura. 2) Alta escalabilidad ya que es relativamente fácil agregar un módulo a este tipo de configuración. 3) Mantenimiento mínimo y de bajísimo costo, debido a que proviene de componentes comerciales, probados y fácilmente reemplazables. 4) El software de base es de dominio público y en constante evolución y 5) Las tecnologías de redes como FastEthernet son mejor conocidas y tienden a bajar de precio. En el sistema Beowulf, el conjunto de nodos (procesadores más almacenamiento) están interconectados por una red de alta velocidad (todos con todos) corriendo un sistema operativo único que la convierte en una única máquina virtual frente al usuario.

La literatura reciente muestra que los clusters Beowulf evolucionan vertiginosamente y que se han erigido en el Standard de facto de sistemas paralelos y que cada vez es mayor el número de aplicaciones que migran a este sistema, y en este sentido no se vislumbra, a mediano plazo, un techo para la evolución de este tipo de herramientas. A manera de ejemplo puede citarse el cluster que se construye actualmente en el Lawrence Livermore National Laboratory, basado en Linux como sistema operativo, y que tiene 1920 procesadores Intel Xeon a 2.4 GHz, que brindarán un pico teórico de 9.2 teraFLOPS (9,2 trillones de operaciones por segundo). (ver http://www.topclusters.org/). Otro ejemplo que puede mencionarse es el High Performance Computing Center (CHPC) de la Universidad de Utah, que posee 300 nodos con 450 procesadores.

En nuestro país, la informática y en particular la computación paralela son áreas de vacancia, dado que están notoriamente rezagadas en relación a otras disciplinas del conocimiento. Las instituciones de promoción de la ciencia, en particular, la FCEyN, el FONCYT y el CONICET, han hecho un esfuerzo considerable para paliar esta situación financiando proyectos limitados de centros de supercomputación. Al respecto, existen en la actualidad, en la FCEyN, centros de computación que desde el año 2001 operan clusters de PC's con arquitectura Beowulf (cluster Bocha de 40 nodos del Departamento de Física y Departamento de Computación; cluster Speedy Gonzalez de 16 nodos del Laboratorio de Sistemas Complejos (LSC) del Departamento de Computación; y cluster Heman de 20 nodos del Departamento de Química Inorgánica).

Desde el punto de vista de las Tecnologías de Información y Comunicación (TIC´s), estas nuevas arquitecturas de cálculo plantean interesantes problemas fundamentales en el dominio de la computación paralela como resultan, por ejemplo, la evaluación del hardware de los nodos del Beowulf, el impacto de la arquitectura y tecnología de redes en el rendimiento de clusters, adecuación de sistemas operativos, la implementación de nuevos algoritmos de cálculo, entre otros, y en lo que respecta a las aplicaciones informáticas cabe mencionar el acceso a base de datos en paralelo, la posibilidad de replicación de bases de datos para seguridad informática, inteligencia artificial, redes neuronales, robótica y criptografía, entre otros.

Desde el punto de vista de la investigación científica, un sistema de esas características permite acceder a una enorme variedad de problemas importantes que previamente se encontraban restringidos a instituciones con recursos suficientes para afrontar el alto costo de un centro de supercómputo. A modo de ejemplo, entre los problemas que la computación masivamente paralela permitiría abordar se destacan la dinámica de fluidos, la turbulencia, la dinámica molecular, la formación de patrones y estructuras en sistemas complejos, la prospección sísmica, los métodos de predicción de estructuras proteicas por analogía de secuencia, los problemas de biología estructural relacionados con plegamiento de proteínas, el estudio de terapias electroquímicas contra el cáncer, la simulación de propiedades de materiales mediante técnicas de estructuras electrónica, el diseño de micro-arrays y secuenciamiento de genoma, las simulaciones de circulación atmosférica en meteorología, y los algoritmos multiobjetivos distribuidos para búsquedas de patrones en bioinformática, entre otros.

En muchos de estos casos resulta necesario resolver numéricamente ecuaciones en derivadas parciales utilizando, entre otras técnicas, Diferencias Finitas, Elementos Finitos, Métodos Espectrales o Monte Carlo, y adecuarlas al procesamiento en paralelo. Esto implica utilizar paralelismo de los datos (distribución de los datos entre los procesos) o

paralelismo del control (partición de las instrucciones). En general se elige la primera alternativa que se reduce a la descomposición geométrica en subdominios y su adecuación a la arquitectura del sistema. Una tarea delicada que surge en este contexto es encontrar la técnica más adecuada para resolver grandes sistemas lineales algebraicos. Los métodos iterativos resultan particularmente atractivos por su alto grado de paralelización.

En este trabajo nos concentramos en la optimización de la performance de un algoritmo paralelo de simulación numérica por diferencias finitas y métodos iterativos de resolución de sistemas algebraicos, de un problema de electrodeposición en celdas planas delgadas. En particular estudiamos, en el contexto de MPI, la relación existente entre el no bloqueo en el pasaje de mensajes entre procesos y el calculo útil en los procesos. Este tema es fundamental en la performance del MPI.

El plan de este trabajo consiste en una introducción al problema de electrodeposición, el modelo matemático y numérico y su solución secuencial, temas que se desarrollan en el presente capítulo. En el capítulo 2 se desarrollan temas de paralelización, tales como speedup, eficiencia, entre otros. El capitulo 3 se aboca al algoritmo de paralelización en 3 dimensiones. El capitulo 4 trata sobre performance y optimización. El capitulo 5 presenta un análisis de los resultados. En el capitulo 6 se realiza una discusión general y finalmente, en el capítulo 7 se sugieren nuevas líneas de investigación.

En este primer capítulo, se describe en forma sucinta, el proceso de electrodeposición electroquímica en celdas delgadas (ECD) en 3 dimensiones, que fue elegido como modelo de investigación en este trabajo de paralelización y optimización. A continuación, se menciona brevemente la implementación serial previa de la simulación numérica de ECD que fue utilizada como punto de partida de este trabajo. Posteriormente, se comenta la motivación de la paralelización de éste y otros sistemas de simulación numérica, basada principalmente en los grandes requerimientos de cálculo y memoria que caracterizan a las implementaciones de este tipo de simulaciones numéricas. Se verán también los criterios de medición de eficiencia y performance de los algoritmos paralelos, que son abordados en mayor detalle en el capítulo 4. Por último, se define el concepto de cluster Beowulf, para el cual fue diseñado el algoritmo paralelo del presente trabajo, y al cual pertenece el cluster "Speedy Gonzalez" del Laboratorio de Sistemas Complejos (LSC) del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires, que fue utilizado para realizar las pruebas de performance y escalabilidad.

1.1 El problema de electrodeposición en celdas delgadas (ECD) en 3 dimensiones

Dado que se utilizó como modelo para este trabajo, el proceso de electrodeposición en celdas delgadas, y se paralelizó un algoritmo preexistente que lo resuelve, se realiza a continuación, a modo de introducción teórica, una breve descripción de este proceso, para luego focalizar en los temas referentes a la paralelización y optimización del algoritmo.

1.1.1 Introducción al proceso de ECD

En un experimento de electrodeposición en celdas planas (ECD), la celda electrolítica consiste en dos portaobjetos de microscopio que encierran dos electrodos paralelos en sus extremos y un electrolito de una sal metálica (por ejemplo, sulfato de cobre o zinc en agua destilada). Una diferencia de potencial aplicada entre electrodos produce una deposición ramificada por reducción de los iones metálicos. La morfología del depósito puede ser fractal, densamente ramificada o dendrítica dependiendo de la geometría de la celda, concentración del electrolito, voltaje aplicado, entre otros parámetros. El ECD es un fenómeno paradigmático en el estudio de la formación de patrones de crecimiento [10-20] Estos fenómenos son particularmente relevantes en la industria electrónica y en problemas de generación de energía eléctrica como las baterías o celdas solares. Al respecto, una nueva e interesante aplicación de ECD es en el estudio del tratamiento electroquímico de tumores, que consiste en la destrucción del tejido tumoral por pasaje de corriente a través del mismo.

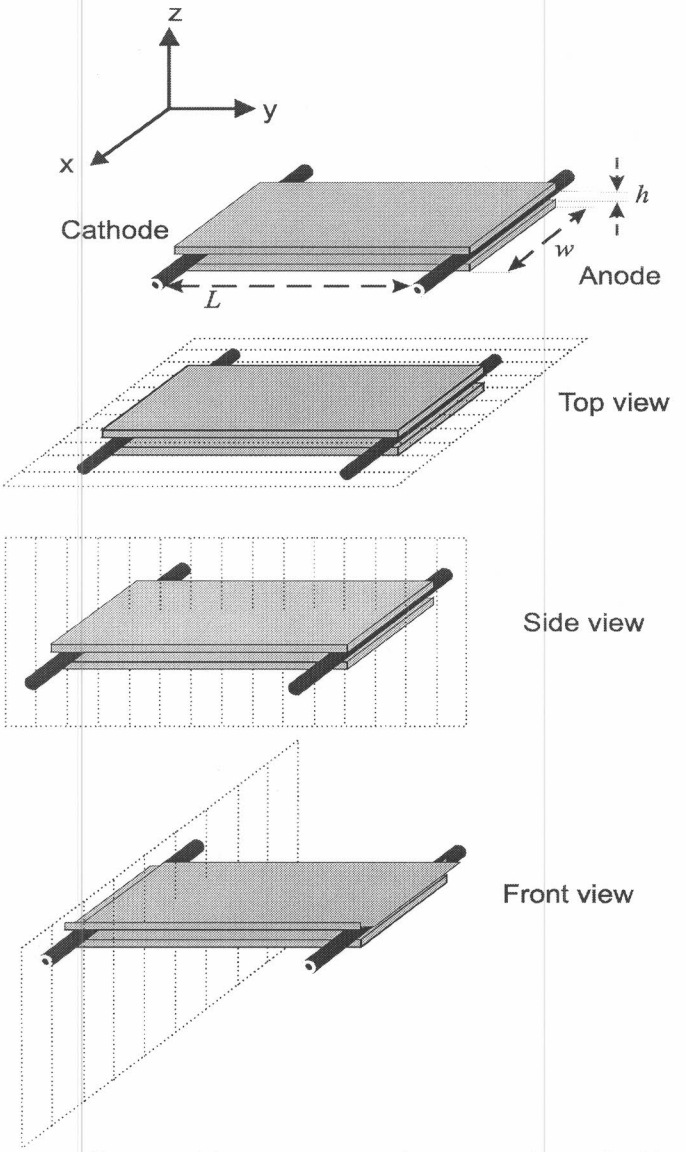


Fig. 1.1.1.a – Diagrama del sistema experimental y convención de vista de celdas [20]

El transporte iónico está básicamente gobernado por la migración, la difusión y la convección. A su vez, la convección está gobernada por la fuerzas de Coulomb debidas a cargas eléctricas locales y por fuerzas de empuje debidas a gradientes de concentración que inducen gradientes de densidad. La relevancia de la convección frente a los otros modos de transporte, para celdas con un espesor mayor a 50 micrones fue demostrado por varios investigadores [11-20]. El estudio del peso relativo de la convección frente a la migración y la difusión es relevante por que determina el tipo de transporte iónico gobernante y por ende del crecimiento y la morfología del deposito. Estas últimas determinan las características del depósito.

En la figura 1.1.1.b se muestra una captura de video de una sección angosta de la celda en posición horizontal con una imagen de Schlieren (contraste diferencial) de dendritas (píxeles oscuros) creciendo desde el cátodo (a la izquierda) hacia el ánodo. Estas figuras experimentales fueron obtenidas con técnicas similares a las utilizadas en la referencia [15]. El crecimiento comienza como una pequeña perturbación en el cátodo y rápidamente se transforma en un bosque de dendritas que finalmente llegan al ánodo, establecen el contacto y mueren. Muchas de estas dendritas detienen su crecimiento a medio camino apantalladas por las sobrevivientes, como se observa en la figura. La explicación de este fenómeno es como sigue. Cuando se cierra el circuito, la corriente eléctrica fluye a través de la celda y se desarrollan capas límites vecinas a los electrodos. En el ánodo la concentración aumenta en relación a la concentración inicial debido al transporte de aniones hacia el cátodo, y a la disolución de aniones metálicos en el ánodo. En el cátodo la concentración de cationes decrece cuando los iones metálicos son reducidos y depositados y los aniones escapan hacia el ánodo. Estas variaciones de concentraciones dan lugar a corrientes de densidad y al desarrollo de vórtices o rollos gravitoconvectivos en la zona vecina a los electrodos [13]. Esto se ilustra en la figura que muestra imágenes de Schlieren de los rollos de concentración catódico y anódico (píxeles claros) rodeando el frente de ramificación y el ánodo (píxeles oscuros), respectivamente.

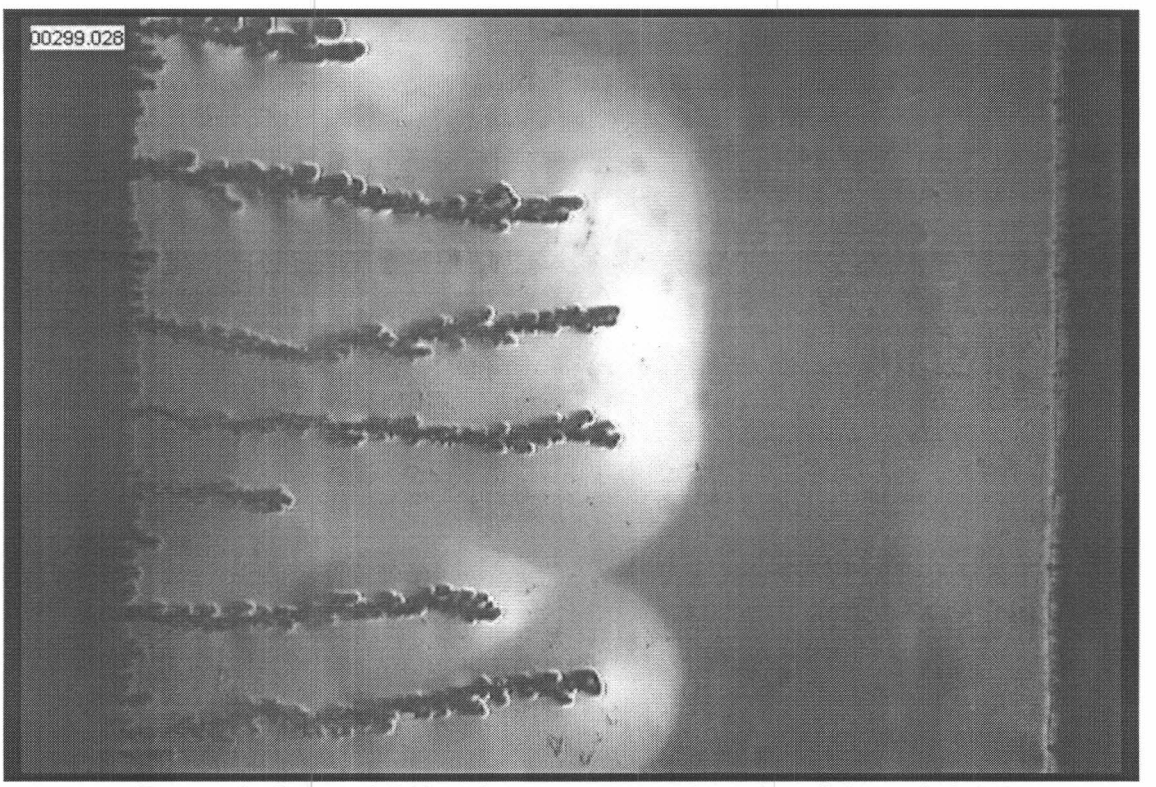


Fig. 1.1.1.b – Imágenes de Schlieren de una agregación parcialmente desarrollada, tomada de [20]

Luego de algunos instantes de iniciado el proceso, se genera una inestabilidad que gatilla el crecimiento del depósito en el cátodo. Este se desarrolla como un arreglo tridimensional de filamentos metálicos porosos. Las fuerzas de Coulomb se concentran en las puntas de acuerdo a un modelo desarrollado en [12]-; cada filamento poroso permite que el fluido penetre por la punta y sea eyectado por sus costados, formando un par de vórtices en el plano de la agregación gobernado por la fuerza eléctrica. Es claro que la ECD en celdas delgadas es un complejo proceso de interacción tridimensional entre los vórtices gobernados por el campo eléctrico y los gobernados por el campo gravitatorio. En un trabajo de reciente publicación, [20] se presenta un modelo tridimensional y su implementación en una maquina secuencial que describe la interacción entre vórtices y el deposito. Dicho trabajo sirve de base para la implementación paralela que se presenta aquí.

1.1.2 – El modelo numérico de ECD

El transporte iónico en electrodeposición en celdas delgadas puede ser descripto con un modelo matemático basado en primeros principios. Este modelo incluye las ecuaciones de Nerst-Planck para el transporte iónico, la ecuación de Poisson para el potencial eléctrico, y la ecuación de Navier-Stokes para el fluido. El sistema de ecuaciones adimensionalizado en 3D se puede escribir como [20]:

$$\frac{\partial C_i}{\partial t} = -\nabla . j_i \quad (1)$$

$$j_i = -M_i C_i \nabla \phi - \frac{1}{Pe_i} \nabla C_i + C_i v \quad (2)$$

$$\nabla^2 \phi = Po \sum_i z_i C_i \quad (3)$$

$$\frac{\partial}{\partial t} + \nabla \times (\zeta \times v) = \frac{1}{\mathrm{Re}} \nabla^2 \zeta + \sum_i \left[Ge_i z_i (\nabla \phi \times \nabla C_i) - Gg_i (\nabla \times C_i) \frac{\delta}{g} \right] \quad (4)$$

$$\zeta = -\nabla^2 \psi \quad (5)$$

$$v = \nabla \times \psi \quad (6)$$

donde C_i y J_i son la concentración y el flujo de la especie iónica i adimensionalizados (para un electrolito ternario como el caso del $ZnSO_4$ /H_2SO_4, i = C, A y H, para los iones zinc, sulfato e hidrógeno respectivamente); v, ϕ, ζ y ψ son la velocidad del fluido, el potencial electrostático, el vector vorticidad y el potencial vector velocidad, respectivamente adimensionalizados; δ / g es el versor en la dirección del campo gravitatorio. Las cantidades: $M_i = \mu_i \phi_0 / x_0 u_0, Pe_i = x_0 u_0 / D_i, Po = x_0^2 C_0 e / \varepsilon \phi$, $\mathrm{Re} = x_0 u_0 / v, Ge_i = e C_i \phi_0 / p_0 u_0^2$ y $Gg_i = x_0 C_i g \alpha_i / u_0^2$ representan los números adimensionales de Migración, Peclet, Poisson eléctrico, Reynolds, Grashof eléctrico y Grashof gravitatorio respectivamente. Las cantidades z_i, μ_i, y D_i son, el número de carga

por ion, la movilidad y la constante de difusión de cada especie iónica i, μ_i y z_i son cantidades enteras, siendo positiva para cationes y negativa para aniones; g es la aceleración gravitatoria; e es la carga eléctrica, ε es la permeabilidad del medio y ν es la viscosidad cinemática. x_0, μ_0, ϕ_0, C_0 y p_0 son valores de referencia de la longitud, velocidad, potencial electrostático, concentración y densidad del fluido respectivamente. Para que el sistema sea consistente, se utiliza la aproximación de Boussinesq para la densidad del fluido:

$$p = p_0(1 + \Sigma_i \alpha_i \Delta C_i) \text{ ; donde } \alpha_i = \frac{1}{p_0}\frac{\partial p}{\partial C_i}$$

El sistema (1-6), con las condiciones de contorno y las condiciones iniciales adecuadas, es válido en un dominio espacio-temporal definido por $G = [\Omega(t) \times (0.t)]$, donde Ω es una región tridimensional con contorno $\Gamma(t)$; este contorno se mueve con una velocidad proporcional a la norma del j_i. Las condiciones de contorno para el vector potencial de velocidades son: en una superficie plana impermeable, el vector es normal a la superficie y su gradiente es cero; en las superficies sin deslizamiento, la derivada tangencial de las componentes de la velocidad son cero.

Dependiendo de cual régimen prevalezca, electroconvectivo o gravitoconvectivo, ambos comparten los números de Migración, Peclet, Poisson eléctrico, Froude y Reynolds, y difieren en el número de Grashof que actúa en la ecuación de Navier-Stokes. En el caso límite de un flujo puramente electroconvectivo, como es el caso de los experimentos de ECD en condiciones de microgravedad, el número de Grashof eléctrico gobierna el flujo; cuando la gravitoconvección es dominante, el número de Grashof gravitatorio es el que gobierna. Aumentando el número de Grashof eléctrico o el gravitatorio, aumenta la electro o gravitoconvección respectivamente.

El modelo computacional resuelve el sistema de ecuaciones presentado en 3D, para cada paso temporal, en un dominio fijo o variable, en una malla uniforme en 3D, usando diferencias finitas (ver detalles, por ejemplo, en [21] y técnicas de relajación determinísticas (detalles sobre el modelo 3D pueden encontrarse en [14-16] y [20]). Para construir las figuras se utilizó OpenDX [22], que es un programa de dominio público para graficación científica.

1.2 La solución numérica secuencial

Como punto de inicio, se tomó una solución secuencial preexistente del problema de ECD en 3 dimensiones presentado en [20]. A modo de ilustración, se explica a grandes rasgos el funcionamiento del algoritmo serial sobre el cual se basó este trabajo. Dicho algoritmo, realiza los siguientes pasos:

1) Lectura del archivo de configuración e Inicialización de variables y matrices:
El proceso lee un archivo de configuración, e inicializa los valores correspondientes, luego reserva la memoria para las matrices y las inicializa.

2) Iteración de Cálculo:
Aquí se recorren todos los elementos de todas las matrices, calculando en el paso i, los valores M_i en base a los valores M_{i-1} o a los vecinos de M_i si ya fueron calculados.

3) Evaluación de condición de fin de iteración:
Una vez terminada la iteración de cálculo, se evalúa el error de convergencia, que es el modulo de la máxima diferencia entre cada elemento M_i y el elemento M_{i-1} de todas las matrices que intervienen en el calculo. Si este valor es menor que un umbral definido, se avanza a un siguiente paso de tiempo y se vuelve a 2. En caso contrario, se realiza otra iteración (paso 2).

4) Escritura de los resultados:
El proceso escribe en archivos con formato NetCDF los resultados de los cálculos.
NetCDF (network Common Data Form) es una interfaz para el acceso y almacenamiento de datos científicos en formato matricial, que provee librerías para diversos lenguajes de programación [20].

5) Evaluación de condición de fin de cálculo:
Si se ejecutaron una cantidad predefinida de iteraciones, o el error entre los diferentes pasos de tiempo es lo suficientemente pequeño, entonces se termina la ejecución

Capítulo 2 – Paralelización

En este capítulo se explica la motivación para la paralelización de los algoritmos de simulación numérica de sistemas complejos, los criterios de medición de performance para dichos algoritmos y la descripción de los clusters Beowulf en los que se corre la aplicación.

2.1 ¿Por qué paralelizar?

Como se mencionara anteriormente, las simulaciones numéricas en sistemas complejos son sistemas de muy alta demanda de cálculo, motivo por el cual, requieren de tiempos de proceso que en muchos casos resultan inviables en máquinas secuenciales. El tamaño de los problemas n-dimensionales crece en forma exponencial al aumentar el tamaño de la malla, lo cual aumenta de la misma manera el tiempo de cálculo así como los requerimientos de memoria. Por esta razón, con la paralelización se busca conseguir 2 fines: El primero es reducir los tiempos de cálculo aprovechando el poder de cómputo de un cluster Beowulf, y el segundo es permitir el cálculo de simulaciones sobre mallas de mayor tamaño, aprovechando la mayor disponibilidad de memoria que se logra al sumar las memorias de los distintos nodos del cluster.

De esta forma, se pueden realizar simulaciones que no podrían llevarse a cabo en las computadoras secuenciales con las que cuentan los laboratorios universitarios.

2.2 Medición de performance de algoritmos paralelos

Al desarrollar sistemas paralelos, se tiene como objetivo no solo alcanzar la mayor performance posible, sino que la misma se mantenga en proporción a medida que se agregan procesadores al cálculo. Para poder tener una noción de la medida de performance de los algoritmos paralelos, se definen las métricas de speedup, eficiencia, overhead y escalabilidad, que se explican a continuación:

2.2.1 Speedup

El Speedup de un programa paralelo, es la relación entre la versión paralela de un algoritmo, para una cantidad dada de procesadores, y el tiempo de ejecución de la versión serial del mismo algoritmo. Para poder calcularlo, necesitamos definir primero:

T1(n): Tiempo necesario para la ejecución de un programa serial
Tp(n,p): Tiempo necesario para la ejecución de la versión paralela del mismo programa con p procesadores.

Se define entonces como speedup de un programa paralelo corriendo en p procesadores a:

$$S(n,p) = \frac{T1(n)}{Tp(n,p)}$$

Esta medida sirve para comparar el rendimiento del algoritmo paralelo con respecto al serial. Cuando $S(n,p) = p$, se dice que se cuenta con un programa de "speedup lineal". El speedup lineal es el óptimo desde el punto de vista de la paralelización, puesto que indica una proporción directa y sin desperdicios entre el tiempo de ejecución y la cantidad de procesadores que intervienen.[1] Es así que $0 < S(n, p) \leq p$. Cuanto mayor sea el speedup de un algoritmo paralelo, mejor será la performance del mismo en la ejecución con p procesadores.

2.2.2 Eficiencia

La eficiencia de un algoritmo paralelo, como su nombre lo indica, expresa el aprovechamiento de los recursos de procesador, por parte de dicho algoritmo. Se define como:

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T1(n)}{pTp(n, p)}$$

de donde se deduce que, para el caso óptimo de speedup lineal $E(n, p) = 1$. Para el caso de la Eficiencia, las cotas son: $0 < E(n, p) < 1$. A medida que nuestro valor de eficiencia se acerque a 1, nuestro algoritmo tendrá mejor performance.

2.2.3 Overhead

Hay muchos factores que influyen negativamente en los tiempos de ejecución de un algoritmo paralelo, y son inherentes a la paralelización (no existen en un algoritmo serial). Algunos de estos factores son:

- Intercambio de mensajes entre los procesos
- Cálculos y decisiones adicionales requeridos en la versión paralela (identificación de ID, partición en subdominios, etc.)
- Tiempos de espera de un proceso (cuando necesita datos de un proceso vecino para poder continuar)
- Partes inherentemente seriales del algoritmo (No siempre se puede paralelizar la totalidad del algoritmo)

Es por esto, que existe el concepto de overhead de un algoritmo paralelo. Antes de definirlo, necesitamos definir el concepto de "trabajo" realizado por un algoritmo. En el caso de un algoritmo secuencial, El trabajo realizado por el mismo, es el tiempo que se necesita para su ejecución:

$$W1(n) = T1(n)$$

[1] En la práctica, es casi imposible lograr un algoritmo con Speedup lineal, debido al overhead que toda paralelización conlleva.

Cuando se trata de un algoritmo paralelo, el trabajo realizado por el mismo es la suma del trabajo de cada uno de los procesadores:

$$Wp(n,p) = \sum_{i=1}^{p} Wi(n,p)$$

Puesto que los tiempos de espera en cada procesador forman parte de su trabajo, resulta que $Wi(n,p)=Tp(n,p)$ con lo cual:

$$Wp(n,p) = \sum_{i=1}^{p} Wi(n,p) = \sum_{i=1}^{p} Tp(n,p) = pTp(n,p)$$

Con estas definiciones previas, decimos entonces que el *overhead* de un algoritmo paralelo es el trabajo adicional que dicho algoritmo realiza respecto a la solución serial equivalente:

$$T0(n,p) = Wp(n,p) - W1(n) = pTp(n,p) - T1(n)$$

El overhead, que está presente en virtualmente todos los algoritmos paralelos, es el responsable de no poder lograr un "speedup lineal". Al diseñar algoritmos paralelos, se trabaja para minimizar el overhead y poder lograr así un aumento del speedup.

2.3 Límites teóricos de la performance de un algoritmo paralelo

Si bien la paralelización de un algoritmo provee un aumento de la performance con respecto al algoritmo serial, este aumento no es necesariamente ilimitado. Inicialmente, Amdahl [3] evaluó que la máxima eficiencia que un algoritmo paralelo puede lograr, con respecto a su solución serial, está sujeta a la proporción paralelizable del algoritmo serial y no depende de la cantidad de procesos que intervengan en el cálculo. Esto es conocido como "Ley de Amdahl". Posteriormente, Gustafson [4] re-evaluó la ley de Amdahl, considerando que un problema paralelo puede "agrandarse" para mantener la performance a medida que se aumentan los procesadores. A continuación se mencionan ambos estudios.

2.3.1 Ley de Amdahl

Al analizar cualquier algoritmo paralelo, se encontrará que al menos una parte del mismo es inherentemente secuencial. Esto limita el speedup que se puede obtener por medio de la utilización de una computadora paralela.
Si la fracción del problema que es inherentemente secuencial es f, entonces el máximo speedup que se puede obtener en una computadora paralela es 1/f. Esto es, el tiempo de ejecución de la solución paralela será:

$$Tp(n,p) = fT1(n) + \frac{(1-f)T1(n)}{p}$$

Y por lo tanto, el factor de speedup será:

$$S(n,p) = \frac{T1(n)}{Tp(n,p)} = \frac{T1(n)}{fT1(n) + \frac{(1-f)T1(n)}{p}} = \frac{1}{f + \frac{(1-f)}{p}} = \frac{p}{1+(p-1)f}$$

Luego, aún contando con infinitos procesadores, el máximo speedup que se puede obtener está limitado por 1/f.

$$\lim_{n\to\infty} S(n,p) = \frac{1}{f}$$

A su vez, la eficiencia de un programa paralelo se acercará a cero a medida que se incorporen nuevos procesadores, puesto que

$$S(n,p) < \frac{1}{f} \quad \text{y} \quad p \to \infty$$

De acuerdo con la Ley de Amdahl y a modo de ejemplo, resulta que si un problema es solo 5% serial, el máximo speedup que se puede obtener con una solución paralela es 20, independientemente de la cantidad de procesadores que se utilicen.

2.3.2 Ley de Gustafson

Gustafson, presenta un argumento para mostrar que la ley de Amdahl no es tan significativa como se pensaba con anterioridad, en limitar el speedup potencial de una solución paralela. Este argumento se basa en la observación de que una computadora paralela de mayor tamaño, permite ejecutar un problema de mayor tamaño también, en un tiempo razonable. De esta forma, se propone mantener fijo el tiempo de ejecución, pero incrementar el tamaño del problema y la cantidad de procesadores de la computadora paralela. Es así que, manteniendo constante el tiempo de ejecución, entonces el factor de speedup resultará diferente al que define Amdahl y se denomina "factor de speedup escalado".

La ley de Gustafson se expresa de la siguiente forma: Sea s la fracción inherentemente serial de un algoritmo paralelo, y sea p la fracción paralelizada. Si se fija de forma constante en $s + p$ el tiempo de ejecución en un procesador, y para simplificar la explicación, se asume que $s + p = 1$, entonces, la ley de Amdahl establece que:

$$S(n) = \frac{s+p}{s+p/n} = \frac{1}{s+(1-s)/n}$$

(Nota: En este caso, la cantidad de procesadores está denotada por n).

Para utilizar el factor de speedup escalado que define Gustafson, se considera constante el tiempo de ejecución en paralelo en lugar del tiempo de ejecución serial. Ahora $s + p$ (la parte serial y paralela del programa, respectivamente) será el tiempo de ejecución en la computadora paralela y, nuevamente, para simplificar la explicación, se asume que $s+p=1$. Entonces, el tiempo que requiere la ejecución en una única máquina para resolver el

mismo problema es $s+np$, puesto que la parte paralela debe ejecutarse en forma secuencial. El speedup escalado será:

$$S_s(n) = \frac{s+np}{s+p} = s + np = s + n(1-s) = n + (1-n)s$$

2.3.3 Clusters Beowulf

Un cluster Beowulf es un conjunto de computadoras personales, no necesariamente homogéneas, interconectadas en red que comparten sus recursos de procesamiento y almacenamiento y pueden funcionar cooperativamente y en forma sincronizada para la ejecución de algoritmos paralelos diseñados especialmente. Estas computadoras, en comparación con las supercomputadoras de alta performance, tienen un costo mucho más accesible, ampliando las posibilidades de equipamiento de los investigadores e instituciones educativas. La posibilidad de coexistencia de diferentes modelos de procesadores y hardware, simplifican la escalabilidad de un cluster Beowulf.

El Laboratorio de Sistemas Complejos del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la UBA posee un cluster Beowulf de 16 nodos llamado "Speedy Gonzalez" que fue utilizado para las simulaciones de este trabajo.

En un cluster Beowulf, cada computadora o procesador se denomina "nodo". De esta forma, se puede abstraer la arquitectura del mismo, aún en los casos de contar tanto con computadoras monoprocesador como multiprocesador en el mismo cluster. Aquí se asume que en cada procesador físico se corre un solo proceso.

Existen diversas aplicaciones de administración de procesos y aplicaciones en clusters Beowulf, muchas de ellas gratuitas y de código abierto. Las más importantes que se utilizan en el laboratorio de Sistemas Complejos son:

- PBS: Administrador de recursos de un cluster. Se utiliza para establecer políticas sobre la asignación y utilización de los recursos (principalmente procesadores) de un cluster. Provee colas de procesos, herramientas para encolar, visualizar y desencolar procesos, herramientas de monitoreo, etc.
- MAUI Scheduler: Encolador avanzado de procesos. Se encarga de administrar la cola de procesos, para permitir la aplicación de prioridades, límites de ejecución, permisos, etc.

Capítulo 3 – Paralelización del algoritmo de ECD en 3 dimensiones

En este capítulo, se detallan los pasos del desarrollo de la solución paralela al sistema de ECD en 3 dimensiones. Se explica el método empleado para la paralelización y el diseño del programa paralelo. Se incluyen también algunos detalles relevantes de la paralelización, tanto de los sistemas complejos en general, como el de ECD en particular.

Durante el desarrollo de la paralelización, se intentó independizar todo lo posible las funciones del cálculo serial original, de la implementación paralela, intentando mantener la mayor fidelidad posible al algoritmo original. De esta forma, se obtiene una solución de paralelización general, que puede ser aplicada sin mayores modificaciones a otros algoritmos de resolución de problemas de cálculo numérico mediante métodos iterativos.

3.1 Partición en subdominios

La paralelización del algoritmo se realiza a través de la paralelización de la cadena de datos (en contraposición a la paralelización de la cadena de instrucciones). Para ello se utiliza el método de descomposición por subdominios. Este método consiste en dividir la malla sobre la cual se realiza el cálculo, en "p" porciones (p = cantidad de procesos), asignándole una porción del dominio a cada uno de los procesos.

Cada uno de los procesos participantes, tiene en memoria el subdominio correspondiente, y realiza sobre el mismo, sus cálculos en forma local, pero, como se explicó en el capítulo 1, para calcular el elemento $A^{n}_{(ij)}$ de la malla, se necesitan, además del valor anterior $A^{n-1}_{(ij)}$, los valores vecinos. Estos valores vecinos, pueden encontrarse dentro de la porción de subdominio perteneciente al proceso local, o bien, pueden pertenecer a un proceso vecino. Es por esto, que es necesario contar con planos de solapamiento y realizar, luego de cada iteración, un intercambio de estos valores con los procesos vecinos. De esta forma, en cada iteración, cada proceso posee, además de su subdominio local, la última copia de los valores correspondientes a los procesos vecinos, para poder realizar sus cálculos locales. Una vez terminados estos cálculos, es necesario obtener de los vecinos los valores actualizados de los planos de intercambio, así como enviar a dichos vecinos los valores actualizados localmente.

Para diseñar la partición en subdominios, se comenzó pensando en un problema bidimensional, para luego extenderlo al caso de 3 dimensiones.
La partición en subdominios, se puede realizar a lo largo de un único eje de la malla de cálculo, o por bloques a lo largo de ambos ejes, como se muestra a continuación.

La partición en 1 eje, es la más simple. En ella, se disponen los procesos "en línea" y cada uno de ellos, intercambia valores con su vecino de la izquierda, y con su vecino de la derecha. De esta forma, se necesitan 2M operaciones de intercambio por proceso (M=cantidad de matrices) excepto para el caso de los procesos de los extremos, para los cuales solo se necesitan M operaciones. En este caso, es relativamente sencillo implementar un esquema de balance de carga, puesto que alcanza con dimensionar los intervalos de manera conveniente. Por último, esta partición permite dividir el problema en cualquier

cantidad de subprocesos, siempre que este número sea menor a los elementos del eje elegido.

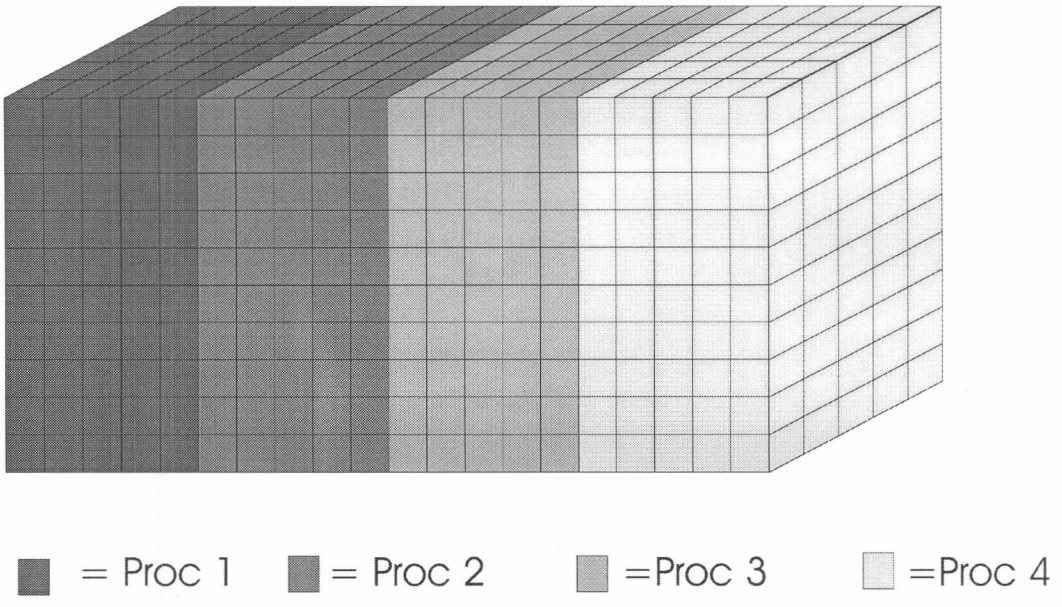


Figura 3.1.a – Partición en subdominios unidimensional

La partición en más de 1 eje es más compleja de implementar, puesto que no es fácil determinar un criterio de asignación de bloques que sea conveniente para distintos tamaños de malla y distintos valores de P. En particular, para algunos valores, la partición puede resultar bastante problemática (ej: 7 procesos en una malla de 20x100x20). Según el proceso, se pueden necesitar 2M, 3M o 4M operaciones de intercambio, puesto que no todos los procesos tienen la misma cantidad de vecinos. Por último, es extremadamente complicado pensar en una solución de balance de carga, conforme la dificultad de dimensionar los bloques de forma adecuada.

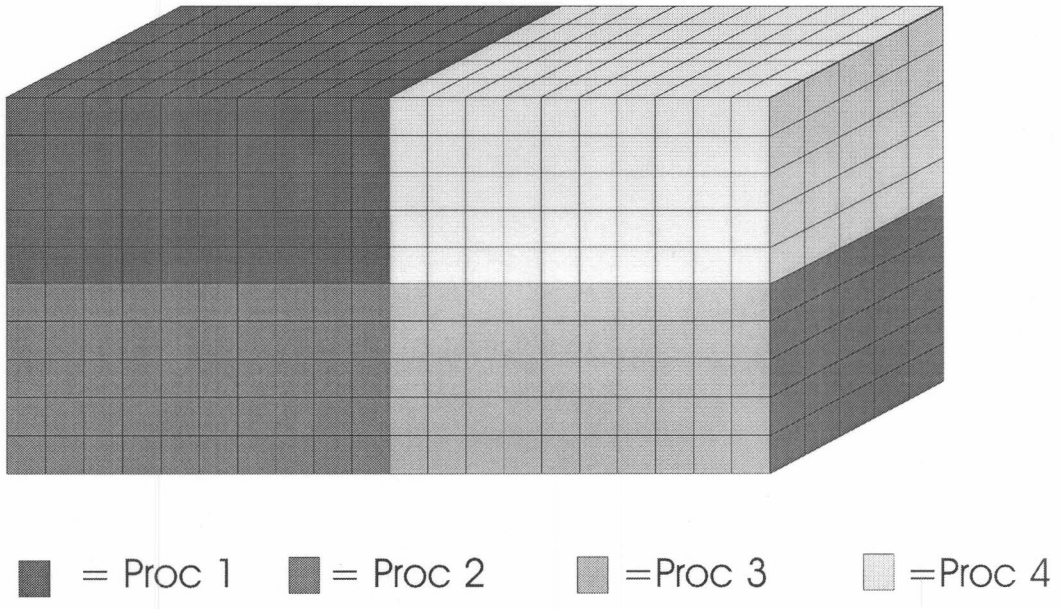


Figura 3.1.b – Partición en subdominios bidimensional

Por todas estas razones, se decidió utilizar la partición en subdominios a lo largo de un eje. Es conveniente realizar la partición a lo largo del eje que es de esperar sea el más largo, para reducir la superficie limite entre procesos y por consiguiente la proporción entre planos de intercambio (que se tendrán que comunicar) y planos de cálculo local.

Dado que la mayoría de las simulaciones de ECD que se corren en el laboratorio, se realizan sobre planos alargados sobre su eje "Y", se eligió este eje para la partición en subdominios.

3.1.1 Implicancias de la partición en subdominios en los resultados del algoritmo paralelo

La partición en subdominios, en su versión "tradicional", esto es, tal como se explicó en la sección anterior, provoca indefectiblemente que los resultados obtenidos con una ejecución paralela, tengan ciertas diferencias con los resultados obtenidos en una ejecución serial equivalente. A continuación, se explica la causa principal de estas diferencias.

En el caso serial, una vez calculados los primeros elementos, éstos son utilizados como entrada para el cálculo de los elementos vecinos subsiguientes. De esta forma, solo los primeros elementos son calculados en base a una totalidad de valores iniciales, mientras que los restantes se calculan en base a algunos valores iniciales y a otros valores calculados recientemente.

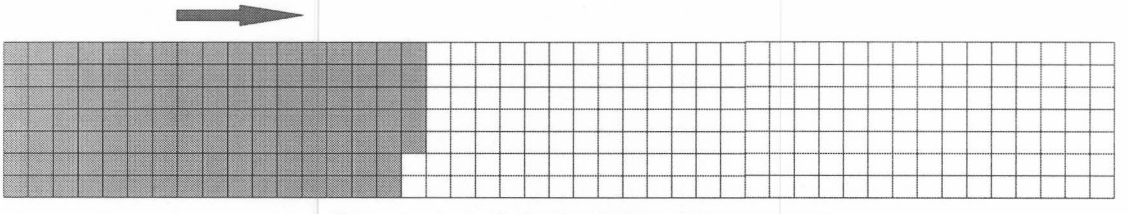


Figura 3.1.1.a – Orden de cálculo en el algoritmo serial

Sin embargo, en la versión paralela, esto ocurre parcialmente, puesto que, cada subproceso, no cuenta con la información de su vecino previo al momento de calcular los elementos de la malla. Por esto, en cada subproceso, el cálculo se inicia e base a valores iniciales, y luego a los valores recientemente calculados. Esto produce "cortes" (gaps) en el cálculo de la malla en el algoritmo paralelo. En cada uno de los gaps sucede lo mismo que en el comienzo del cálculo en el caso serial.

En adelante, nos referiremos a "gaps" como el corte que se produce en el órden lexicográfico de cálculo en el algoritmo paralelo.

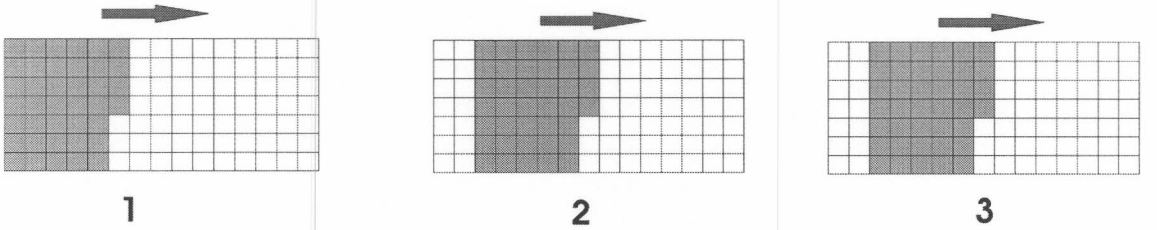


Fig. 2.1.1.b – Orden de cálculo en los distintos procesos del algoritmo paralelo

Por ende, el resultado que se obtendrá con el algoritmo paralelo, será el equivalente a correr el algoritmo serial con un orden de cálculo diferente al lexicográfico, en donde se cuentan con *n* gaps, donde *n* = número de procesos, tal como muestra la siguiente figura:

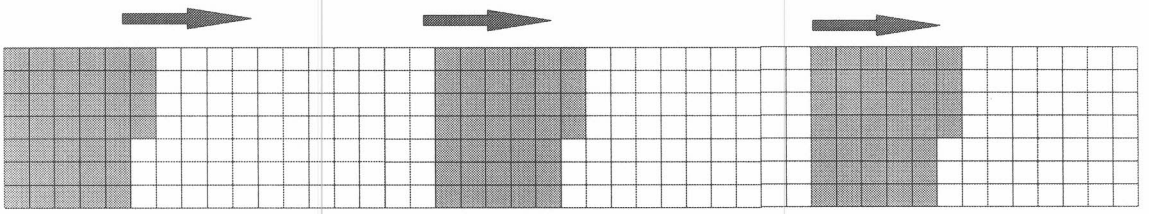


Fig. 3.1.1.c – Orden de cálculo del algoritmo paralelo, llevado a un ejemplo serial.

Este fenómeno, es propio de la paralelización de este tipo de algoritmos. La cantidad de cortes es directamente proporcional a la cantidad de subprocesos en los que se dividió el dominio. Estos "gaps" pueden hacer necesarias algunas iteraciones adicionales (con respecto al algoritmo serial) para llegar al criterio de convergencia. Es por esto, que el resultado de una simulación realizada con un algoritmo paralelo, tiene diferencias con respecto a la misma simulación, realizada en un algoritmo serial.

Aunque los resultados no son exactamente iguales, es de esperar que las diferencias sean lo suficientemente pequeñas como para considerarse despreciables. Esto se constató en las observaciones empíricas y se ve con mayor detalle en el capítulo 5. Es importante tener en cuenta que estas diferencias son más notorias cuanto menor es la cantidad de intervalos por subdominio. Así, si se reparte un dominio pequeño entre muchos procesos, la cantidad de "gaps" puede resultar demasiado grande en relación a la superficie de cálculo, pudiendo alterar los resultados hasta llegar inclusive al caso de no convergencia. Es por esto, que antes de realizar una simulación en paralelo, se debe tener en cuenta la cantidad de elementos totales de la malla y la cantidad de procesos en los que se desea dividir dicha superficie de cálculo.

Una propuesta para evitar completamente los "gaps", es rediseñar el algoritmo paralelo para que funcione como un "pipeline", el cual, operaría del siguiente modo:

Iteración 1 (inicial): Solo el primer proceso es el que calcula la primer iteración. El resto de los procesos permanecen ociosos.

Iteración 2: Una vez terminada la iteración 1, el proceso 1 le envía los valores de borde al proceso 2, quien puede comenzar a calcular la iteración 1 con los mismos valores que tendría el algoritmo serial. Mientras tanto, el proceso 1 puede comenzar la segunda iteración.

Iteraciones sucesivas: En cada iteración, serán más los procesos que participen del cálculo, a medida que se "llene el pipe". Llamaremos "modo pipeline" a este modo de calcular en un algoritmo paralelo, y "modo tradicional" al modo tradicional explicado en la sección anterior.

En el "modo pipeline", todos los procesos contarán, al momento de calcular, con los valores vecinos equivalentes a los de una ejecución serial y, por lo tanto, el resultado de una corrida paralela en "modo pipeline" dará resultados exactamente iguales a los de la misma corrida con el algoritmo serial.

Si bien esta propuesta puede ser de interés para ciertos casos, en los que la extremada precisión sea un requisito, la performance de un algoritmo paralelo en "modo pipeline" es menor que la de un algoritmo en "modo Standard" debido a los tiempos de llenado y vaciado de pipeline, en los que hay procesadores ociosos.
Como se verá más adelante, en el capítulo 5, las diferencias en los resultados entre las corridas seriales y las paralelas en el "modo Standard" son lo suficientemente pequeñas para el problema de ECD en 3D, por este motivo, no se implementó un algoritmo en "modo pipeline"

3.2 Intercambio de valores entre procesos vecinos

Una vez establecido el método de partición en subdominios, es necesario definir la manera en la que los procesos intercambian los valores de borde con sus vecinos. Las opciones son:

- Memoria compartida
- Almacenamiento compartido
- Intercambio de mensajes por red (Message Passing)

Memoria compartida:
El intercambio de mensajes por memoria compartida, requiere contar con hardware que posibilite el acceso común de los procesadores al área de memoria. Es el método más eficiente en cuanto a tiempos de intercambio, pero no es implementable en un cluster Beowulf, puesto que se cuenta con computadoras separadas, con memoria local en cada una de ellas.

Almacenamiento compartido:
El intercambio de valores mediante almacenamiento compartido, requiere que todos los procesadores tengan acceso común a un medio de almacenamiento (ej: discos). Esto es factible, pero es poco eficiente en general, y en un cluster Beowulf en particular, puesto que requiere de escrituras en algún medio de almacenamiento que pueda ser leído por todos los procesos (ej: NFS o similar), y esto requiere a su vez, la transmisión de los datos por la red. Esto sumado a las bajas tasas de transferencia de los medios de almacenamiento, hacen descartar esta opción.

Intercambio de mensajes (Message Passing):
Este método consiste en el envío y recepción de mensajes entre los procesadores, conteniendo los datos necesarios para el proceso. Es el más adecuado y el más ampliamente utilizado sobre clusters Beowulf, y fue el utilizado en este trabajo.

Para implementar este mecanismo de intercambio de valores, se utilizó el Standard MPI (Message Passing Interface) que provee funciones para el envío y recepción de mensajes de proceso a proceso, así como operaciones colectivas y de cálculo de tiempos. MPI cuenta con soporte para C++ que es el lenguaje en el que estaba implementada la solución serial previa y con el que se programó este algoritmo paralela.

3.3 Diseño e implementación inicial del programa paralelo

Al momento de diseñar el programa paralelo, se pensó en independizarlo todo lo posible de la implementación original del algoritmo serial, tratando de aplicar un concepto que pueda ser utilizado en la paralelización de cualquier otra implementación de un algoritmo que resuelva problemas de cálculo numérico mediante métodos iterativos. Es por esto que, dentro de cada nodo, el programa paralelo se comporta de una manera muy similar al programa serial, pero realiza los cálculos sobre un subconjunto del dominio. El diseño inicial del programa paralelo sigue los siguientes pasos:

1) Etapa de sincronización inicial:

Los procesos se sincronizan y se reparten el dominio de cálculo. Cada uno de ellos reserva la cantidad de memoria necesaria para las matrices y buffers, según el tamaño de intervalo que le haya sido asignado. Los procesos de los bordes son identificados y se inicializa una cantidad predefinida de planos de intercambio. Estos planos son los que se intercambian con los vecinos "laterales" y los que son utilizados para poder calcular los valores de los bordes de cada uno de los subintervalos locales a cada proceso.

2) Iteración de cálculo:

Aquí se recorren todos los elementos de todas las matrices, calculando en el paso i, los valores $M^{i}_{(xyz)}$ en base a los valores $M^{i-1}_{(xyz)}$ o a los vecinos de $M^{i}_{(xyz)}$ si ya fueron calculados.

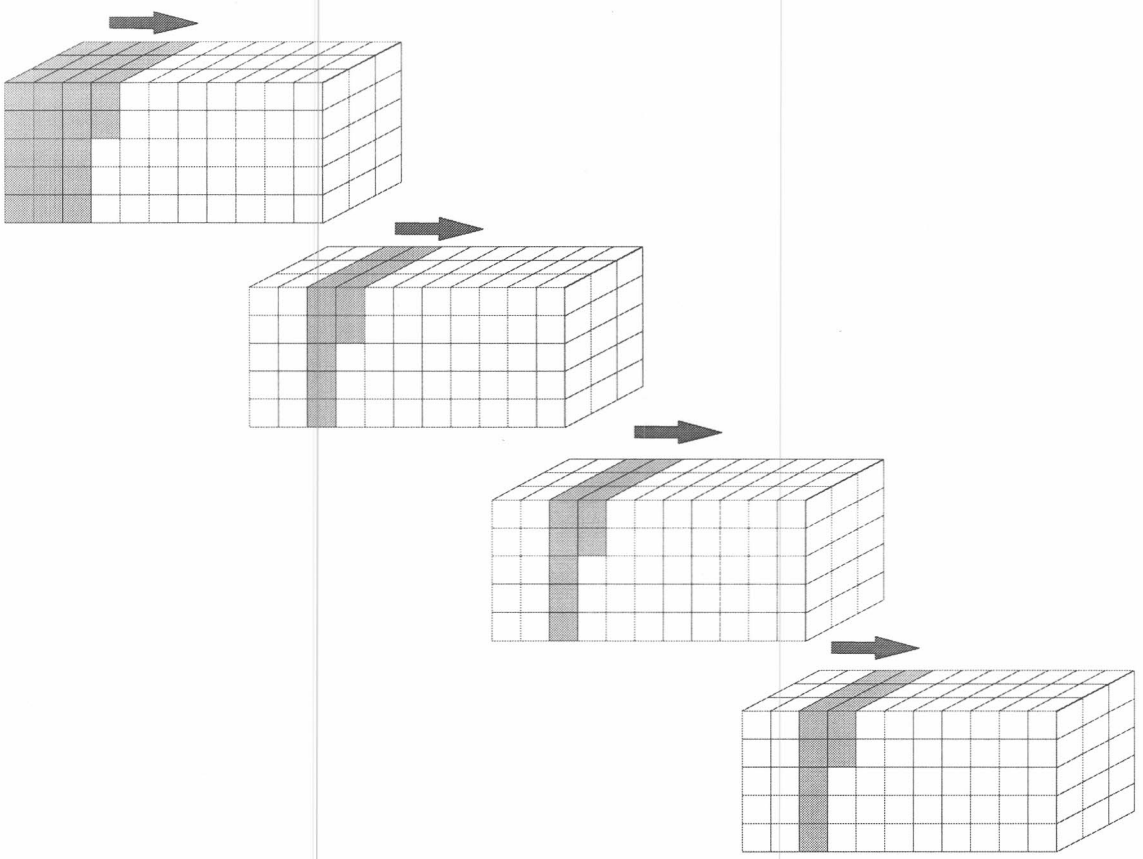


Fig. 3.3.a – Orden de cálculo lexicográfico en el algoritmo paralelo

Cada proceso, comienza el cálculo de los elementos de las diferentes matrices. Para ello, recorre todos los elementos de cada matriz, en orden lexicográfico (esto es, iterando en orden, desde el elemento 0 al n para cada uno de los ejes) calculando en la iteración i, los valores $M^{i}_{(xyz)}$ en base a los valores $M^{i-1}_{(xyz)}$ o a los vecinos de $M^{i}_{(xyz)}$ si ya fueron calculados.

3) Intercambio de mensajes

Una vez que se obtuvieron todos los valores para la iteración i para todas las matrices, todos los procesos realizan un intercambio de los planos de Intercambio con sus respectivos vecinos. Desde el proceso 0 al proceso *"p-1"*, todos envían sus planos de borde derecho al vecino de la derecha, y desde el proceso 1 al proceso *p*, los nodos envían los planos del borde izquierdo a su vecino de la izquierda. De esta forma, cada proceso cuenta con los datos necesarios para continuar con la iteración siguiente.

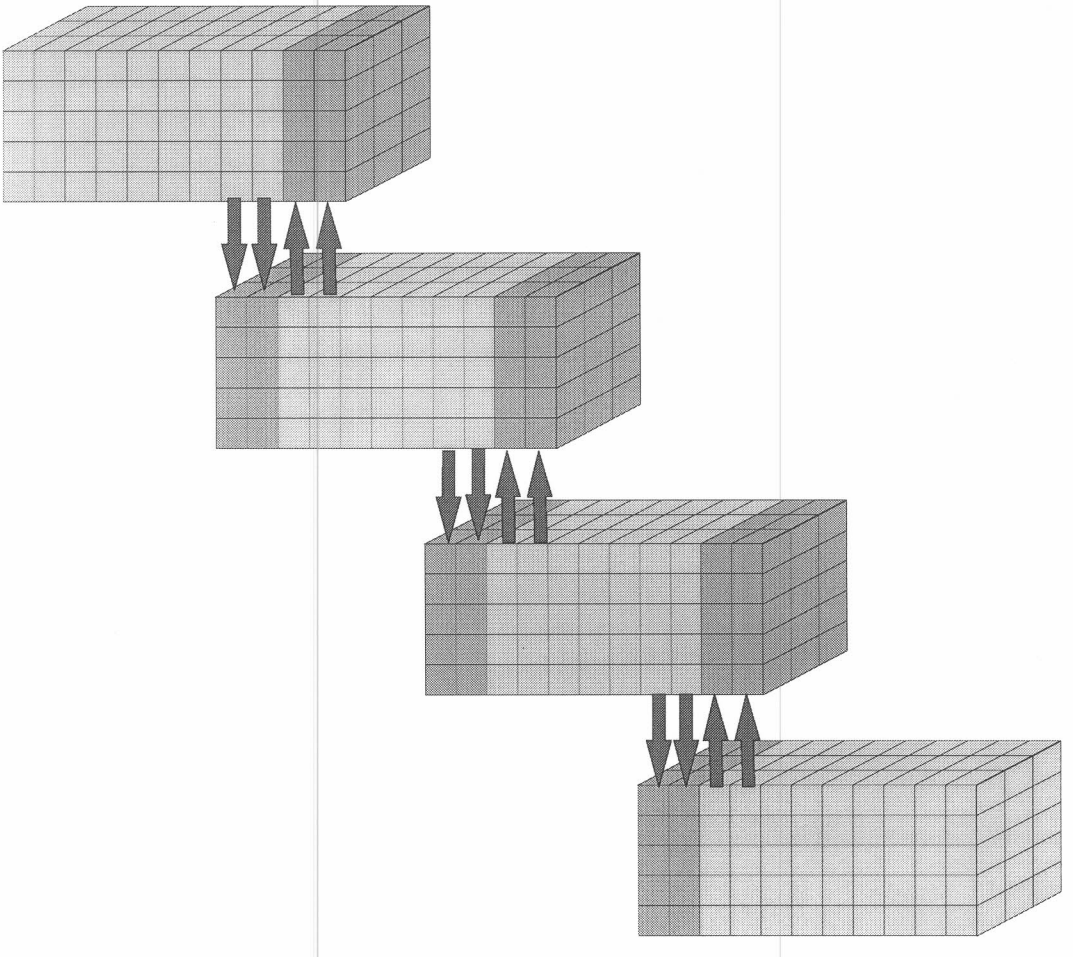


Fig. 3.3.b – Intercambio de mensajes en el algoritmo paralelo

En primer instancia, cada uno de los nodos, excepto el último, intercambia sus valores de borde con su vecino de la derecha, realizando primero el envío y luego la recepción. Posteriormente, cada uno de los nodos, con excepción del primero (nodo 0) intercambia sus valores de borde con su vecino de la izquierda, realizando primero la recepción y luego el envío. De esta forma, el envío del anteúltimo proceso se

corresponderá con la recepción del último, lo cual posibilita su desbloqueo, para dar lugar a la recepción del anteúltimo que desbloquea el envío del antepenúltimo y así sucesivamente. El envío del proceso n será correspondido con la recepción del proceso *n+1* y permitirá su desbloqueo, para dar lugar a la recepción del envío del proceso *n-1*. Una vez que esta cadena alcanza al nodo 0, se habrá completado el anillo lógico.

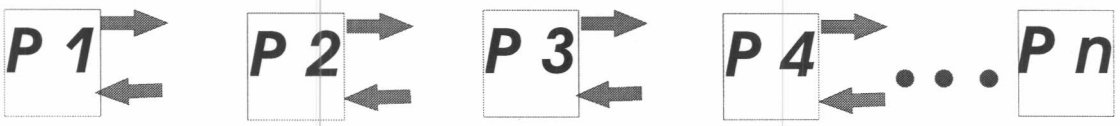


Fig. 3.3.c - Primer paso del anillo, en donde todos los procesos envían sus datos a sus vecinos de la derecha, y esperan la recepción de los datos del mismo vecino, con excepción del proceso "n".

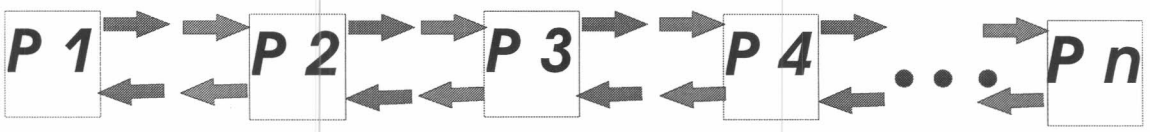


Fig. 3.3.d -Segundo paso del anillo, en donde los procesos inician los envíos y recepciones contra los procesos de la izquierda

4) Verificación de condiciones de fin

Luego del intercambio de valores, es necesario verificar la condición de fin del ciclo. En el algoritmo serial, se compara el error de convergencia con una cota mínima. Este error es la mayor diferencia entre cada valor A_i, y su valor anterior A_{i-1}

En el algoritmo paralelo, cada nodo calcula su error de convergencia local, y luego, mediante una operación de comunicación colectiva, se informa este valor a todos los nodos restantes. Finalmente, los nodos obtienen el error de convergencia global, seleccionando el mayor de los valores trasmitidos. Con este valor, se puede decidir si continuar con la siguiente iteración o avanzar al siguiente paso de tiempo.

5) Grabación de los resultados:

El programa serial original, graba los resultados en los archivos de salida, para luego visualizarlos. En la versión paralela, esto implica una complicación adicional, puesto que cada nodo contiene información parcial relativa al subdominio que se le asignó y solo es capaz de escribir resultados en base a dicha información. La solución a este problema es la siguiente:

Cada proceso escribe sus resultados locales, en base a los datos del subdominio que le corresponde. Para identificar los archivos de cada proceso, se agrega el ID de proceso al final del nombre del archivo. Se juntan los resultados en un lugar común, ya sea manualmente luego de la ejecución o directamente durante la misma, si todos los nodos escriben en una misma unidad compartida por algún sistema de archivos de red (Ej: NFS)

Se desarrolló una rutina de post-procesamiento, que, una vez terminado el cálculo, lee todos los fragmentos de resultado de las matrices escritas por cada uno de los nodos, los interpreta y los une en un único archivo por matriz, equivalente al obtenido por la

solución serial. De esta forma, se puede utilizar la misma utilidad de visualización de resultados que se utiliza para la solución serial, brindando transparencia en los resultados.

3.4 Algunas consideraciones del proceso de diseño e implementación

A lo largo del desarrollo de este trabajo de tesis, y mientras se realizaba la paralelización del algoritmo de simulación numérica de ECD en 3 dimensiones, se continuó trabajando en forma simultánea en el desarrollo del algoritmo serial para dicho proceso. Es así que periódicamente, fueron surgiendo cambios en la versión serial que debieron ser "mapeados" a la versión paralela. Algunos de estos cambios, tuvieron un impacto menor y fueron fáciles de implementar en el caso paralelo, como por ejemplo, los cambios en la rutina de cálculo propiamente dicha, pero otros cambios requirieron atención especial al momento de ser llevados a la versión paralela. Algunos de los cambios fueron los siguientes:

Cambio del formato de los archivos de salida, de ASCII a NetCDF:

Implicó, además del cambio en la salida de cada subproceso, el desarrollo de una rutina de post-procesamiento, utilizando el formato NetCDF que interpretara las submatrices de cada proceso para pegarlas en una matriz unificada

Agregado de nuevas matrices de cálculo:

Implicó nuevos intercambios de valores y nuevos lugares en los que verificar condiciones de borde (en donde se diferencian los procesos de los extremos)

Cambios en las condiciones de borde y en los "dedos" (máscara de bits) que atraviesan las matrices:

Estos casos requieren especial atención, cuando se trata de procesos de los bordes y cuando se cuenta con datos que traspasan los límites de un proceso (tal es el caso de los dedos)

Capítulo 4 – Performance y Optimización

La paralelización de un algoritmo ofrece por si misma una mejora en la performance del mismo, puesto que se divide la carga total de cálculo entre los procesadores involucrados que operan en paralelo. De este modo, se realiza más cálculo en menos tiempo. No obstante, existen diversos factores adicionales que pueden ayudar a mejorar mucho más la performance del algoritmo paralelo.

En este trabajo se abordaron dos técnicas adicionales tendientes a mejorar la performance de la implementación paralela de la simulación numérica de ECD en 3 dimensiones. Aunque estas mejoras fueron aplicadas a este algoritmo en particular, son lo suficientemente generales como para ser implementadas en otros algoritmos similares de simulación numérica.

Por un lado, se diseñó una estrategia de balance semi-dinámico de carga al inicio de la ejecución del algoritmo paralelo, para lograr un mejor aprovechamiento del poder de cálculo en los clusters heterogéneos, sin necesidad de conocer de antemano las características del mismo. Por otro lado, se experimentó con diferentes modos de intercambio de mensajes entre los procesadores del cluster Beowulf, con la intención de lograr el solapamiento de los tiempos de comunicación y de cálculo, y reducir de esta manera los tiempos totales de ejecución.

4.1 Balance de carga

A continuación, se discuten diferentes problemáticas relativas al balance de carga entre los procesadores de un cluster Beowulf y las soluciones a las mismas que fueron empleadas en este trabajo.

4.1.1 Problemática de los clusters heterogéneos

Los clusters Beowulf, por su capacidad de escalamiento temporal, suelen contar con procesadores (computadoras) heterogéneas de diferentes capacidades de cálculo. A modo de ejemplo, podemos citar el cluster "Speedy Gonzalez" del LSC, que cuenta con 16 computadoras con la siguiente configuración:

- 4 nodos con procesador Pentium II 733 MHz
- 6 nodos con procesador AMD Athlon 950 Mhz
- 4 nodos con procesador AMD Athlon 1200 Mhz
- 2 nodos con procesador AMD Athlon 900 Mhz

Como el algoritmo paralelo requiere que los procesos se sincronicen una vez terminada cada iteración de cálculo, intercambiando los valores con sus vecinos, puede suceder que algunos procesadores más rápidos terminen antes de realizar sus cálculos locales, y queden en consecuencia ociosos a la espera de que los nuevos datos de entrada sean generados por el procesador vecino (más lento), y poder así continuar con la siguiente iteración. Si todos los procesadores, independientemente de su capacidad de cálculo, tienen

igual cantidad de trabajo, entonces el cluster funcionará como si todos sus procesadores integrantes tuvieran el poder de cálculo del nodo más lento. Esto puede resultar tan poco eficiente como se quiera, pudiendo llegar al punto de resultar más conveniente ejecutar el algoritmo serial en la computadora más rápida que ejecutar el algoritmo paralelo en el cluster completo. El siguiente ejemplo ilustra la problemática planteada:

Se cuenta con un cluster de 3 nodos, uno de ellos (A) con poder un poder de cálculo de 20 unidades y las otras 2 (B y C) con poder de cálculo de 5 unidades. Un programa serial, que tarda 5 unidades de tiempo corriendo en el nodo A, tardará 20 unidades de tiempo en el nodo B o en el C. Suponiendo que se obtuviera un algoritmo paralelo ideal, con speedup lineal, si se reparten los subdominios en cantidades iguales entre estos 3 nodos, el algoritmo paralelo tardará 20/3=6.66 unidades de tiempo en ser calculado, mientras que en la computadora A, el algoritmo serial tardaba 5 unidades de tiempo.

Queda en evidencia la necesidad de replantear la partición en subdominios para asignarle a cada procesador una cantidad de carga proporcional al su poder de cálculo.

4.1.2 Balance de carga semi-dinámico al comienzo de la ejecución

En este trabajo, se adoptó un método de balance de carga semidinámico al comienzo de la ejecución en paralelo. La idea del mismo es poder detectar a priori, el poder de cálculo de cada uno de los nodos, para poder asignar a cada uno de ellos una cantidad de subprocesos proporcional a dicho poder de cálculo, y solucionar el problema planteado en [4.1.1]

Para otorgar mayor flexibilidad al programa, se realizó una rutina independiente encargada del cálculo de performance de los diferentes nodos del cluster. En esta rutina, se realiza una pequeña "simulación" de los cálculos reales durante una cantidad predefinida de iteraciones y sobre un espacio de memoria creado dinámicamente a tal fin. Dicha simulación realiza las mismas operaciones que se necesitan en el cálculo real, pero no afecta al resultado de la corrida. La simulación es cronometrada en cada uno de los nodos, luego de lo cual, todos los nodos informan a los demás su valor de "poder de cálculo" obtenido, utilizando una operación de comunicación colectiva (MPI_Allgather). Con esta información, todos los nodos calculan la proporción de subdominios que le corresponderá a cada uno. Una vez obtenido este valor, los nodos inicializan sus matrices con los valores adecuados, según la cantidad de subintervalos que les correspondan.

En la figura [4.1.2.a] se muestra un ejemplo del resultado del particionamiento en subdominios sujeto al balance de carga semidinámico para 4 nodos de diferente capacidad de cálculo.

Se considera a este método "semi-dinámico" puesto que, por un lado, se ajusta dinámicamente a cualquier tipo de clusters sin necesidad de conocer a priori la composición del mismo ni el poder de cálculo de sus nodos, pero, por otro lado, una vez establecida la partición en subdominios, esta permanece constante hasta la finalización del cálculo.

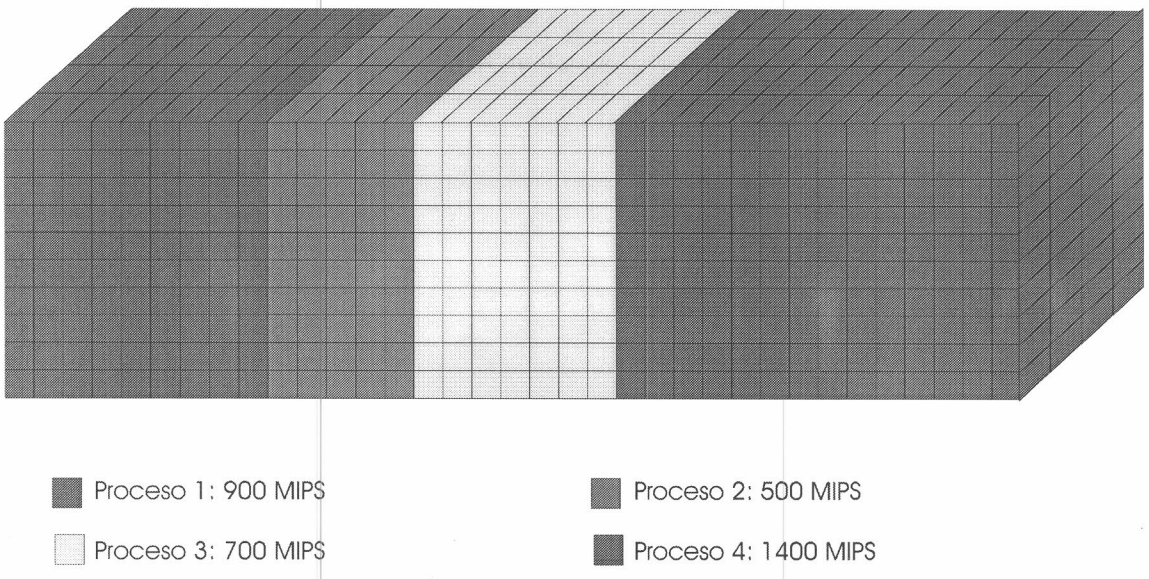


Fig. 4.1.2.a – Partición en subdominios según el balanceador de carga

4.2 Solapamiento de cálculo con envío de mensajes

Aquí se discutirá la motivación para implementar una solución que permita el solapamiento el tiempo entre el cálculo principal y las operaciones de transferencia de datos. Se evaluarán también opciones y modificaciones al algoritmo paralelo para tal fin.

4.2.1 – Motivación

El diseño inicial de la solución paralela al proceso de ECD en 3 dimensiones, requiere que, cada vez que se termina una iteración de cálculo, todos los procesos intercambien los valores de borde con sus vecinos, enviando los valores calculados localmente, y obteniendo los valores que los vecinos calcularon. La ejecución de la siguiente iteración en cada nodo no puede continuar hasta tanto se hayan enviado y recibido los datos de sus vecinos.

Aquí surgen varios aspectos que degradan la performance del algoritmo paralelo:

a) Diferencias en los tiempos de cálculo de cada nodo:

Puede suceder que, pese a haber balanceado los subdominios entre los nodos participantes, no todos ellos terminen la iteración en forma sincronizada. Esto provoca tiempos ociosos de espera por parte de los primeros procesadores en terminar la iteración de cálculo. Hay varias causas que influyen para que esto suceda:

Ejecución de otros programas en el nodo:
Esto incluye a las rutinas del sistema operativo, que pueden ser diferentes en cada nodo y llegar a interferir negativamente en la performance.

Diferencias en la cantidad de memoria:

Pueden provocar que un nodo a realice mayor cantidad de operaciones de paginación que los demás, con su consiguiente detrimento de performance.

El balance de carga no resultó óptimo:
Como la rutina de balance de carga no representa de forma exacta la totalidad del proceso de cálculo, la asignación de subdominios puede no ser la óptima

La cantidad de celdas a calcular dentro de la malla no es igual en todos los procesos
Las simulaciones de ECD incluyen "dedos" que son protuberancias generalmente en el extremo izquierdo de la malla de cálculo. Como las celdas ocupadas por "dedos" no se calculan (mantienen su valor constante a lo largo de la ejecución del algoritmo), los subprocesos que contengan estos "dedos" tendrán una cantidad de celdas a calcular menor a la supuesta por la rutina de balance de carga. Esto puede provocar que el proceso termine antes que sus vecinos y tenga tiempo ocioso de espera.

Diferencias en el tiempo de envío/recepción de mensajes:
Porque por ejemplo los procesos de los bordes intercambian la mitad de las veces, o porque haya diferencias en el medio de comunicación (placa de red, etc.). Estas diferencias de tiempo fuerzan a que un proceso vecino tenga que esperar para poder enviar o recibir su mensaje.

En todos estos casos, existe un tiempo ocioso en algunos procesadores que es el de esperar a que todos los nodos vecinos terminen su cálculo para realizar el intercambio de valores. Esto degrada la performance general de la corrida en paralelo.

b) Tiempo dedicado exclusivamente al envío / recepción de mensajes:

La transferencia física de los mensajes entre cada par de procesos, lleva un tiempo no despreciable. Este tiempo es proporcional al tamaño de los buffers de intercambio y se ve influenciado por el medio de comunicación (placa de red, cableado, Hub/Switch, etc.). Como las comunicaciones son bloqueantes, el cálculo se suspende durante el tiempo en el que se realizan las mismas, lo cual tiene un impacto negativo sobre la performance total del algoritmo.

Para evaluar este problema con mayor detalle, se incluyeron cronómetros en la implementación del algoritmo, que permiten medir los tiempos internos. Utilizando estos cronómetros, se separó la medición del tiempo de cálculo de la medición del tiempo empleado para el envío y recepción de mensajes y el tiempo de espera entre procesos.

Con la intención de apreciar cuán importante es el tiempo ocioso en las simulaciones de casos reales, se realizaron corridas del algoritmo paralelo con 3 tamaños de malla diferentes (40x100x40, 40x200x40 y 40x400x40), cada una de ellas con 2, 4, 6, 8, 10, 12 y 14 procesos. En cada caso, se midió el porcentaje de tiempo ocioso y se observaron los siguientes resultados:

	Tiempo ocioso							
Tamaño	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**
40x100x40	0.00%	0.65%	16.41%	34.78%	53.00%	64.15%	72.30%	78.41%
40x200x40	0.00%	0.43%	6.28%	18.41%	32.73%	45.06%	55.78%	63.96%
40x400x40	0.00%	2.18%	3.69%	8.84%	18.23%	28.12%	37.62%	46.03%

Tabla 4.2.1.a Porcentaje del tiempo de corrida destinado a transferencia de datos y espera entre procesos.

Como se puede apreciar, los tiempos de espera entre procesos sumados a los de envío y recepción de mensajes son realmente considerables, llegando a ser de más del 78% en el caso de mayor cantidad de procesos con menor tamaño de malla. Esto es causado por la desproporción entre tiempo interno de cálculo en cada uno de los nodos y tiempo de intercambio de mensajes.

En la figura 4.2.1 se observa claramente que, a medida que se aumenta la cantidad de procesos utilizada para resolver un problema de un tamaño de malla fijo, aumenta también el porcentaje de tiempo utilizado para la comunicación, puesto que cada proceso tiene una malla menor para calcular, y la misma cantidad de planos para intercambiar, por ende, la proporción entre superficie de cálculo y superficie de intercambio es menor. Esto último se aprecia al ver la reducción en los porcentajes de tiempo ocioso a medida que se amplía la malla de cálculo, sobre el eje de paralelización (eje Y).

Puesto que este fenómeno es el mayor responsable de la pérdida de performance en este tipo de algoritmos paralelos, surge la necesidad de rediseñar este último con el objetivo de permitir la superposición de tiempos de cálculo con tiempos de comunicación y reducir, así, los tiempos de espera y sincronización entre procesos.

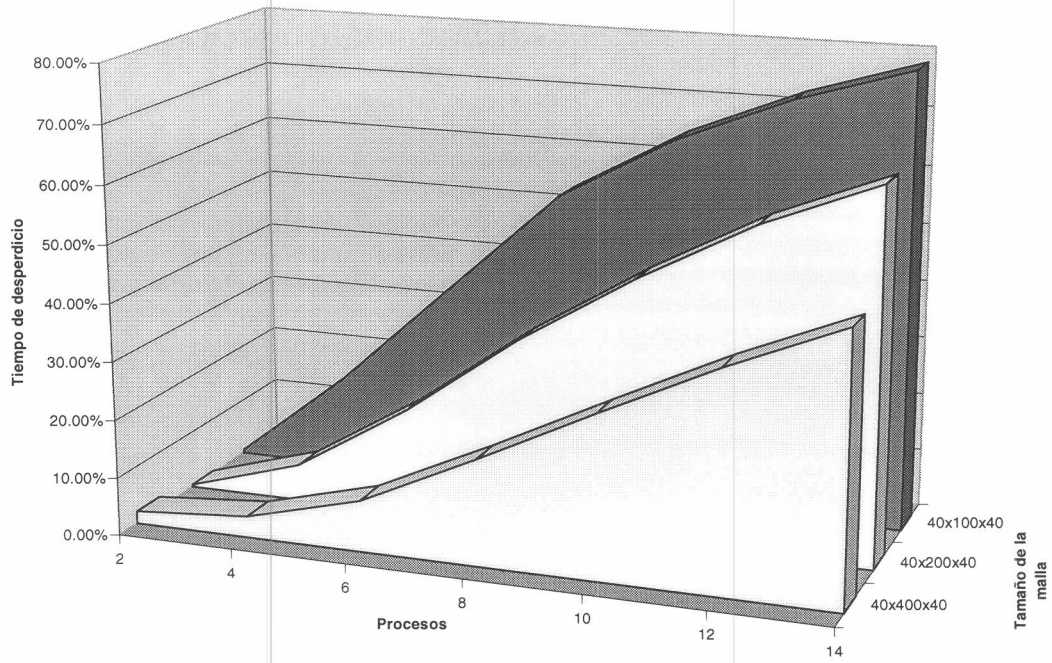


Fig. 4.2.1.b – Porcentaje del tiempo de ejecución utilizado para comunicación de datos

4.2.2 – Implementación de la modificación.

A raíz de lo expuesto en el punto anterior, se realizaron modificaciones al diseño original del algoritmo paralelo para permitir el solapamiento de las operaciones de comunicación con cálculo neto.

En primer lugar, se alteró el orden lexicográfico de cálculo, para calcular primero los valores de borde, y luego los valores restantes, como muestra la figura 4.2.2.a.

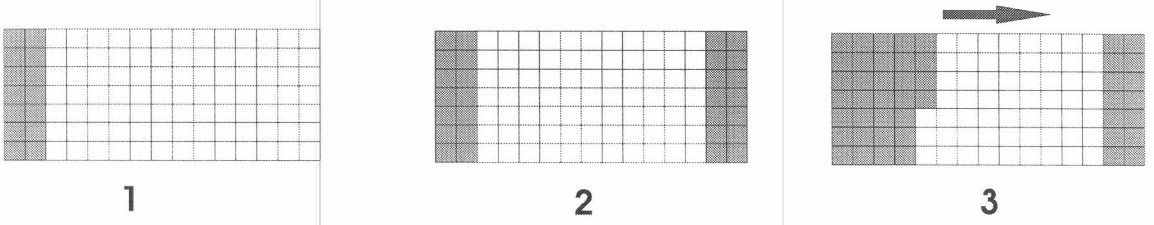


Fig. 4.2.2.a – Orden de cálculo "No lexicográfico"

A diferencia del diseño inicial, no se comienza a calcular desde el primer plano, sino que se calculan primero los últimos *n* planos de intercambio, y luego se continúa con los primeros. De esta forma, luego de calcular las primeras "2 x INTERC" columnas, ya se dispone de los valores necesarios para enviar a los vecinos (INTERC = Cantidad de planos de intercambio).

La segunda modificación entonces consiste en adelantar el punto de intercambio de mensajes, para que sea realizado apenas están disponibles los valores de borde. Una vez realizado dicho intercambio, se continúa con el cálculo de las columnas restantes.
Como es de esperar que el tamaño de la malla sea grande, al momento de realizar el intercambio de planos de cada proceso con sus procesos vecinos, aún restará realizar la mayor parte de cálculo local, de este modo, si se utilizan comunicaciones no bloqueantes, es posible solapar el tiempo de transferencia de datos de estas comunicaciones con parte del tiempo de cálculo de las columnas restantes, puesto que ambas tareas se realizan al mismo tiempo. Todo el tiempo que se utiliza para calcular estas columnas, puede ser solapado con el tiempo de envío/ recepción de datos, siempre que la implementación de las librerías, el hardware y el sistema operativo lo permitan.

Al terminar de calcular todos los planos, será necesario esperar a que todas las comunicaciones finalicen (si es que no lo hicieron), para poder entonces continuar con la siguiente iteración.

Esta variante al diseño original reduce el tiempo que un nodo espera para obtener los datos de sus vecinos, pudiendo llegar a eliminarlo por completo. Las pruebas realizadas con las modificaciones propuestas en este capítulo arrojaron resultados muy satisfactorios que son evaluados con mayor detalle en el capítulo 5.

Dado que se realiza una alteración en el orden lexicográfico de cálculo, es de esperar que se observen algunas diferencias en los valores de los resultados de las corridas, en comparación con las calculadas con orden lexicográfico. Para evaluar el impacto real de estas diferencias en los resultados de las corridas, y decidir la validez del algoritmo con orden de cálculo "no lexicográfico", se realizaron diversas mediciones que son evaluadas con mayor detalle en el capítulo 5. Estas mediciones mostraron que las diferencias son lo suficientemente pequeñas como para no influir de manera apreciable en el resultado final.

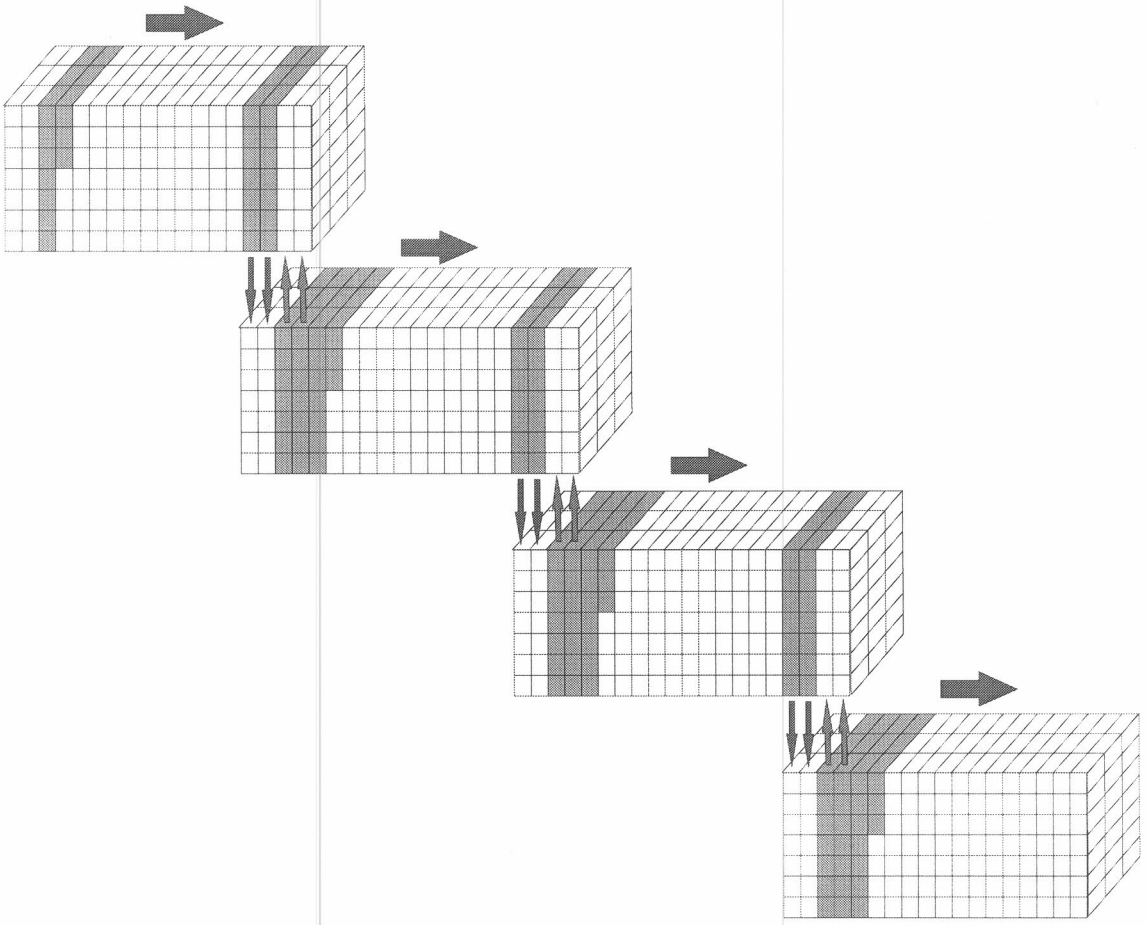


Fig. 4.2.2-b – Intercambio de vecinos en el orden "no lexicográfico" de cálculo

4.2.2.1 Problemas encontrados y sus soluciones

Los primeros resultados con operaciones no bloqueantes fueron desalentadores. La performance de simulaciones con comunicaciones no bloqueantes, dependía del tamaño de los buffers de intercambio, resultando notablemente peor que la performance de la misma simulación con comunicaciones bloqueantes. Luego de investigar el fenómeno, se descubrió que se debía a un problema en el manejo de las comunicaciones no bloqueantes, tanto de MPICH como de LAM-MPI. El problema reside en que ambas implementaciones de MPI son "single threaded" y funcionan de la siguiente manera:

Cuando se inicia un envío no bloqueante, estas implementaciones de MPI llenan los buffers TCP y derivan la tarea de envío/recepción a las capas inferiores, luego, el control pasa al algoritmo principal. Mientras tanto, los datos en el buffer se transmiten, pero una vez vaciado este buffer, al no haber un thread encargado de monitorearlo, el mismo no se vuelve a llenar. Esto tiene como resultado que todas las comunicaciones queden "en suspenso" , y se completen en el punto de sincronización entre procesos , antes de pasar a la siguiente iteración. Resulta entonces que no se aprovecha la posibilidad de solapamiento entre los tiempos de ejecución y de comunicación y en vez de mejorar la performance, se empeora. Es por este motivo que se abordaron dos alternativas:

La primera alternativa consistió en obtener una licencia temporal de una implementación comercial de MPI denominada SCALI, que emula threads sobre Linux y permite el progreso de las comunicaciones no bloqueantes en segundo plano.

La segunda alternativa consistió en investigar una manera de "emparchar" el problema del llenado de buffers en MPICH. Este parche consiste en lo siguiente:
Periódicamente, dentro del ciclo de cálculo, se realizan llamadas a una función de testeo de MPI (MPI_Test) con la única finalidad de pasar el "control" del programa a la rutina de MPI. Con esto, se le da la oportunidad a MPI de que vuelva a llenar los buffers vacíos de TCP, para darle continuidad a la transferencia en segundo plano. El problema ahora es determinar la cantidad óptima de llamadas a MPI_Test dentro del ciclo de iteraciones, puesto que demasiadas llamadas generarían overhead innecesario, mientras que muy pocas llamadas no darían tiempo a MPI de mantener los buffers de TCP llenos. Por otro lado, la cantidad óptima de llamadas a MPI_Test, varía según el tamaño del problema (y por ende de los buffers de intercambio). Por esto, fue necesario encontrar una manera óptima y al mismo tiempo dinámica de realizar las llamadas a MPI_Test.

Luego de experimentar al respecto, se encontró que la mejor forma de realizar esto fue la siguiente:

Como el tamaño de los buffers de intercambio, es proporcional al eje X y al eje Z , pero no al eje Y, se alteró el orden del ciclo, para que la iteración sobre el eje Y sea la más interna de las 3. De esta forma, se puede colocar la llamada a MPI_Test en el segundo nivel de anidación del ciclo, logrando que se realice X*Z veces.

```
Para i desde 1 hasta X
     Para j desde 1 hasta Z
          MPI_Test(...)
               Para k desde 1 hasta Y
                    Cálculo()
               Fin para
     Fin para
Fin para
```

Pseudocódigo 4.2.2.1.a – Muestra cómo se implementó el parche para permitir en MPICH el progreso de las comunicaciones no bloqueantes en segundo plano.

Este método demostró ser el más adecuado, puesto que realiza una cantidad de llamadas a MPI_Test proporcional al tamaño de los buffers de intercambio, para cualquier tamaño de dominio de la simulación. Además, las llamadas están lo suficientemente espaciadas como para permitir el envío de datos en segundo plano entre cada una de ellas. La aplicación de las llamadas a MPI_Test dentro del ciclo de cálculo, mejoró la performance de la aplicación con comunicaciones no bloqueantes, pero no logró que dicha performance sea mejor que la obtenida mediante la implementación de comunicaciones bloqueantes, como era de esperar. Este segundo escollo, puso de manifiesto que aún restaba un problema por solucionar, puesto que la teoría indicaba que las comunicaciones no bloqueantes debían permitir valores de performance muy superiores.

Para avanzar un nivel más en la investigación de este inconveniente, se realizaron mediciones de los tiempos de cada operación de envío y recepción, tanto para las comunicaciones bloqueantes (MPI_Send y MPI_Recv) como para las no bloqueantes (MPI_Isend y MPI_Irecv). Aumentando paulatinamente el tamaño del buffer de intercambio.

Los resultados obtenidos fueron los siguientes (en milisegundos):

	Operación			
Buffer	MPI_Send	MPI_Recv	MPI_Isend	MPI_Irecv
21kb	0.200	3.800	1.300	0.005
42kb	2.200	5.500	3.000	0.006
84kb	5.200	21.000	7.300	0.008
168kb	13.000	35.000	0.011	0.004
336kb	39.400	48.200	0.030	0.025
672kb	74.000	86.300	0.090	0.025
1.31mb	122.000	145.000	0.080	0.025

Tabla 4.2.2.1.b – Medición te tiempos de operaciones de envío y recepción bloqueantes y no bloqueantes

Estos tiempos fueron cronometrados utilizando la función MPI_Wtime() de MPI. Algunos valores son tan pequeños que pueden no ser correctamente medidos. De todas formas, los resultados muestran algo realmente importante y es que, si bien las operaciones de comunicación bloqueante se comportan de manera esperada (los tiempos aumentan al aumentar el tamaño del buffer), con la operación de MPI_Isend ocurre algo muy curioso. Aunque dicha operación debería tardar un tiempo fijo (el de lanzamiento de la operación), la misma se comporta de manera muy extraña, tardando para tamaños de buffer menores a los 100k más tiempo que la totalidad de una operación no bloqueante, lo cual no es para nada lo esperado. Al aumentar el tamaño de buffer a más de 168kb, los tiempos de lanzamiento de la operación pasan a ser los esperados. (Recordar que la transferencia de datos propiamente dicha se realiza en segundo plano y no está cronometrada).

Con estos datos a la vista, se revisó el algoritmo paralelo. En el mismo, existía un buffer de intercambio por cada una de las 16 matrices. Esto se reemplazó por un buffer único, en donde los datos de cada matriz se diferencian por el offset dentro del mismo. Con esto, se multiplica por 16 el tamaño del buffer lo cual permite que las comunicaciones no bloqueantes se comporten de manera correcta. Esta última modificación, permitió finalmente obtener valores de performance mucho mejores para el caso de la utilización de comunicaciones no bloqueantes, que son explicadas con mayor detalle en el capítulo 5.

Una vez solucionados los problemas inherentes a los tiempos de la comunicación no bloqueante, restaba entonces elegir la implementación de MPI más conveniente entre las disponibles en el cluster "Speedy". Las implementaciones a considerar fueron:

- MPICH (Gratuita y Open Source)
- LAM-MPI (Gratuita y Open Source)
- SCALI (Comercial)

Se realizaron entonces algunas pruebas para decidir cuál de ellas es la más conveniente. Para ello, se ejecutaron 6 corridas de una simulación con 5 procesos, utilizando ambas implementaciones empleando tanto comunicaciones bloqueantes como no bloqueantes y utilizando el parche descripto anteriormente para el caso de MPICH y LAM-MPI con comunicaciones no bloqueantes. Los resultados fueron los siguientes (en minutos de ejecución):

	Scali	MPICH	LAM-MPI
Bloqueante	84.15	83.80	84.97
No Bloqueante	56.70	50.30	52.05

Tabla 4.2.2.1.c – Comparación entre las implementaciones Scali, MPICH y LAM-MPI

A la vista de estos resultados, se decidió utilizar la implementación MPICH, tanto para los casos de comunicaciones bloqueantes como para los de comunicaciones no bloqueantes, utilizando en estos últimos el parche descripto anteriormente. Con las ideas planteadas en este capítulo, se realizaron diversas pruebas para comprobar que los resultados prácticos acompañan a los planteos teóricos. Los resultados de estas pruebas, así como su evaluación detallada se incluyen en el capítulo 5.

Capítulo 5 – Análisis de resultados

En esta sección se muestran y analizan los resultados obtenidos. Se hace un análisis particular sobre las diferencias obtenidas entre las ejecuciones seriales y paralelas y el error inherente al paralelismo. Se incluyen algunas imágenes de los resultados y por último se evalúan y analizan los tiempos de las diferentes corridas de test para sacar conclusiones sobre la performance del algoritmo paralelo en comparación con el algoritmo serial.

5.1 Comparación de resultados

Se verá a continuación, un análisis comparativo entre los resultados obtenidos mediante la versión serial original del algoritmo ECD y las diferentes variantes de las corridas paralelas, así como una comparación entre los diferentes métodos de cálculo de la versión paralela, tendientes a demostrar la aceptabilidad de la versión paralela en sus diferentes versiones.

5.1.1 Diferencias entre las corridas serial y paralela

Como se explicó anteriormente, la paralelización "tradicional" de un algoritmo de cálculo numérico que utilice métodos iterativos para la resolución de los problemas, produce resultados levemente diferentes a los obtenidos por un algoritmo serial, a menos que se emplee una solución de tipo "pipeline" como la propuesta en el capítulo 3.1. En esta sección, se realizaron comparaciones entre los resultados obtenidos con la versión serial original, y diferentes corridas de la versión paralela con 2, 4 y 8 procesos para determinar la validez de la solución paralela. Para ello, se realizaron corridas con 3 tamaños diferentes de malla, luego, mediante la utilidad "ncdiff" se obtuvo para cada resultado paralelo, una matriz cuyos valores consistían en las diferencias entre los resultados de la corrida paralela con respecto al resultado serial. Si bien estos valores podrían ser utilizados como parámetro, para que la comparación tenga sentido realmente, consideramos que es necesario medir las diferencias no por su valor absoluto, sino como porcentajes de variación respecto de la versión serial. Es por esto que se realizó un cálculo elemento a elemento de cada una de las matrices, para determinar en cada caso el porcentaje de variación del resultado paralelo con respecto al resultado serial. Como valor final para la comparación, se tomó el promedio de los valores absolutos de los porcentajes de variación. Dichos valores están expuestos para cada una de las matrices de cálculo en las tablas siguientes:

Aniones	Procesos		
Dim Y	2	4	8
32	0.004 %	0.130%	0.462%
96	0.000%	0.005%	0.094%
128	0.000%	0.002%	0.017%

Potencial	Procesos		
Dim Y	2	4	8
32	0.564%	8.930%	32.646%
96	0.032%	0.077%	0.796%
128	0.000%	0.000%	0.005%

Psi	Procesos		
Dim Y	2	4	8
32	1.026%	6.581%	74.188%
96	0.702%	1.654%	3.586%
128	0.230%	0.494%	1.428%

Velocidad	Procesos		
Dim Y	2	4	8
32	0.841%	10.740%	35.823%
96	1.808%	3.526%	5.111%
128	1.104%	1.348%	2.421%

Tablas 5.1.1.a – Porcentaje de diferencia entre los resultados seriales y paralelos

Los resultados de las comparaciones, muestran varias cosas: Por un lado, dado un tamaño de malla fijo, a medida que se aumenta la cantidad de procesadores, el porcentaje de variación aumenta. Esto, creemos que se debe a la mayor cantidad de "gaps" en el cálculo de la malla dentro de cada iteración. Del mismo modo, se observa que, para los casos de tamaño más pequeño, con la mayor cantidad de procesadores, las variaciones son realmente grandes (35%, 74%, etc.). Estos valores decrecen muy notablemente al aumentar el tamaño de la malla sobre el eje Y. Este resultado muestra que una ejecución en paralelo puede dar resultados extremadamente diferentes a la ejecución serial si no se asigna una cantidad mínima de intervalos a cada proceso (en este caso 8), según la dimensión de la malla de cálculo en el eje de partición (en este caso, eje "y")

Se ve también que las matrices escalares (aniones, potencial) mostraron menores variaciones que las vectoriales (psi, velocidad) . Por último, se ve que para tamaños de malla lo suficientemente grandes, en la dimensión de la partición (en este caso, el eje "y") , los porcentajes de variación son realmente pequeños, lo cual demuestra que los resultados de una ejecución en paralelo son confiables siempre que se mantenga una proporción entre la superficie de cálculo y la cantidad de procesos involucrados.

Los casos anteriores contemplan una visión "global" de las diferencias de resultados entre las corridas serial y paralela. Resulta interesante poder ver más puntualmente cuáles son los lugares, dentro de la malla de cálculo, en donde se observan las mayores diferencias. También resulta interesante observar el comportamiento de estas diferencias, a medida que el cálculo va avanzando pasos de tiempo en la simulación, para tener una mejor idea de la validez de la solución paralela en el resultado final. Es por esto que se tomó un ejemplo de comparación entre un resultado serial y uno paralelo (en este caso de 3 procesos) . En este ejemplo, utilizando la utilidad "NCdiff" se obtuvieron las diferencias entre ambos casos y se graficaron utilizando "Vis5d" [23]. Los gráficos muestran en color rojo las zonas en donde las diferencias son mayores, en verde las intermedias y en azul las menores. Las zonas negras representan diferencia nula. Para visualizar mejor los gráficos, se rotó el prisma, de manera que el eje Y sea el vertical.

La primer serie de imágenes (Fig. 5.1.1.b) muestra las diferencias entre las corridas serial y paralela de la matriz "PSI" (una de las componentes de la función de corriente) en 4 pasos de tiempo sucesivos. En estas imágenes se ve claramente que las diferencias más significativas están en el plano de intercambio entre los vecinos, para luego ir disminuyendo hacia el interior del dominio de cada subproceso. Creemos que estas diferencias se deben a los "gaps" en el cálculo paralelo, motivo por el cual, como se vió anteriormente, para un tamaño de malla fijo, las diferencias son mayores al aumentar la cantidad de procesos que intervienen en el cálculo de la misma.

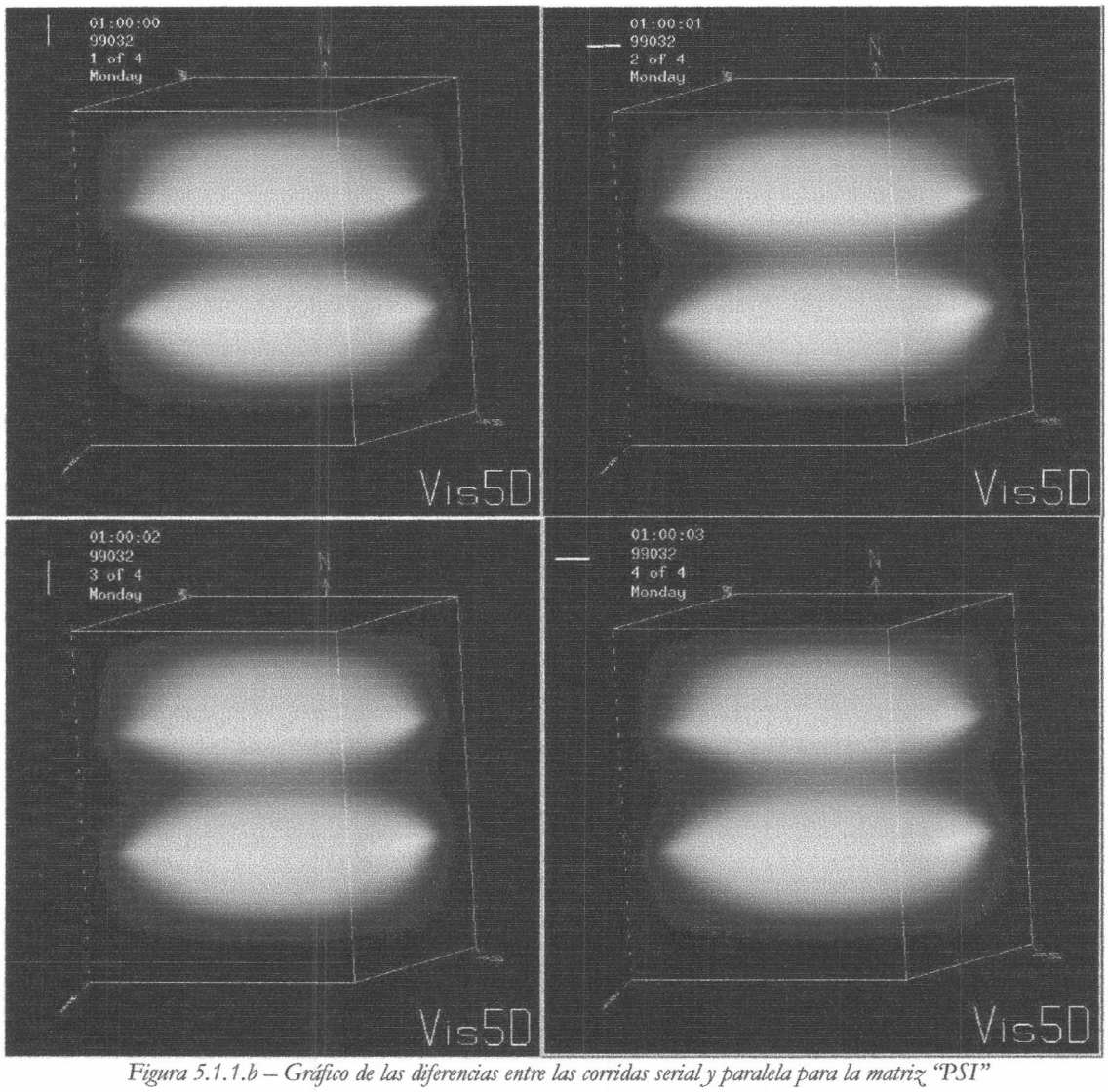


Figura 5.1.1.b – Gráfico de las diferencias entre las corridas serial y paralela para la matriz "PSI"

En este caso, el módulo de la diferencia se mantiene aproximadamente constante a lo largo del tiempo. Creemos que esto se debe al tipo de cálculo que se realiza en esa matriz, con lo cual estamos en presencia de un error inherente a la paralelización por división en subdominios.

La segunda serie de gráficos corresponde a la matriz "A" (concentración de aniones), con una vista lateral del prisma, en 5 pasos de tiempo sucesivos.. En estos gráficos se puede apreciar que, las principales diferencias se encuentran en las zonas cercanas a la máscara de bits (los "dedos"), esto creemos que se debe a que es esa la zona en donde son más marcados los cambios en los valores. A diferencia del gráfico anterior, se observa en este caso que, a medida que avanza el tiempo, las diferencias se van "dispersando" a lo largo de la malla de cálculo, con un valor menor (desaparecen las zonas de color rojo y aparecen zonas verdes y azules). Se observa entonces que el comportamiento de las diferentes matrices no es siempre igual

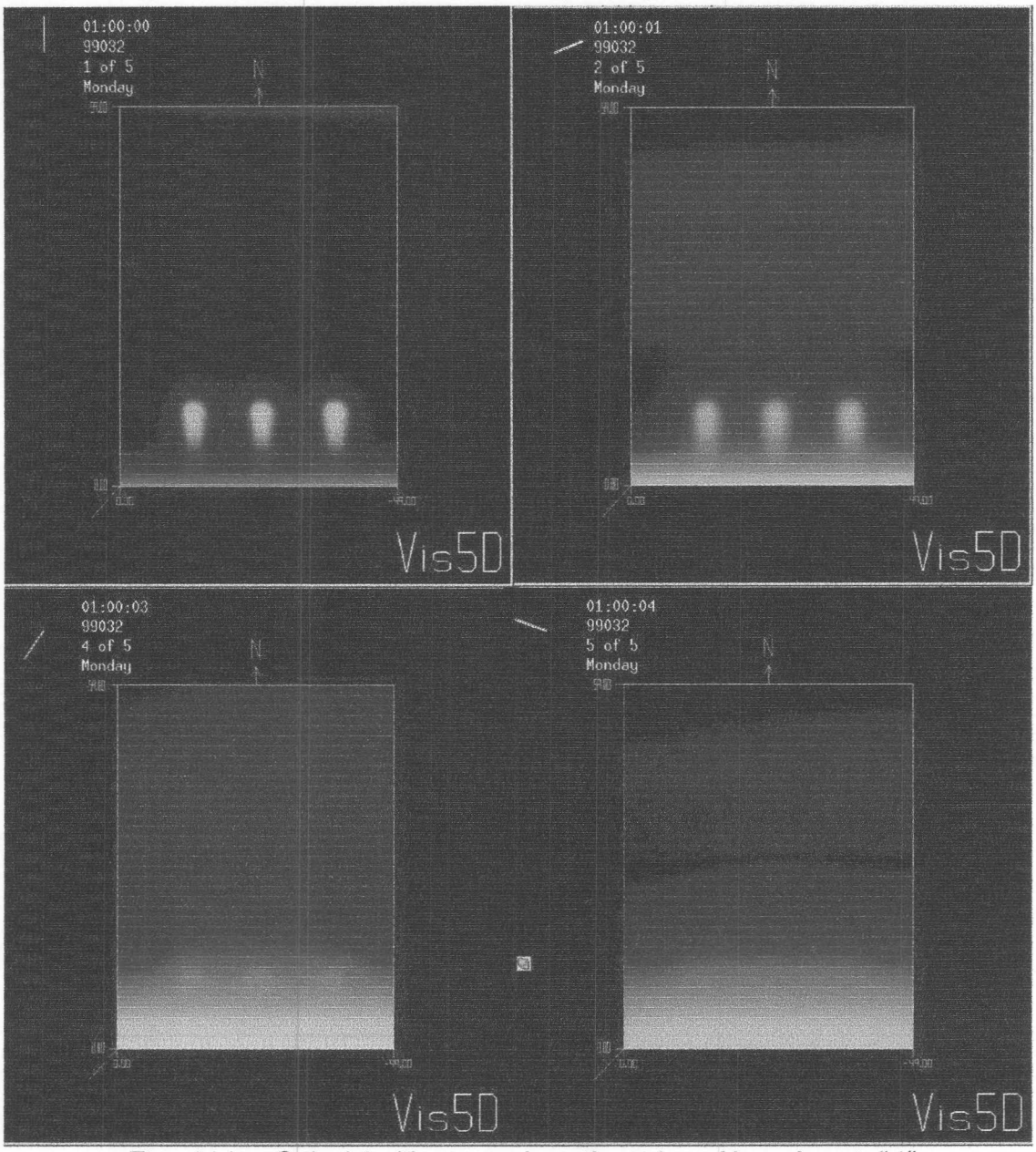


Figura 5.1.1.c – Gráfico de las diferencias entre las corridas serial y paralela para la matriz "A"

A modo de conclusión, se puede decir que, en algunas matrices, las diferencias más apreciables se encuentran en las zonas de intercambio, mientras que en otras, se encuentran en las zonas cercanas a los bits de la máscara ("dedos"). En este último caso, las diferencias se dispersan a medida que avanza el tiempo, posiblemente porque la solución alcanza un estado estacionario en el cual la diferencia entre dos valores sucesivos es muy pequeña.

En todos estos gráficos, la escala de colores se armó tomando cero como valor mínimo el cero (negro) y la máxima diferencia como valor máximo (rojo), lo cual muestra la disposición de las diferencias dentro de la malla de cálculo. Como se mostró anteriormente, estos valores son lo suficientemente pequeños, en cuanto al porcentaje de variación, como para darle validez a los resultados del algoritmo paralelo.

5.1.2 Diferencias entre corridas con orden de cálculo lexicográfico y con orden no-lexicográfico

Como se explicó anteriormente en el capitulo 3, en un algoritmo secuencial de cálculo numérico que sigue un orden lexicográfico de cálculo, una vez calculados los primeros valores de la malla, estos sirven de entrada para el cálculo de los siguientes. De esta forma se acelera la convergencia. Al paralelizar este tipo de algoritmos, se introducen "gaps" en cada subdominio. En estos "gaps", como los valores del proceso vecino no son conocidos hasta que se realice el intercambio de valores, estos no pueden ser usado como entrada para el cálculo de los siguientes.

El cambio de orden lexicográfico de cálculo propuesto en el capítulo 3 para permitir solapamiento entre envío de mensajes y operaciones de cálculo local, introduce "gaps" adicionales dentro de cada subproceso, puesto que se calcula primero la parte final del intervalo, cuando aún no se calcularon los valores previos. Por este motivo, es de esperar que este cambio impacte en la velocidad de convergencia, puesto que, como son menos los casos en los que se cuenta con vecinos recientes para calcular un elemento, el algoritmo necesitará más iteraciones para alcanzar el criterio de convergencia.

Para verificar la medida de este impacto, se realizaron comparaciones entre corridas con orden lexicográfico y con orden no-lexicográfico. Estas comparaciones consisten en identificar, para una cantidad determinada de pasos de tiempo (en este caso, los primeros 5) la cantidad de iteraciones necesarias para avanzar al siguiente paso de tiempo. Una vez obtenidos estos valores, se sumaron y se calculó el porcentaje de variación.

Los resultados obtenidos fueron los siguientes:

Y=50	2 Proc		4 Proc		8 Proc	
	Lex	*No Lex*	*Lex*	*No Lex*	*Lex*	*No Lex*
1	2548	2553	2562	2595	2588	2669
2	1719	1722	1730	1755	1749	1812
3	892	899	901	913	903	950
4	810	826	802	825	813	855
5	655	669	596	612	614	641
Tot	*6624*	*6669*	*6591*	*6700*	*6667*	*6927*
Dif	**+0.68%**		**+1.65%**		**+3.90%**	

Y=100	2 Proc		4 Proc		8 Proc	
Tempo	*Lex*	*No Lex*	*Lex*	*No Lex*	*Lex*	*No Lex*
1	2587	2587	2585	2586	2567	2580
2	2027	2028	2023	2026	2005	2020
3	563	564	560	562	557	560
4	428	428	427	428	279	286
5	414	414	416	420	104	104
Tot	*6019*	*6021*	*6011*	*6022*	*5512*	*5550*
Dif	**+0.03%**		**+0.18%**		**+0.69%**	

Y=200	2 Proc		4 Proc		8 Proc	
	Lex	*No Lex*	*Lex*	*No Lex*	*Lex*	*No Lex*
1	2977	2980	2977	2981	2992	3011
2	2458	2460	2461	2461	2470	2489
3	1151	1149	1148	1148	1124	1131
4	766	765	753	755	695	716
5	807	806	509	516	501	510
Tot	*8159*	*8160*	*7848*	*7861*	*7782*	*7857*
Dif	**+0.01%**		**+0.17%**		**+0.96%**	

Tablas 5.1.2.a- Diferencias entre las corridas con orden de cálculo lexicográfico y no-lexicográfico.

Como era de esperar, las ejecuciones con orden de cálculo no-lexicográfico utilizaron algunas iteraciones más, pero la diferencia es muy pequeña. Esto motiva la utilización de este método en conjunción con comunicaciones no bloqueantes, (en entornos de SO y Hardware que lo permitan), para aumentar la performance.

Aquí también se observa que los casos con tamaños de malla más grande en el eje de partición (eje "Y") son los que tienen menor diferencia con respecto al orden lexicográfico. La razón de este comportamiento es la menor proporción entre "gaps" e intervalos de cálculo.

5.2 Gráficos de algunos resultados

A continuación se muestran los resultados de una simulación numérica paralela de un problema de ECD en 3 dimensiones. Cuando han transcurrido aproximadamente 70 segundos del inicio del experimento. Se utilizaron 14 nodos del cluster "Speedy Gonzalez" del LSC. La agregación dendrítica se simula con 3 "dedos" que sobresalen perpendicularmente al plano correspondiente al cátodo. La celda está descripta por una malla prismática de 80x500x80 elementos que fue distribuida entre los 14 nodos según la rutina de balance de carga. Los parámetros de esta simulación son los siguientes:

Reynolds = 0.0025
Peclet "A" = 2.4
Peclet "C" = 3.6
Migración "A" = 1.6
Migración "C" = 2.4
Poisson = 0.053
Grashoft eléctrico = 10^4
Grashoft gravitatorio = 10^4

La figura 5.2.1 muestra las líneas de contour de la función de corriente en un corte vertical paralelo al eje Y, en el centro de la celda que indican los rollos gravitoconvectivos, unos segundos antes de la colisión. Es de notar, que no existen discontinuidades a lo largo del eje Y, lo cual demuestra que la incidencia de los "gaps" en la paralelización es efectivamente despreciable, como se vio anteriormente.

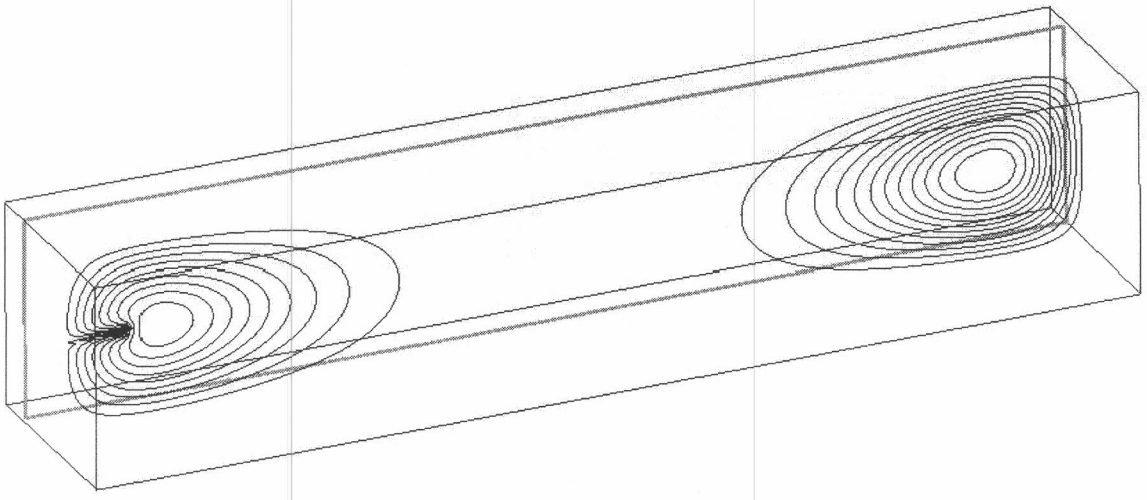


Fig. 5.2.1 – Lineas de contour de la función de corriente

La figura 5.2.2 muestra un corte vertical de la distribución de velocidades y una superficie de contour que envuelve a los dedos. Nuevamente se observa que la solución es contínua a lo largo de toda la malla

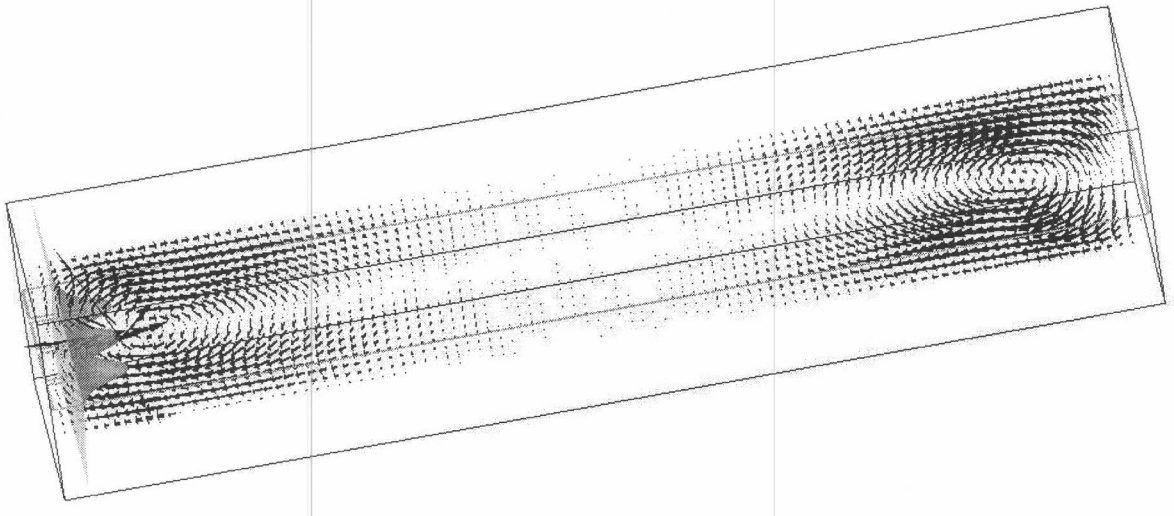


Fig. 5.2.2 – Lineas de contour del vector velocidad en un plano vertical y superficie de isoconcentración de cationes.

La figura 5.2.3 muestra trayectorias de partículas lanzadas en zonas cercanas al cátodo y al anodo. Se puede observar la característica compleja de estas trayectorias, que están fuertemente afectadas por la geometría impuesta.

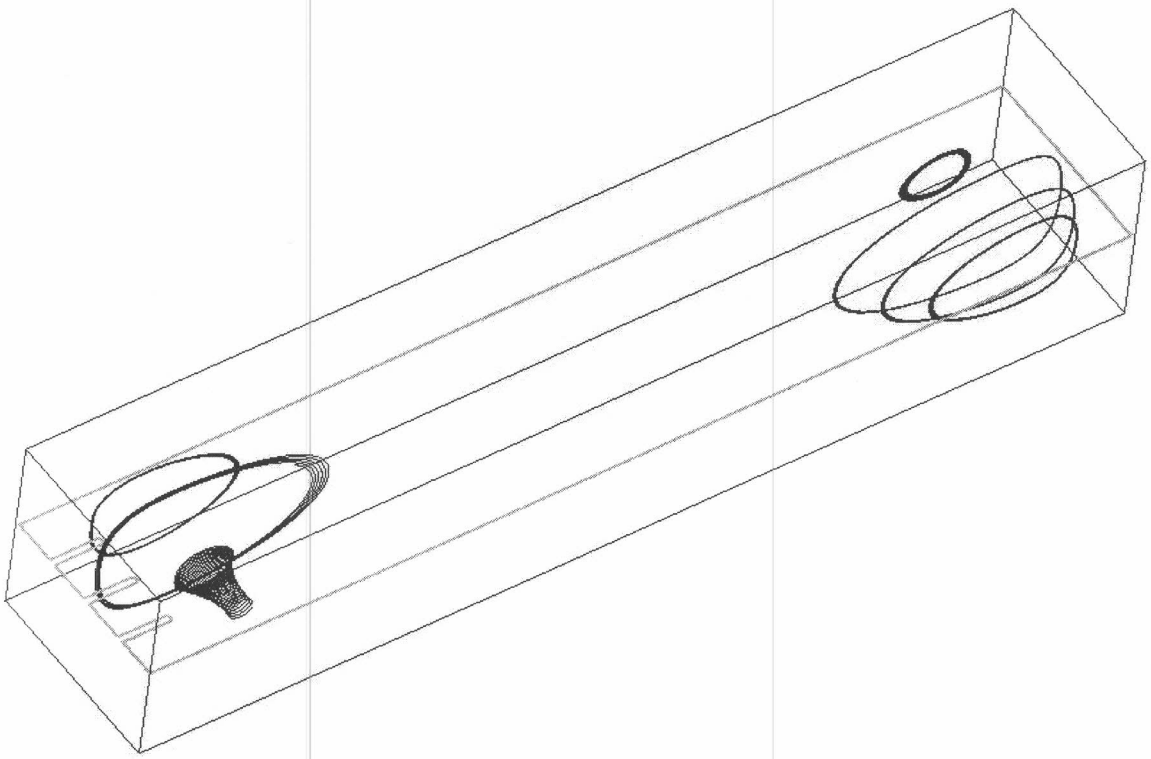


Fig. 5.2.3- Trayectorias de partículas lanzadas cerca de los electrodos.

La figura 5.2.4 muestra una isosuperficie del módulo del vector potencial de velocidades, que representan los rollos catódico y anódico. El rollo catódico tiene los 3 orificios correspondientes a los 3 "dedos" de la agregación, y el rollo anódico permanece sin perturbación

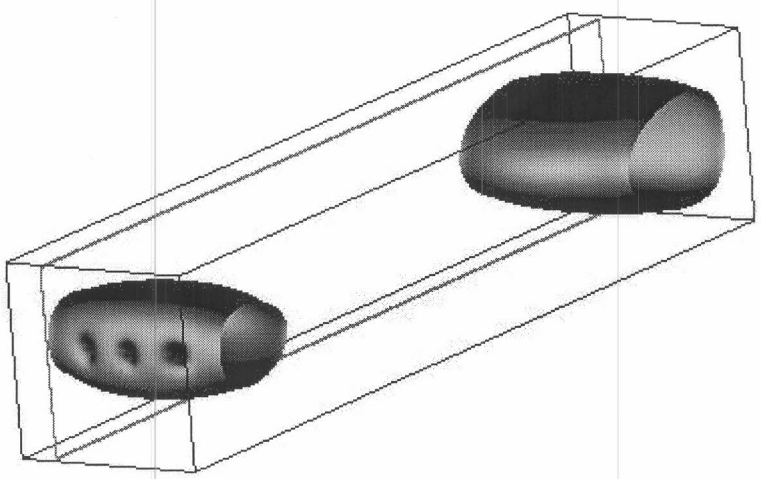


Fig 5.2.4 - isosuperficie del módulo del vector potencial de velocidades

Por último, la figura 5.2.5 muestra una isosuperficie del módulo del campo eléctrico rodeando los 3 dedos, y líneas de campo en el plano central horizontal. Las líneas perfectamente rectas, muestran nuevamente la continuidad de la solución, pese a los "gaps" de la paralelización.

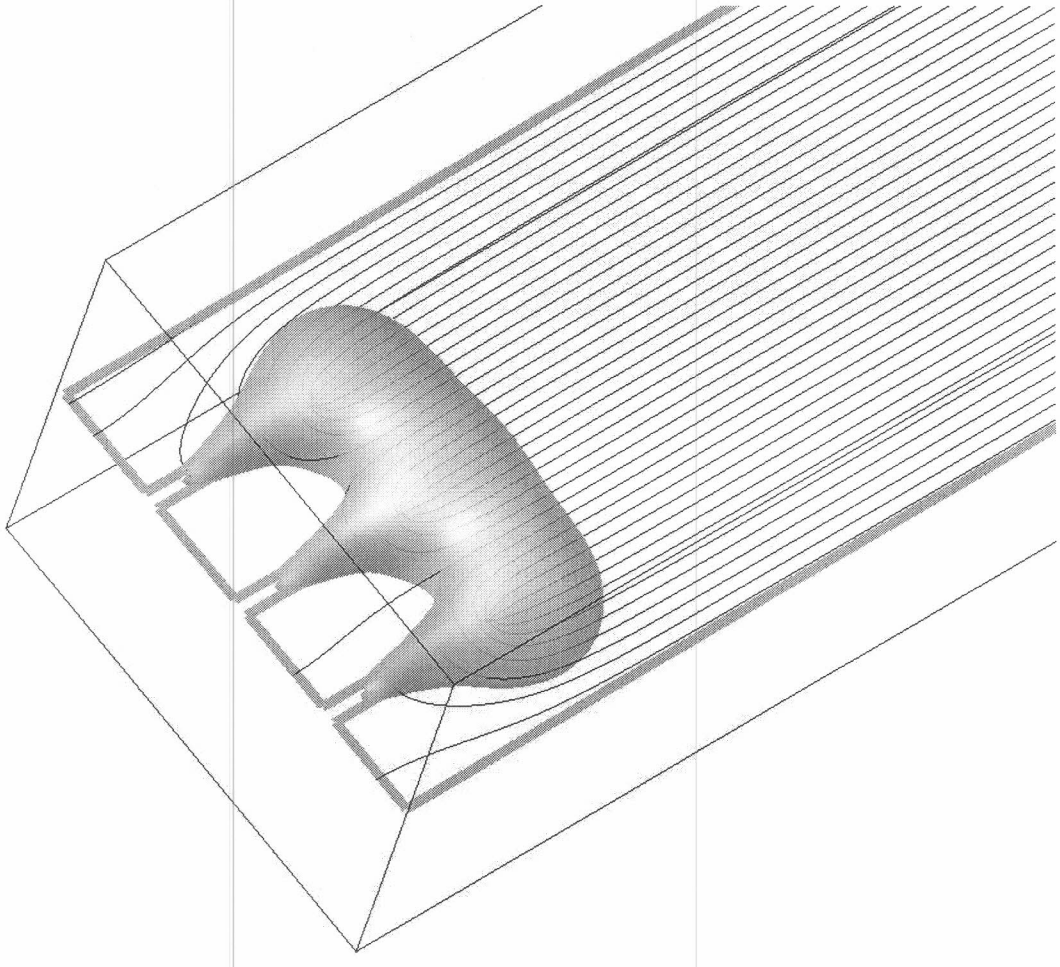


Fig. 5.2.5, Isosuperficie del módulo del campo eléctrico.

Si bien en estos ejemplos se han presentado mallas de tamaño mediano, se ha logrado correr simulaciones con mallas de 40x1600x100, lo que equivale aproximadamente a 1.3GB de memoria RAM

5.3 Análisis de tiempos y performance

En esta sección, se muestran los tiempos obtenidos de diferentes corridas seriales y paralelas en diferentes clusters. En base a las mismas, se realiza el cálculo de la eficiencia y, speedup del algoritmo paralelo para la simulación del proceso de ECD.

5.3.1 Tiempos de distintas corridas

Para tener una base sobre la cual calcular el speedup y la eficiencia del algoritmo, se realizaron 24 ejecuciones de test sobre 3 tamaños de malla diferente con 2, 4 , 6, 8, 10, 12 y 14 procesadores, y se efectuó también la misma ejecución en el algoritmo serial. Los tiempos empleados en la ejecución de estas corridas son los que se muestran a continuación.

Nota: El tamaño está expresado en cantidad de elementos por matriz, y los tiempos están expresados en minutos.

Tiempos de Corrida									
Tamaño	**Procesos**	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**
40x100x40	**Bloq**	345.73	174.00	103.40	88.35	91.94	96.44	104.00	114.40
	Nobloq	345.73	176.52	96.40	74.20	59.59	53.53	47.50	46.66
40x200x40	**Bloq**	606.55	304.60	161.80	123.90	112.70	110.40	114.30	120.20
	Nobloq	606.55	308.20	152.40	101.80	78.77	65.20	59.61	57.87
40x400x40	**Bloq**	974.65	498.20	253.00	178.20	149.00	135.60	130.20	129.00
	Nobloq	974.65	502.30	247.50	164.10	122.90	106.10	91.51	82.02

Tabla 5.3.1.a- Tiempos de corridas para diferentes tamaños y cantidad de procesos.

De los resultados de la tabla, se desprenden las siguientes observaciones:
En primer lugar, dado un tamaño de malla fijo y partiendo del tiempo registrado para la ejecución serial (1 solo proceso), puede verse la reducción en el tiempo de cálculo es cada vez menos marcada, a medida que se aumenta la cantidad de procesos involucrados, llegando a invertir la tendencia, y aumentar los tiempos de cálculo, si la cantidad de procesos es demasiado grande. Esto resulta más notorio en las corridas con tamaño de malla menor y en los casos de comunicación bloqueante. Se ve entonces que, siempre para un tamaño de malla fijo, hay un límite en la cantidad de procesadores que se pueden utilizar para lograr un incremento en la performance.

Dentro de los casos con comunicaciones bloqueantes, la corrida con tamaño de malla más pequeña (40x100x40) resulta ineficiente a partir de los 8 procesadores, mientras que la de tamaño 40x200x40 lo hace a partir de los 12 procesadores. Por último, la de tamaño 40x400x40 arroja los mismos valores de tiempo, tanto para 12 como para 14 procesos. Se ve entonces que, a medida que se aumenta el tamaño de la malla, se aumenta también la cantidad de procesadores en la que se puede particionar el problema, manteniendo una mejora en los tiempos de cálculo.

En cuanto a los resultados con comunicaciones no bloqueantes, se puede apreciar que la mejora obtenida en los tiempos de ejecución, es notable con respecto a los resultados con comunicaciones bloqueantes. Esta mejora es tal que, a diferencia de lo visto anteriormente, la partición en 14 procesos sigue resultando conveniente en los 3 casos testeados.

5.3.2 Performance obtenida con el algoritmo paralelo

Para tener una noción cierta de la performance del algoritmo paralelo, es necesario en primer lugar, realizar mediciones de speedup y eficiencia sin tomar en cuenta el balance

de carga. Lo ideal en estos casos es contar con un cluster homogéneo, pero al no disponer de una cantidad suficiente de computadoras similares en el cluster "Speedy Gonzalez", el método fue condicionar las ejecuciones paralelas para que siempre incluyan al menos 1 procesador del tipo más lento (en este caso, Pentium III 733Mz). De esta manera, todas los procesadores restantes se comportarán como el más lento, puesto que tendrán que esperar a este para poder intercambiar los mensajes y continuar con la ejecución paralela. Así pues, se simula un cluster virtual homogéneo dentro de uno real heterogéneo. Del mismo modo, las ejecuciones del algoritmo serial se realizaron sobre los nodos Pentium III 733MHz

Se realizaron entonces 24 ejecuciones incluyendo seriales y paralelas, con diferentes cantidades de procesos y cada una de ellas con diferentes tamaños de malla.

5.3.2.1 – Speedup

La primer medición que analizaremos será el speedup obtenido por el algoritmo.

Speedup									
Tamaño	Procesos	1	2	4	6	8	10	12	14
40x100x40	Bloq	1.00	1.99	3.34	3.91	3.76	3.58	3.32	3.02
	Nobloq	1.00	1.96	3.59	4.66	5.80	6.46	7.28	7.41
40x200x40	Bloq	1.00	1.99	3.75	4.90	5.38	5.49	5.31	5.05
	Nobloq	1.00	1.97	3.98	5.96	7.70	9.30	10.18	10.48
40x400x40	Bloq	1.00	1.96	3.85	5.47	6.54	7.19	7.49	7.56
	Nobloq	1.00	1.94	3.94	5.94	7.93	9.19	10.65	11.88

Tabla 5.3.2.1.a - Valores de Speedup

Para poder apreciar con mayor claridad los resultados de la tabla, se realizaron gráficos de speedup para los diferentes tamaños de malla, en los que se compara el desempeño de la implementación bloqueante y la no bloqueante, contra el speedup lineal teórico. Los gráficos son los siguientes:

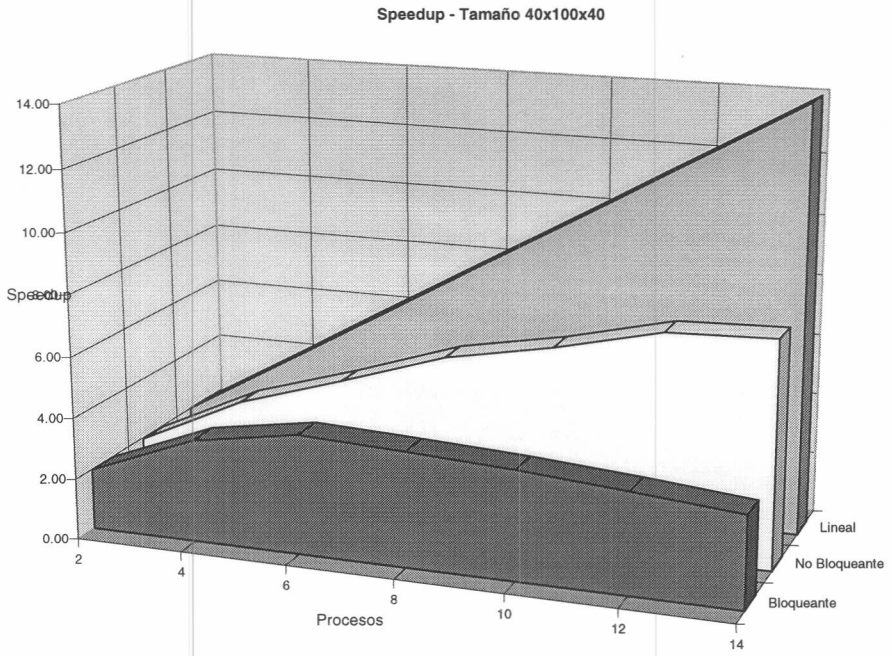


Fig. 5.3.2.1.b – Speedup del algoritmo paralelo para sobre una malla de tamaño 40x100x50 diferentes tamaños de malla y diferentes cantidades de procesadores involucrados

El gráfico muestra con claridad cómo el speedup se incrementa, para el caso bloqueante, hasta los 6 procesadores, para luego disminuir en forma casi lineal. Por el contrario, el caso no bloqueante el speedup se incrementa de forma mucho más marcada, lo cual demuestra el mejor aprovechamiento de los tiempos de cálculo, al solapar las operaciones de envío y recepción de mensajes. Se ve también que aún en el caso no bloqueante, el speedup está muy por debajo del máximo teórico (speedup lineal), lo cual se debe a que el tamaño de la malla es demasiado pequeño.

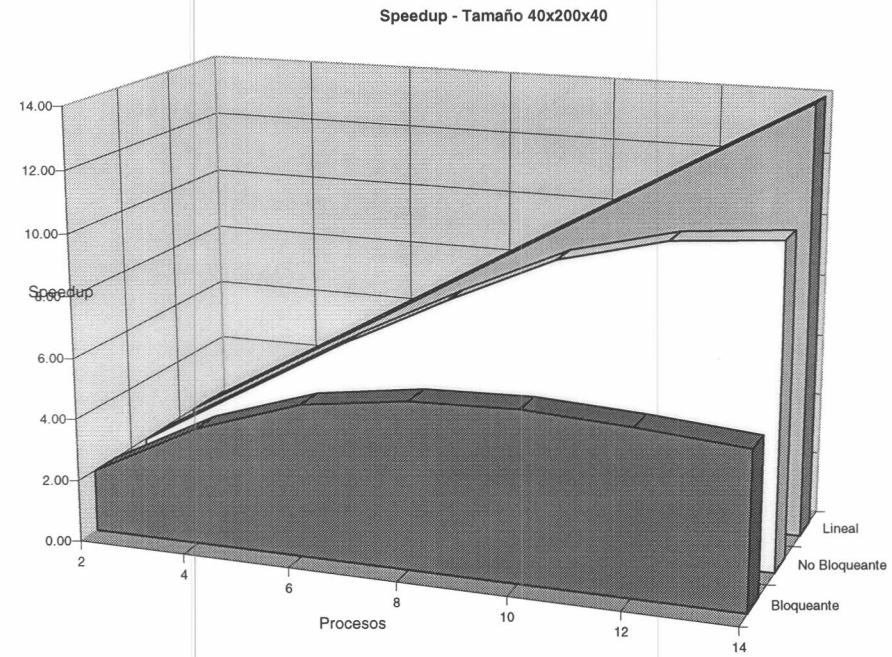


Fig. 5.3.2.1.c – Speedup del algoritmo paralelo para sobre una malla de tamaño 40x200x40 diferentes tamaños de malla y diferentes cantidades de procesadores involucrados

Al aumentar el tamaño de la malla a 50x200x50 elementos, se aprecia un incremento en el speedup en general. En el caso bloqueante se nota el incremento de speedup hasta los 10 procesos y el caso no bloqueante se acerca bastante más al speedup lineal.

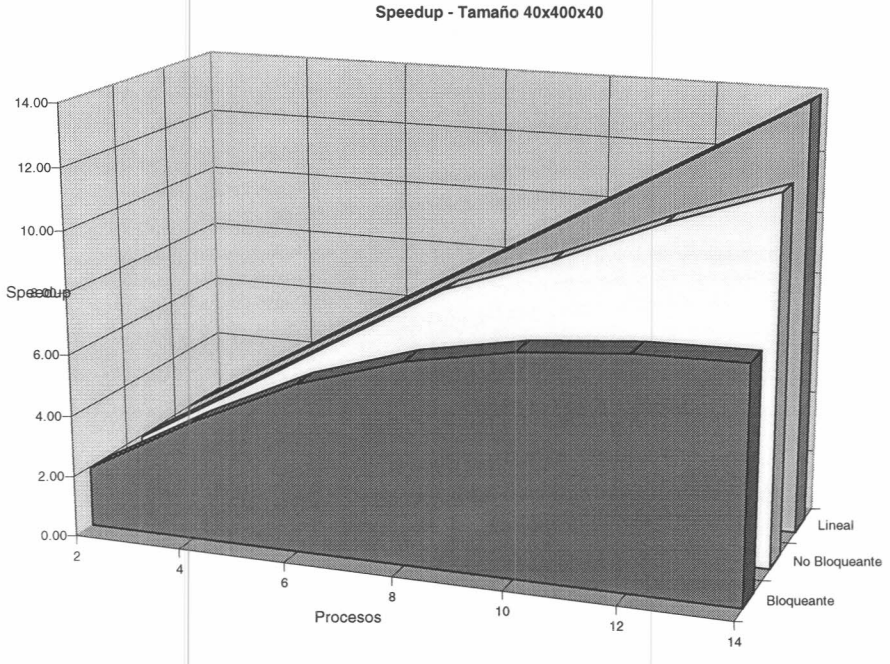


Fig. 5.3.2.1.d – Speedup del algoritmo paralelo para sobre una malla de tamaño 40x400x40 diferentes tamaños de malla y diferentes cantidades de procesadores involucrados

Por último, las corridas con tamaño de malla de 40x400x40 tienen los mejores valores de speedup. El gráfico de la corrida no bloqueante se acerca aún más al speedup lineal.

5.3.2.2 - Eficiencia

La segunda medición de relevancia es la eficiencia del algoritmo paralelo para las mismas ejecuciones del caso anterior. Los resultados obtenidos fueron los siguientes:

Eficiencia									
Tamaño	**Procesos**	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**
40x100x40	**Bloq**	1.00	0.99	0.84	0.65	0.47	0.36	0.28	0.22
	Nobloq	1.00	0.98	0.90	0.78	0.73	0.65	0.61	0.53
40x200x40	**Bloq**	1.00	1.00	0.94	0.82	0.67	0.55	0.44	0.36
	Nobloq	1.00	0.98	0.99	0.99	0.96	0.92	0.85	0.75
40x400x40	**Bloq**	1.00	0.98	0.96	0.91	0.82	0.72	0.62	0.54
	Nobloq	1.00	0.97	0.98	0.99	0.99	0.93	0.89	0.85

Tabla 5.3.2.2-a – Valores de Eficiencia

Se puede observar que la eficiencia es mayor cuando hay menos procesadores y cuando el tamaño de malla es mayor, lo cual es consistente con lo observado anteriormente. El peor de los casos, fue una eficiencia del 22% para el caso de menor malla (40x100x40) con la mayor cantidad de procesos (14) y comunicaciones bloqueantes . En este caso, cada proceso tenía entre 7 y 8 intervalos para calcular, y 4 para intercambiar, por lo que la relación entre superficie de cálculo y superficie de intercambio era bastante desfavorable. En cambio, para el tamaño de malla más grande, en la partición en la misma cantidad de procesadores se obtuvo una eficiencia del 54% para el caso bloqueante. Es importante notar que, las mismas ejecuciones con comunicaciones no bloqueantes, llevaron la eficiencia del 22% al 53% y del 54% al 85% respectivamente. Para poder apreciar mejor los valores de eficiencia, se realizaron gráficos comparativos para cada tamaño de malla, entre las ejecuciones con comunicaciones bloqueantes, no bloqueantes y la máxima eficiencia teórica (eficiencia 1)

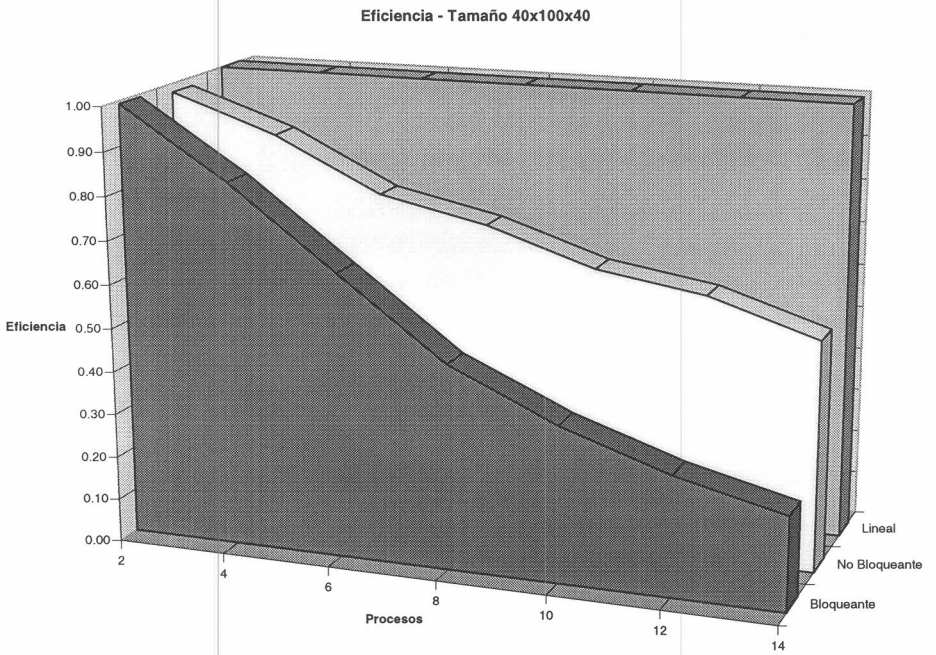


Fig. 5.3.2.2.b – Eficiencia del algoritmo para tamaños de malla 40x100x40

En este primer gráfico correspondiente a la corrida de menor tamaño de malla, se observa la notable disminución de la eficiencia en el caso bloqueante, que es reducida en gran medida en el caso no bloqueante. La diferencia con la eficiencia máxima es aún muy grande.

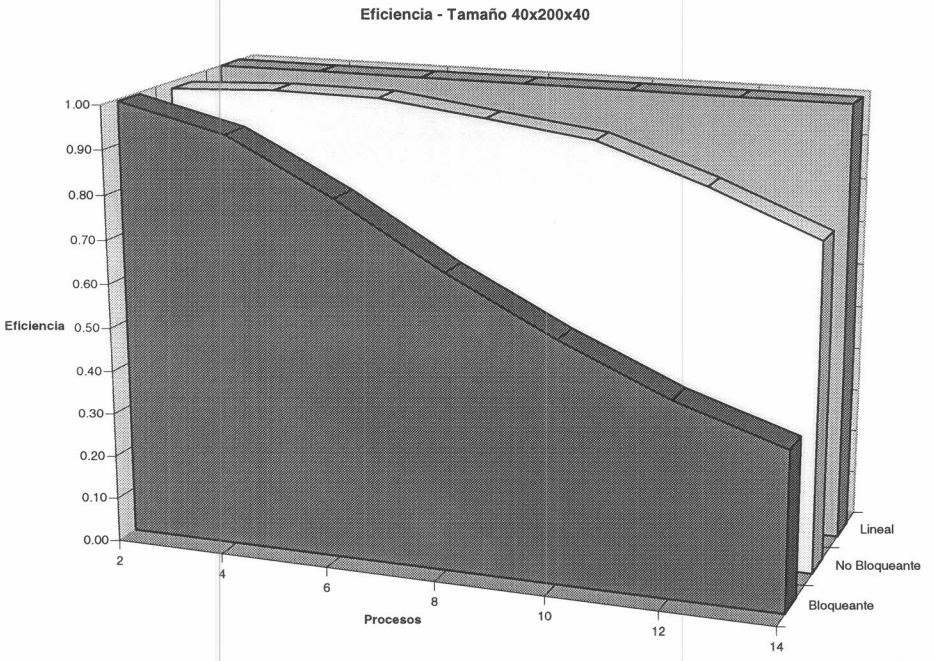


Fig. 5.3.2.2.c – Eficiencia del algoritmo para tamaños de malla 40x200x40

El aumento del tamaño de la malla a 40x200x40 da como resultado una eficiencia mucho mayor. El caso no bloqueante se acerca mucho más a la eficiencia del 100% y es aún mayor la diferencia con el caso bloqueante. Esto puede deberse a que, al ser mayor el tamaño de la malla, es también mayor el tiempo de cálculo y se pueden solapar por completo las operaciones de envío y recepción de mensajes con el cálculo local.

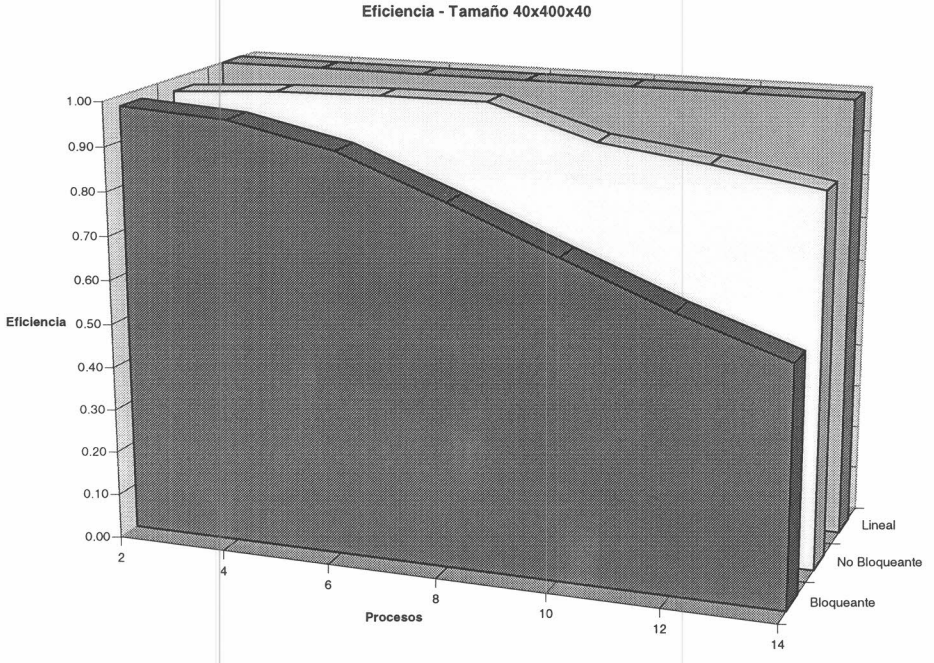


Fig. 5.3.2.2.d – Eficiencia del algoritmo para tamaños de malla 40x400x40

En este último gráfico, correspondiente a la corrida con un tamaño de malla de 40x400x40, se observan, como era de esperar los mejores valore de eficiencia. El caso no

bloqueante una eficiencia casi óptima hasta los 8 procesadores, reduciéndose lentamente a partir de los 10 procesadores.

Es importante notar que, para poder realizar una comparación cierta entre las corridas serial y paralela, es requisito que el tamaño de la malla sea lo suficientemente pequeño como para poder correr en un solo procesador (un único nodo del cluster). Esto limita inherentemente el tamaño de la malla de cálculo a utilizar en estas pruebas y es la razón por la que se hicieron simulaciones de tamaño hasta 40x400x40. No obstante, para brindar una noción cierta acerca de los tiempos de corrida para otros casos que quedan fuera del alcance de un único nodo de los disponibles en el cluster del LSC, se puede calcular, en base a los valores obtenidos anteriormente, una estimación de los mismos.

Supongamos entonces una ejecución sobre una malla de 120x1200x120 elementos. La proporción entre cálculo local e intercambio de mensajes es la misma que para el caso de 40x400x40, con lo que podemos esperar que los valores de eficiencia y speedup sean los mismos. (Speedup de 11.88 y eficiencia de 0.85 para 14 procesos utilizando comunicaciones no bloqueantes). Con estos valores, podemos estimar los tiempos de la siguiente forma:

Ejecución secuencial: 18 días, 6 horas y 35 minutos (26.315 minutos)
Ejecución paralela (14 procesos): 1 día, 12 horas y 55 minutos (2215 minutos)

Así, una simulación pasaría de tardar mas de 2 semanas y media, a tardar 1 día y medio. Esta progresión, muestra claramente los beneficios de la paralelización, en cuanto a los tiempos necesarios para la obtención de los resultados.

5.3.3 Resultados del balance de carga

Como se explicó en la sección 3.1, el balance de carga es una herramienta fundamental para el correcto aprovechamiento del poder de cálculo de un cluster heterogéneo. Los resultados de las corridas con balance de carga no influyen en el speedup del algoritmo propiamente dicho, y es por esto que se estudian en un capítulo separado.

Para medir el comportamiento del balance de carga semi-dinámico explicado en la sección 3.1.2, primero es necesario conocer el cluster sobre el cual se van a hacer las mediciones. En cuanto a procesadores, el cluster Beowulf del Laboratorio de Sistemas Complejos cuenta con la siguiente distribución:

- 4 nodos con procesador Intel Pentium III de 733 MHZ
- 2 nodos con procesador AMD Athlon 900 MHZ
- 6 nodos con procesador AMD Athlon 946 MHZ
- 4 nodos con procesador AMD Athlon 1.2 GHZ

Nota: Uno de los nodos con procesador Intel Pentium III es el master y normalmente no se utiliza en las corridas sino para proveer conexión con el exterior y para correr los programas de administración de colas y procesos (MAUI, PBS, etc.). Es por eso que la parte operativa del cluster consiste en 15 nodos.

Con esta distribución, se realizaron 2 observaciones, tendientes a evaluar el desempeño de la rutina de balance de carga semidinámco al comienzo de la ejecución:

5.3.3.1 - Observación número 1 con balance de carga

Como en el cluster se cuenta con 4 tipos diferentes de procesadores, se tomó un subconjunto del cluster, de 4 procesadores, con 1 procesador de cada clase. Esto es, un Pentium III de 733MHZ, un Athlon de 900 MHZ, un Athlon de 956 MHZ y un Athlon de 1.2GHZ. Sobre este subconjunto, se realizaron 2 ejecuciones de la misma simulación, sobre una malla de 30x400x30, una de ellas sin balance de carga, y la otra con balance de carga. Los tiempos empleados por cada simulación fueron los siguientes:

Sin Balance	Con Balance
313.96	265.21

Tabla 5.3.3.1..a- Comparación de corridas con y sin balance de carga

Se puede apreciar que la corrida con balance de carga fue un 18,38% más rápida que la misma corrida sin balance, lo que demuestra un mejor aprovechamiento del poder de cálculo dentro de cada nodo.

En cuanto al porcentaje de tiempo dedicado al cálculo neto, en cada uno de los nodos, los resultados fueron los siguientes:

	Sin Balance	Con Balance
Pentium II 733	97.27%	90.18%
Athlon 900	81.18%	91.57%
Athlon 950	74.36%	94.63%
Athlon 1200	67.69%	93.06%

Tabla 5.3.3.1.b- Porcentaje de tiempo dedicado al cálculo neto, en los diferentes procesadores, con y sin balance de carga.

Los valores de porcentaje de uso neto de procesador notan que los procesadores más lentos, tienen un mayor porcentaje de tiempo neto de cálculo, mientras que los procesadores más rápidos utilizan un porcentaje de tiempo de la corrida mucho menor para el cálculo neto. Esto demuestra claramente cómo, en la corrida sin balance de carga, los nodos más rápidos tienen que esperar a los nodos más lentos, desaprovechando así parte de su poder de cálculo.

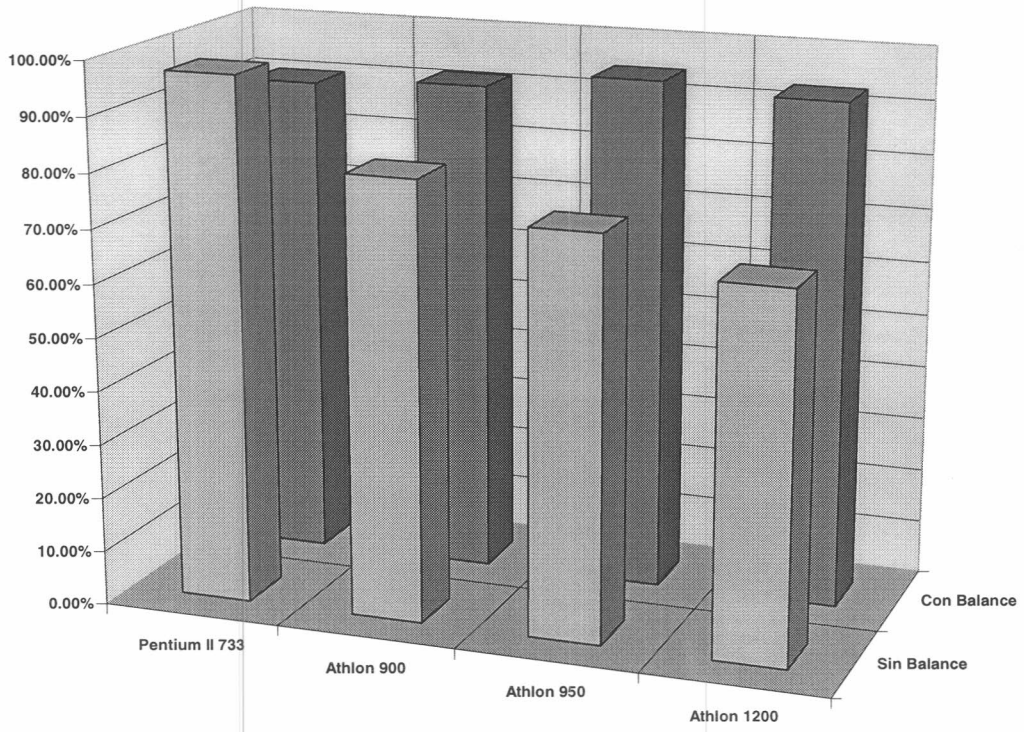
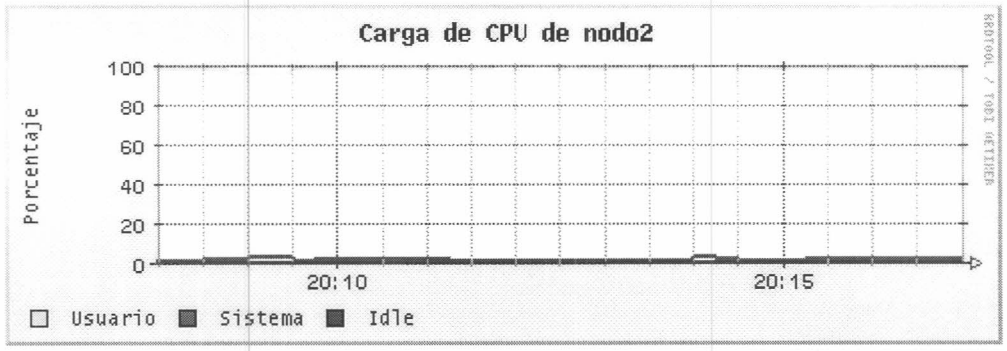


Fig. 5.3.3.1.c - Porcentaje de tiempo dedicado al cálculo neto, en los diferentes procesadores, con y sin balance de carga.

El gráfico muestra que, para el caso de las corridas sin balance de carga, la utilización del poder de cálculo disminuye a medida que se aumenta la performance del procesador, mientras que en las corridas con balance de carga, esta utilización se mantiene relativamente constante.

Mediante la utilización de las utilidades de monitoreo "RRDtool y Ganglia" se realizaron mediciones del uso de procesador, por parte del usuario, y del sistema durante un tramo de la corrida. Se obtuvieron así, los gráficos de utilización de procesador. Los siguientes son los gráficos correspondientes a la corrida sin balance de carga.



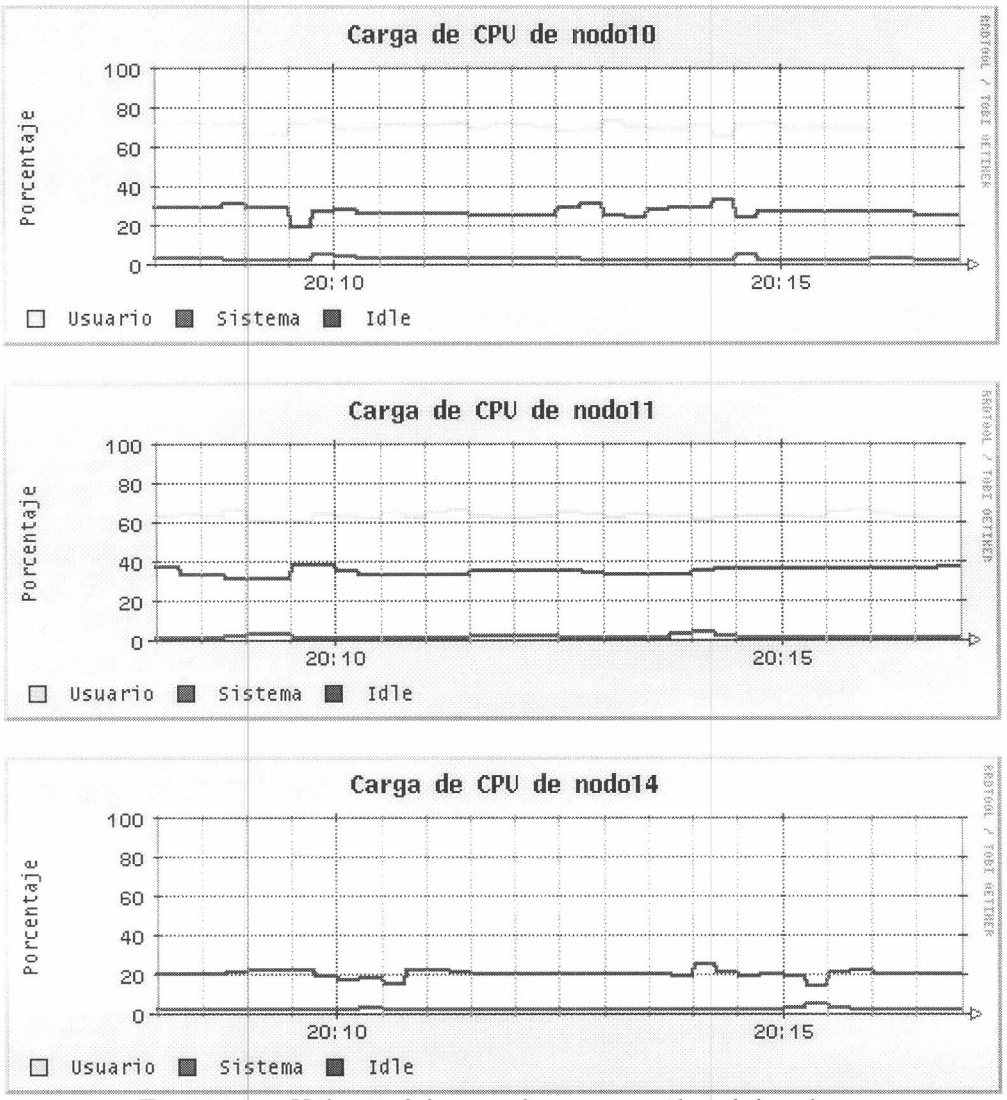
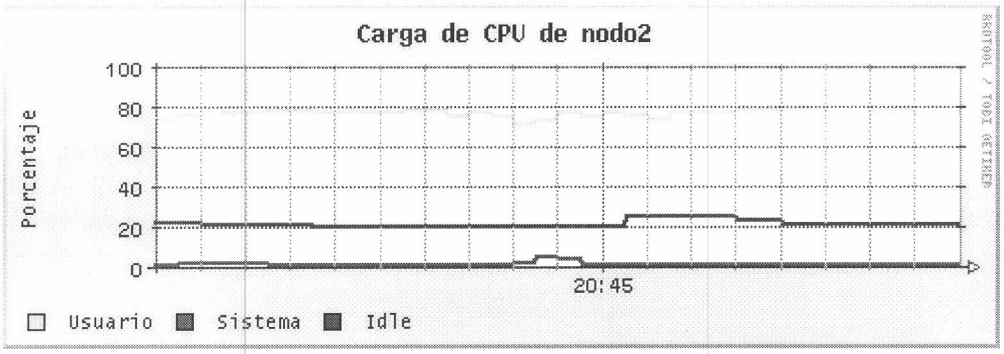


Fig. 5.3.3.1.c – Utilización de los procesadores en una corrida sin balance de carga

Estos gráficos, muestran cómo se está desperdiciando capacidad de cálculo en los nodos más rápidos, debido a que estos tienen que esperar a que los más lentos terminen su rutina de cálculo local para poder sincronizarse, enviar y recibir los planos de intercambio y pasar a la siguiente iteración.

Al ejecutar la misma corrida, pero utilizando las rutinas de balance de carga semidinámico, los gráficos de utilización de procesador obtenidos fueron los siguientes:



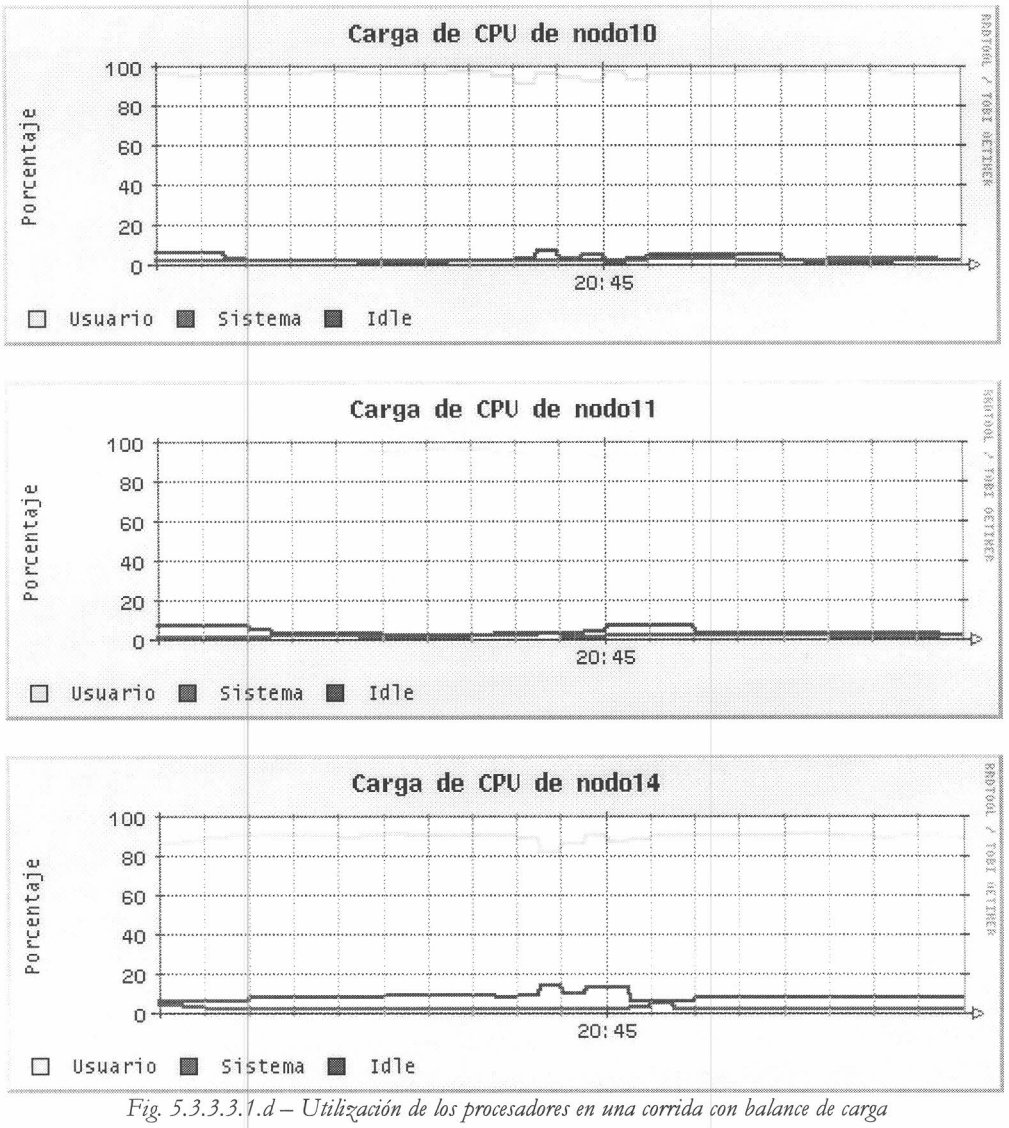


Fig. 5.3.3.3.1.d – Utilización de los procesadores en una corrida con balance de carga

Estos últimos gráficos, demuestran que el balance de carga produjo un mejor aprovechamiento del recurso procesador en los diferentes nodos, manteniendo en casi todos ellos una tasa de uso de más del 90%..

5.3.3.2 - Observación número 2 con balance de carga

Se realizaron 2 ejecuciones de la misma simulación anterior, pero utilizando los 15 nodos del cluster. La primera de las simulaciones se realizó sin balance de carga y la segunda con balance de carga. Los tiempos de ejecución de las corridas fueron los siguientes:

Sin Balance	Con Balance
78.22	66.75

Tabla 5.3.3.2.a- Comparación de corridas con y sin balance de carga

Se observa aquí, que la performance total del cluster, utilizando la técnica de balance de carga semidinámico al inicio de la corrida, se incrementó en un 17,18% con respecto a la utilización de asignación de intervalos por partes iguales (sin balance de carga).

Estos valores se obtienen en un cluster que, pese a ser heterogéneo, no lo es en una medida demasiado extrema. En casos de clusters heterogéneos en donde la diferencia entre el poder de cálculo de los procesadores sea mayor, es de esperar que esta diferencia de performance sea también mayor, puesto que de no aplicar balance de carga, será mayor el desperdicio de poder de cálculo por parte de los procesadores más rápidos.

Capítulo 6 – Conclusiones

Este trabajo de paralelización, permitió observar la medida en la que un algoritmo paralelo corriendo en un cluster Beowulf, proporciona niveles de performance muy superiores a los obtenidos en una computadora serial Standard.

Se vio que la eficiencia del algoritmo aumenta en los casos en que es mayor la proporción entre tamaño de dominio de cálculo local de cada proceso y tamaño de buffers de intercambio. Esta eficiencia es mejor en problemas de mayor tamaño y va disminuyendo a medida que se aumenta la cantidad de procesadores (manteniendo el tamaño de problema. Esto muestra que, por un lado, hay overhead correspondiente a la paralelización, y por otro lado, el costo de la comunicación es apreciable. Esto último se pone de manifiesto en el análisis de resultado, en donde se ve que, para un mismo dominio, cuanto mayor es la división en subdominios, mayor es el porcentaje de tiempo dedicado a la comunicación, y por ende, es menor la proporción de tiempo neto de cálculo.

Las comunicaciones no bloqueantes, permiten realizar solapamiento de los tiempos de comunicación (o parte de ellos) con el tiempo de cálculo. Esto mejora notablemente la performance del algoritmo paralelo, llegando a obtener mejoras de performance superiores al 40%. Esta mejora es más notoria a medida que se aumenta la cantidad de nodos de la ejecución paralela, puesto que en estos casos es mayor la proporción de tiempo de ejecución con respecto a la proporción de cálculo. Se vió que para poder aprovechar los beneficios de las comunicaciones no bloqueantes, es necesario contar con buffers de intercambio lo suficientemente "grandes" y la experiencia mostró que es mucho más conveniente realizar menos comunicaciones con buffers grandes (agrupados) que realizar más comunicaciones con buffers pequeños

Diferentes implementaciones del Standard de MPI pueden ocasionar tiempos de ejecución diferentes, sobre todo, en lo referente a la implementación de las comunicaciones no bloqueantes.

El balance de carga semidinámico mostró un aprovechamiento mucho mejor del tiempo de procesador, llegando en todos los procesadores a niveles superiores al 90%. Se lograron mejoras en el tiempo de proceso superiores al 18% en el cluster "Speedy Gonzalez" del LSC

Las diferencias entre los resultados del algoritmo serial y el algoritmo paralelo no son apreciables, y en la mayoría de los casos se reducen a medida que se transcurre el tiempo de simulación.

Capítulo 7 - Trabajo futuro

Un trabajo interesante a realizar en el futuro, es lograr un balance totalmente dinámico durante la ejecución de la simulación. Este balance, debe contemplar los sectores de "convergencia dura". Esto es, sectores del dominio en donde la convergencia tarda más que en otros sectores. Un método de balanceo totalmente dinámico debería ser capaz de identificar estos sectores de "convergencia dura" y reparticionar "en caliente" los subdominios para tender a homogeneizar los tiempos de convergencia y, por ende, mejorar la performance.

Es de utilidad también, implementar "checkpoints" para poder interrumpir y reanudar una ejecución del algoritmo paralelo. Esto es de utilidad ante fallas en el sistema o necesidad de compartir el uso del cluster con otros usuarios, y permitiría un mejor aprovechamiento de los tiempos de ejecución.

Por último, a quienes les interese trabajar con problemas en los que la precisión extrema sea un requerimiento, les puede resultar útil diseñar un algoritmo paralelo que opere en "modo pipeline" y por ende, arroje resultados exactamente iguales a la versión serial.

Apéndice A: Especificaciones técnicas

De la implementación del algoritmo paralelo

Lenguaje de programación: C++
Compilador: gcc 2.96
Sistema Operativo: Linux (kernel 2.4.18)
Implementaciones utilizadas de MPI: LAM-MPI 6.5.9 y MPICH 1.2.4

Del cluster Beowulf "Speedy Gonzalez"

El cluster Beowulf del Laboratorio de Sistemas Complejos del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales, cuenta actualmente con 16 nodos con las siguientes características:

Nodo	CPU	MHz	Disco	RAM (Mb)
Nodo1	Pentium III	733	ST3120022A	191
Nodo2	Pentium III	733	ST320423A	128
Nodo3	Pentium III	733	ST320423A	128
Nodo4	Pentium III	733	ST320423A	128
Nodo5	AMD Athlon	946	SV2042H	183
Nodo6	AMD Athlon	946	ST320410A	120
Nodo7	AMD Athlon	946	SV2042H	183
Nodo8	AMD Athlon	946	SV2042H	183
Nodo9	AMD Athlon	946	FIREBALLlct20	183
nodo10	AMD Athlon	946	FIREBALLlct20	183
nodo11	AMD Athlon	1200	ST320423A	505
nodo12	AMD Athlon	1200	ST320423A	505
nodo13	AMD Athlon	1200	ST320410A	505
nodo14	AMD Athlon	900	ST320410A	183
nodo15	AMD Athlon	1200	ST320410A	505
nodo16	AMD Athlon	900	ST320410A	183

El cluster cuenta también con los siguientes paquetes de software relativos a la administración de los procesos y recursos:

- MAUI 3.0.7
- Open PBS 2.3.16

Apéndice B: Bibliografía

[1] Peter S. Pacheco *"Parallel Programming with MPI"*, Morgan Kaufmann Publishers, inc., San Francisco, 1997.

[2] William Gropp, Wwing Lusk, Anthony Skjellum *"Using MPI: Portable Parallel Programmin with the Messae Passing Interface, second Edition"*, MIT Press, London, 1999

[3] G. Amdahl, *"The validity of the single processor approach to achieving large scale computing capabilities"* AFIPS cnference proceedings, Spring Joint Computing Conference, 1967

[4] J.L Gustafson, *"Amdhal's law re-evaluated"*, Communications of the ACM, 1988

[5] ChaMPIon PRO, http://www.mpi-softtech.com/products/cluster/champion_pro)

[6] J.B. White III and S.W. Bova, *"Where is the Overlap?, An analysys of Popular MPI Implementations"*, CEWES MSRC/PET TR/99-09

[7] SCALI MPI, http://www.scali.com.

[8] LAM-MPI, http://www.lam-mpi.org/.

[9] MPICH, http://www-unix.mcs.anl.gov/mpi/mpich/.

[10] T. Vicsek, Fractal Growth Phenomena, World Scientific, Singapore (1992).

[11] R. M. Brady and Ball, Nature 309, 225 (1984).

[12] V. Fleury, J_N. Chazalviel and M. Rosso, Phys. Rev. Lett. 68, 2492 (1992).

[13] J. Huth, H. Swinney, W. McCormick, A. Kuhn and F. Argoul, Phys. Rev. E 51, 3444 (1995).

[14] G. Marshall, E. Perone, P. Tarela and P. Mocskos, Chaos, Solitons and Fractals 6, 315 (1995).

[15] S. Dengra, G. Marshall and F. Molina, J. Phys. Soc. Japan 69, 963 (2000).

[16] G. Marshall and P. Mocskos, Phys. Rev E 55, 549 (1997).

[17] G. Marshall, P. Mocskos, H. L. Swinney and J. M. Huth, Phys. Rev E 59, 2157 (1999).

[18] G. González, G. Marshall, F.V. Molina, S. Dengra, Physical Review E, volume 65 051607 (2002)

[19] G. Marshall, F. V. Molina, S. Dengra, E. Mocskos, and E. Arias, Electrodeposition physicochemical hydrodynamics: experiments, theory, and numerical simulation studies, in Ed. S. Idelsohn and V. Sonzogni Guest Editors, Journal of Computational Methods in Sciences and Engineering, (2003, in press).

[20] G. Marshall, E. Mocskos, F. V. Molina and S. Dengra, The Three-Dimensional Nature of Ion Transport in Electrochemical Deposition, (Physical Review E 68, 2003, in press,).

[21] G. Marshall, Solución Numérica de Ecuaciones Diferenciales, Tomo II, Ecuaciones en Derivadas Parciales, Editorial Reverté, Buenos Aires, 1986, 300 pp.

[22] La ultima version de OPENDX para Linux se encuentra en la World Wide Web en http//www.opendx.org

[23] El programa de graficación Vis5D, así como la información del producto, se encuentran en http://www.ssec.wisc.edu/~billh/vis5d.html