Resumen

La simulación numérica de sistemas complejos, donde coexisten una amplia variedad de escalas, requiere gran capacidad de cálculo para el tratamiento de grandes volúmenes de información, sólo disponible hasta hace poco tiempo en supercomputadoras. Una alternativa reciente, más económica y de rendimiento razonable, es utilizar una computadora paralela tipo Beowulf, consistente en la interconexión de un grupo de computadoras personales en una red de alta velocidad.

Entre los sistemas complejos, la formación de patrones de crecimiento en electrodeposición (ECD), es un fenómeno común a una amplia gama de problemas de la física a la biología. De allí la importancia de su estudio experimental y numérico. La posibilidad de extender los conocimientos actuales sobre ECD es el objetivo último de nuestro trabajo. Esto se puede lograr mediante la utilización de mallas espacio-temporales más densas para resolver la diversidad de escalas de la parametrizacin física, tarea que escapa a las máquinas secuenciales.

En este trabajo describimos la construcción de un cluster Beowulf consistente en 16 nodos (PC's) conectados con un switch Fast Ethernet, y la simulación numérica de problemas de ECD para diferentes configuraciones geométricas y físicas.

En nuestra implementación paralela del algoritmo numérico, incorporamos distintas estrategias de descomposición y asignación de dominios, y de intercambio de datos entre cada uno de los procesos ejecutados en nuestro Beowulf. Estudiamos la performance del algoritmo desarrollado y aplicamos a nuestro problema de ECD la configuración de mayor rendimiento.

El desarrollo de una solución paralela ha permitido ejecutar simulaciones de ECD que requieren tres días en la más veloz de las computadoras disponibles, en menos de seis horas; alcanzando un speedup superior a 12 en un cluster de 16 procesadores. Asimismo, se ha logrado extender la estabilidad numérica a una parametrización física más amplia.

Los resultados del trabajo permiten además estimar tiempos de simulación bajo diferentes configuraciones del problema o el Beowulf (para evaluar, por ejemplo, la utilidad de incorporar nuevas computadoras al cluster), determinando el speedup y la eficiencia a esperar.

Abstract

Numerical simulation of complex systems, where different scales coexist, requires large computing resources to process large volumes of information that, until recent times, were only available in supercomputers. A recent alternative, less expensive and with reasonable performance, is the use of a Beowulf parallel computer, where a group of personal computers are interconnected by a high-speed network.

Among complex systems, growth pattern formation in electrodeposition (ECD), is a common phenomenon in a wide range of problems from physics to biology. Thus the interest in studying it through experiments and numerical simulation. The ultimate goal of our work is the possibility of extending current knowledge in electrodeposition. This can be achieved by using denser spatio-temporal grids to resolve the scale diversity of the physical parameterization, a task outside the reach of sequential computers.

In this work we describe the construction of a Beowulf cluster, with 16 nodes (PCs) connected through a Fast Ethernet switch, and the numerical simulation of ECD problems under different geometric and physical configurations.

In our parallel implementation of the numerical algorithm, different strategies for domain decomposition and assignment were introduced. Different strategies were also developed to perform data interchange between each of the processes executed by our Beowulf. We studied the performance of our algorithm and applied to the ECD problem the best performing configuration.

The parallel solution we describe allowed a reduction in the execution time of a typical ECD simulation from three and a half days (in the fastest computer available) to less than six hours, thus reaching a speedup well above 12 in a cluster with 16 processors. Moreover, numerical stability was extended to a broader physical parameterization range.

Finally, the results of this work allow the estimation of the simulation time for different problem and cluster configurations (e.g., adding new computers to the Beowulf), and in this way determining the speedup and efficiency to be expected.

Índice general

Agradecimientos

Quisiera agradecer a mi director por su supervisión y consejo; a todos los integrantes del Laboratorio de Física Computacional del Departamento de Computación, en especial a Silvina Dengra por estar siempre disponible para completar las simulaciones en el Beowulf y obtener sus resultados; a los administradores de la red del Departamento (Cristian, Maxi y Juan), por su asistencia ante mis consultas en la configuración del Beowulf; y a Fernando Molina por los comentarios sobre este trabajo.

Capítulo 1

Introducción

En este capítulo comenzaremos describiendo el proceso de electrodeposición electroquímica (ECD) en celdas delgadas que motiva el desarrollo de este trabajo, y los experimentos que se realizan para comparar los resultados de las simulaciones derivadas del modelo matemático con la realidad.

Debido a que la simulación numérica de este proceso requiere una gran capacidad de cálculo, fuera del alcance de una única computadora secuencial, es necesario desarrollar una implementación paralela para aplicarla al problema. Tras la presentación del proceso físico, entonces, describiremos a las computadoras paralelas, que clasificaremos de acuerdo a la taxonomía de Flynn. Luego se discutirá la evolución reciente de los clusters de computadoras personales en la resolución de problemas de alta performance. En particular, definiremos a las computadoras paralelas Beowulf, clasificación a la que corresponde el cluster de PCs configurado para utilizar en este trabajo.

Veremos luego herramientas que nos permitirán analizar la performance de la solución paralela a desarrollar, por medio del análisis de complejidad de algoritmos paralelos; y definiremos conceptos útiles para evaluar el rendimiento obtenido como: speedup, eficiencia y escalabilidad.

Haremos un breve repaso del paradigma de *message-passing*, que consiste en que un grupo de procesos se comunican entre si por medio del intercambio de mensajes para realizar una tarea determinada. En nuestro trabajo se utilizará la biblioteca standard MPI (Message Passing Interface) para la implementación de la solución paralela.

Antes de pasar al desarrollo del algoritmo paralelo, presentaremos una metodología de diseño de algoritmos paralelos, que aplicaremos a la simulación de nuestro problema.

1.1 Transporte iónico en deposición electroquímica en celdas delgadas

La deposición electroquímica (ECD) de depósitos ramificados en celdas delgadas, que producen complejas geometrías de carácter fractal o dendrítica, es un modelo paradigmático para el estudio de la formación de patrones de crecimiento. La celda electrolítica consiste en dos placas de vidrio que intercalan dos electrodos en disposición paralela y un electrolito de una sal metálica. Se aplica entre los electrodos una diferencia de voltaje que produce un depósito en forma de árbol por la reducción de los iones metálicos. La estructura del electrodepósito varía desde una morfología fractal hasta una ramificación densa, dependiendo de los parámetros como la geometría de la celda, la concentración de la solución, o la diferencia de voltaje; las variaciones de la morfología no son aún completamente comprendidas.

En la fig. 1.1 se muestra una típica celda utilizada en los experimentos de electroconvección. Una delgada capa de una solución de una sal metálica es confinada entre dos placas paralelas de vidrio separadas por una pequeña distancia d, limitada en ambos extremos por electrodos (usualmente de zinc o cobre) de longitud w, separados por una distancia l. El flujo del fluido es visualizado con partículas trazadoras -*tracer particles*- (1-3 μm de diámetro) de la misma densidad que la solución electrolítica.

Para explicar ECD necesitamos desarrollar un modelo fenomenológico. En un experimento típico, cuando el circuito es cerrado la corriente comienza a fluir a través del electrolito: cationes y aniones se mueven hacia el cátodo y ánodo, respectivamente. Debido a la deficiencia de iones cerca del cátodo (agregación de cationes y migración de aniones) y al aumento de iones cerca del ánodo (los aniones se apilan y los cationes entran por disolución anódica), en un período inicial -es decir, antes que algún crecimiento sea visible en el cátodo- se desarrollan rápidamente zonas de baja concentración cerca del cátodo y alta concentración cerca del ánodo. Estas son separadas por una zona de concentración inicial. Cerca del cátodo la densidad es más baja que en el seno del electrolito, mientras que en el ánodo es más alta. Esta configuración inestable genera un flujo de densidad de corriente en ambos electrodos: en el cátodo el fluido asciende hacia la placa superior y en el ánodo desciende hacia el fondo. Estas corrientes generan dos vórtices o rollos en cada electrodo, rotando en el mismo sentido, que se mueven y expanden en dirección del otro, invadiendo el centro de la celda. Durante este período inicial, la deficiencia de cationes en el cátodo se supone uniforme. Simultáneamente, en una estrecha región cerca del cátodo una carga local se desarrolla, dando origen a fuerzas de Coulomb que inicialmente apuntan hacia el cátodo.

Tras el período inicial, que típicamente dura unos pocos segundos, se desarrolla una inestabilidad por la cual se inicia el crecimiento ramificado del depósito. El depósito se presenta como un arreglo tridimensional de filamentos metálicos finos y porosos, las fuerzas de Coulomb se concentran en las puntas; cada filamento permite al fluido penetrar su punta y ser eyectado por los lados, configurando un vórtice toroidal gobernado por la fuerza eléctrica. En el casi-plano del crecimiento, la imagen tridimensional previa se reduce al par de vórtices rotando en dirección contraria y a los arcos en las puntas de cada filamento; en el plano

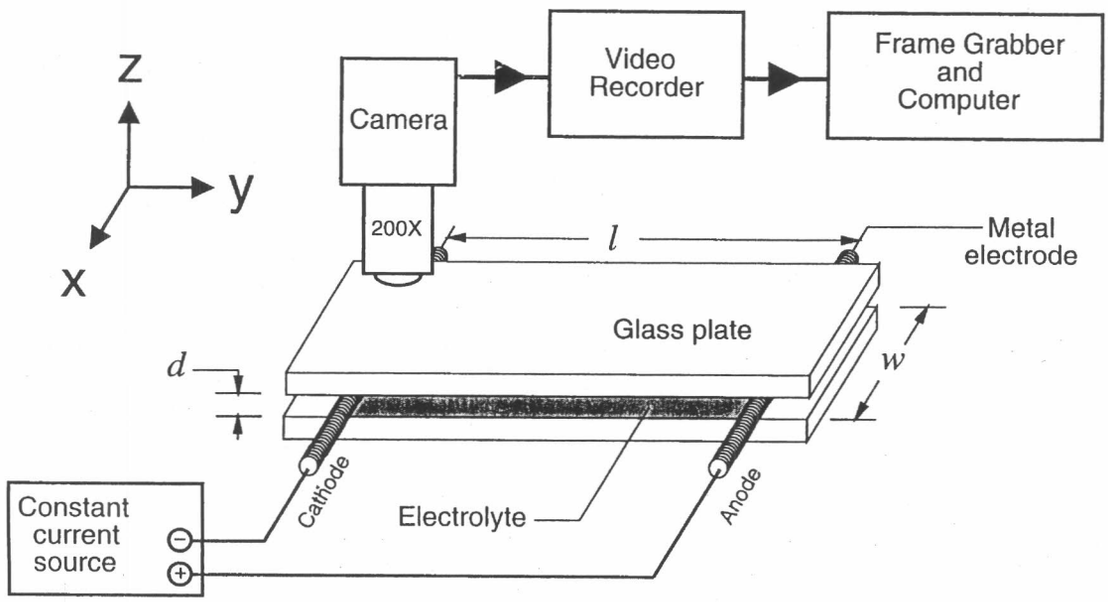


Figura 1.1: Diagrama del sistema utilizado en electrodeposición. [14]

normal a ambos, el casi-plano que contiene los electrodos, se reduce a un par de tubos vorticosos.

En [19] se desarrolla un modelo macroscópico tridimensional partiendo de primeros principios, que incluye todas las variables físicas relevantes para el transporte iónico: difusión, migración, convección regida por fuerzas de Coulomb y de empuje, y un mecanismo que imita el crecimiento del frente.

El modelo tridimensional se aproxima numéricamente en una malla, como dos modelos bidimensionales, una en un plano horizontal y la otra en plano vertical. El modelo del plano horizontal simula ECD en el casi-plano del crecimiento (el plano que contiene los electrodos); este modelo describe el caso límite dominado por la electroconvección. El modelo del plano vertical simula ECD en el plano perpendicular al casi-plano del crecimiento y a los electrodos; este modelo describe el caso límite dominado por la gravitoconvección.

Estas simulaciones están en directa analogía con los experimentos físicos: el modelo del plano horizontal corresponde a la vista superior de la celda; mientras que el modelo del plano vertical corresponde a la vista lateral de la celda. En este trabajo se presentarán resultados obtenidos con el modelo de vista lateral en la simulación del experimento de ECD.

1.2 Computadoras paralelas

"Una computadora paralela es un conjunto de procesadores capaces de trabajar cooperativamente para resolver un problema computacional" [8] La definición es lo suficientemente amplia para incluir supercomputadoras paralelas con cientos o miles de procesadores, **NoWs** (*Networks of Workstations* -Redes de estaciones de trabajo-), estaciones de trabajo con multiples-procesadores, entre otros.

En esta sección, comenzamos describiendo la arquitectura de la máquina clásica de von Neumann, para luego hacer un repaso de distintas arquitecturas paralelas a partir de la clasificación de Flynn. Finalmente se presenta el origen y definición de las máquinas paralelas Beowulf.

1.2.1 Máquina de von Neumann

La máquina clásica de von Neumann está compuesta por un *procesador central* (CPU) y la memoria principal (ver fig. 1.2). A su vez, el CPU está dividido en una *unidad de control* y una *unidad aritmético-lógica* (ALU). En la memoria principal se encuentran tanto los programas (compuestos por instrucciones) como los datos que estos utilizan. La unidad de control dirige la ejecución de los programas y la ALU realiza los cálculos requeridos por cada programa.

Cuando los datos o instrucciones están siendo utilizados por un programa, ellos son almacenados en posiciones de memoria de alta velocidad, denominados *registros*. Para comunicar los registros con la memoria principal, se los conecta por medio del *bus*, que permite el traslado de datos e instrucciones entre ellos.

El cuello de botella de una máquina de von Neumann se encuentra en la transferencia de datos e instrucciones entre la memoria y el CPU. Es decir, a pesar de los aumentos en performance de la CPU, la velocidad de ejecución estará limitada por la tasa de transferencia de la secuencia de instrucciones y datos entre la memoria y el CPU.[1]

1.2.2 Taxonomía de Flynn

La clasificación más popular de máquinas paralelas es la denominada *Taxonomía de Flynn* [7], que se organiza por la forma en que las instrucciones y los datos son utilizados, dando lugar a cuatro arquitecturas principales.

La máquina clásica de Von Neumann tiene sólo un flujo de instrucciones y otro de datos, por lo que es identificada como una máquina *single-instruction single-data* (**SISD**). El extremo opuesto consiste en un sistema *multiple-instruction multiple-data* (**MIMD**), donde un grupo de procesadores autónomos operan sobre su propio flujo de datos. Entre los sis-

[1] Por esta razón, pocas computadoras actualmente responden en forma estricta a la definición de la máquina de von Neumann. La mayor parte de las computadoras actuales utilizan un esquema de memoria jerarquizada, introduciendo un nuevo nivel de memoria entre los registros y la principal, denominada *cache*.

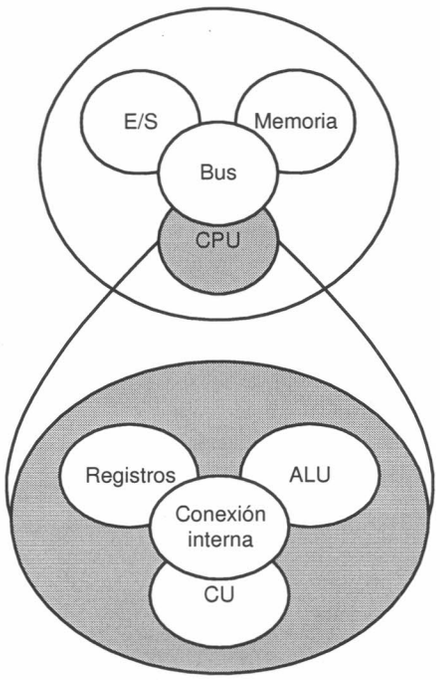


Figura 1.2: Máquina de von Neumann

temas **SISD** y **MIMD** se encuentran los **SIMD** (*single-instruction multiple-data*) y **MISD** (*multiple-instruction single-data*).

1.2.3 Sistemas SIMD

Un sistema **SIMD** (*single-instruction multiple-data*) ejecuta la misma instrucción en un conjunto de datos. Existen dos clases preponderantes de diseños basados en este modelo: los *procesadores matriciales (array processors)* y los *procesadores vectoriales (vector processors)*.

El primero de esos diseños se caracteriza por poseer una unidad de control dedicada a manejar un grupo de unidades aritmético-lógicas subordinadas (generalmente entre 1024 y 16.384 ALUs), cada una de ellas con una pequeña cantidad de memoria. En cada uno de los ciclos de procesamiento, la unidad de control se encarga de enviar la instrucción a realizar a cada uno de los procesadores subordinados que, según el caso, pueden ejecutar la operación o permanecer inactivos (lo que deriva en la degradación de la performance de la máquina). La velocidad de procesamiento de un procesador matricial también se ve afectada en el caso en que un procesador requiera datos almacenados en la memoria asociada a otro, debido a que la información debe ser trasladada por la red de interconexión hasta alcanzar su destino.

Esta clase de procesadores suele sobresalir en el procesamiento de imágenes y señales digitales, así como también en cierta clase de simulaciones de tipo Monte Carlo en donde casi no exista intercambio de datos entre procesadores y el mismo tipo de operaciones deba

ser realizado sobre un conjunto de masivo de datos. Entre los sistemas que responden a la clasificación se encuentran, entre otros: los CM-1 y CM-2 de Thinking Machines, el CPP DAP Gamma II y el Alenia Quadrics.

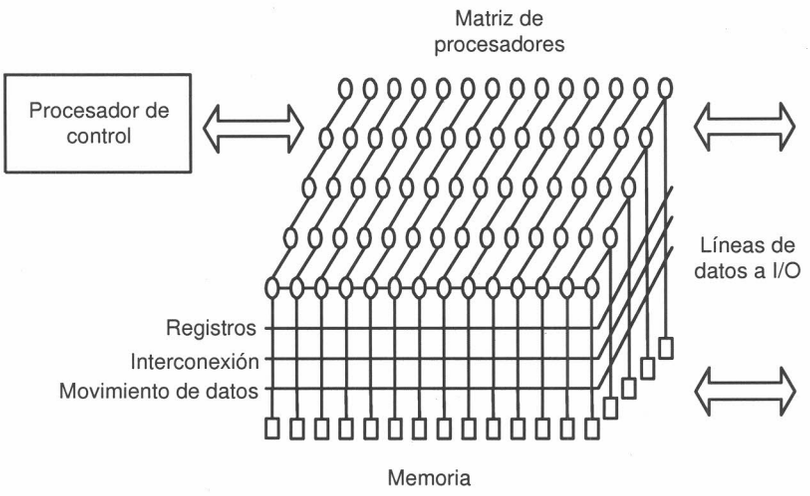


Figura 1.3: Diagrama de un procesador matricial.

En la categoría restante, los *procesadores vectoriales* actúan en conjuntos de datos similares en lugar de datos individuales, por medio de CPUs especialmente diseñadas. En general, cuando los datos pueden ser procesados por las unidades vectoriales, los resultados pueden ser obtenidos en uno, dos o (en casos especiales) tres ciclos de reloj. Los procesadores vectoriales, entonces, trabajan sobre los datos casi en forma paralela pero sólo en operaciones bajo el modo vectorial. Un sistema que responde a esta arquitectura es el Hitachi S3600.

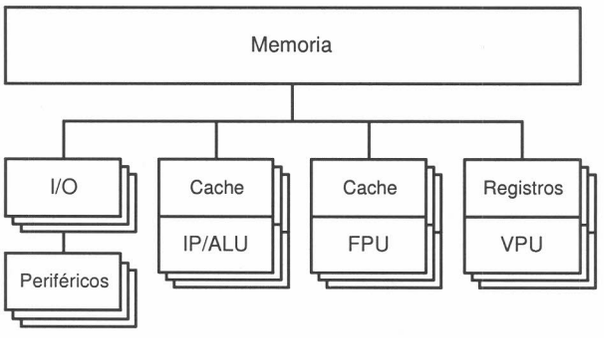


Figura 1.4: Diagrama genérico de un procesador vectorial.

Una consecuencia de la utilización de una unidad dedicada al procesamiento vectorial es la necesidad de implementar un canal de comunicación con la memoria principal que

permita guardar la salida de los resultados de la operación de cálculo anterior mientras que simultáneamente se cargan los que corresponden a la siguiente. En los casos en que el ancho de banda es bajo, la baja preponderancia de las operaciones aritméticas sobre las de lectura/escritura puede producir pérdidas de performance severas.

1.2.4 Sistemas MISD

Teóricamente, en las máquinas **MISD** (*multiple-instruction single-data*), diversas instrucciones deberían operar sobre un único flujo de datos. No existen máquinas de uso práctico pertenecientes a esta clase ni son fáciles de imaginar.

1.2.5 Sistemas MIMD

Las máquinas de la categoría **MIMD** (*multiple-instruction multiple-data*) ejecutan varios flujos de instrucciones sobre diferentes datos en forma paralela. A diferencia de una configuración de múltiples procesadores SISD, las instrucciones y datos que son utilizados en cada procesador están relacionados porque representan distintas partes de la tarea a realizar.[2]

La existencia de una gran variedad de sistemas MIMD hace que la taxonomía de Flynn no sea completamente adecuada para clasificarlos a todos. Por ejemplo, sistemas que se comportan de modo muy diferente como NEC SX-5 de cuatro procesadores y un SGI/Cray T3E de mil procesadores son representantes de la misma categoría. Una distinción usual consiste en agruparlos según utilicen un esquema de memoria compartida o uno de memoria distribuida.

Sistemas MIMD de memoria compartida

Los sistemas MIMD de memoria compartida (**SM-MIMD**) poseen múltiples CPUs que utilizan el mismo espacio de direcciones de memoria. Esto significa que el conocimiento sobre dónde está almacenada la información no es importante debido a que el costo de su acceso es igual para cada procesador.

Existen distintos tipos de interconexión de los CPUs con la memoria. El modelo más simple utiliza un bus (ver fig. 1.5.a) y, por lo tanto, si varios procesadores quieren acceder simultaneamente a la memoria, el bus se saturará, resultando en demoras entre la operación de acceso por lectura/escritura. Para mitigar este problema, cada procesador suele estar acompañado de una unidad de cache. Sin embargo, el limitado ancho de banda del bus no permite que el modelo tenga un buen rendimiento al utilizar una gran cantidad de procesadores.

Otro tipo de interconexión, utilizado por la mayor parte de las restantes computadoras basadas en SM-MIMD, une a los procesadores con la memoria por medio de una red basada en el uso de switches (ver fig. 1.5.b). Tal es el caso del HP Convex SPP1200, que utiliza un

[2]Sin embargo, esto no quiere decir que deba existir un reloj global que sincronice los procesadores (aunque puedan ser programados para realizarla). Es decir, los sistemas MIMD son *asincrónicos*.

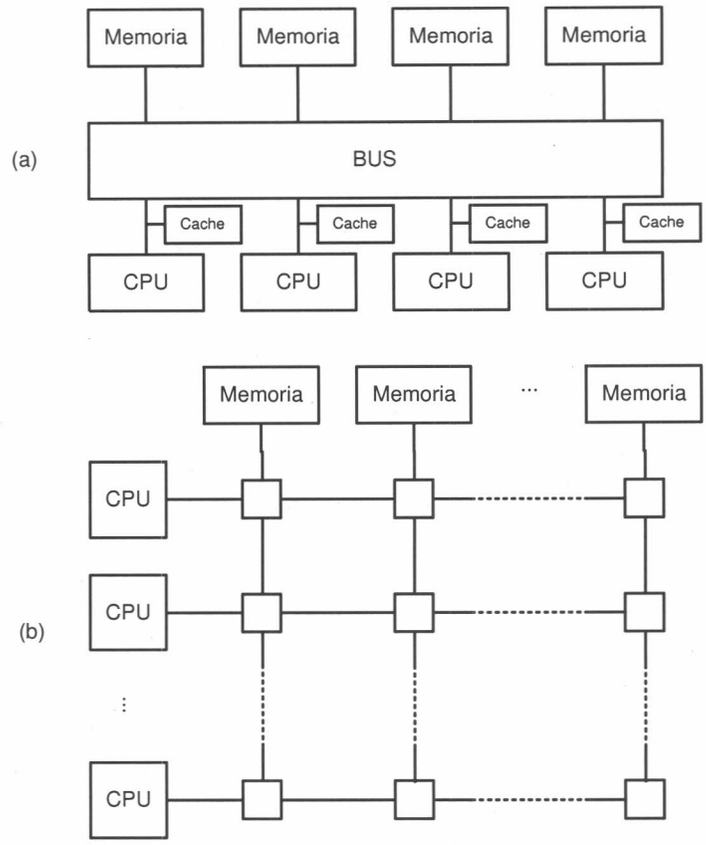


Figura 1.5: Arquitectura de memoria compartida, según interconexión: (a) bus y (b) switch.

crossbar switch de 5x5. El *crossbar switch* puede ser visto como una malla rectangular de cables con switches en los puntos de interconexión, que une a los procesadores y las unidades de memoria, ubicados en los extremos superiores e izquierdos. Cada switch puede tanto permitir a una señal pasar en dirección vertical u horizontal simultáneamente, como redirigir una señal de vertical a horizontal o viceversa.

Como la interconexión entre los procesadores y las unidades de memorias es total, una comunicación entre un par de ellos no interfiere con los intentos de otros procesadores para conectarse a las unidades de memoria restantes, lo que evita los problemas de saturación del esquema basada en la utilización de un bus. Sin embargo, la gran cantidad de switches necesarios para la completa interconexión de las unidades (en un crossbar switch de mxn se utilizan mn switches) los hace muy caros, y los limita a una cantidad pequeña de procesadores.

Una solución de compromiso entre el bus y el crossbar switch, consiste en utilizar un esquema de crossbars en múltiples niveles, con lo que se obtiene una red con complejidad de

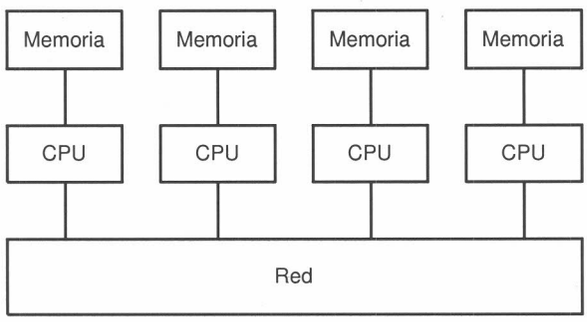
acceso logarítimica (utilizado, entre otras, en las IBM RS/6000 SP y las SGI Origin 2000)[3] En este caso, los switches en los niveles intermedios deben trabajar velozmente para cubrir el ancho de banda requerido por el traslado de los datos desde la memoria a los procesadores.

Sistemas MIMD de memoria distribuida

En el caso de los sistemas MIMD de memoria distribuida (**DM-MIMD**), cada procesador tiene una memoria asociada. Los CPUs son conectados por medio de una red y pueden intercambiar datos entre sus respectivas memorias cuando es necesario.

Aunque inicialmente no fueron considerados como la mejor alternativa para el desarrollo de soluciones paralelas, existe actualmente una intensa corriente hacia su utilización. Según [28], debido al desarrollo en los últimos años de software de comunicaciones standard (como MPI [27] y PVM [9]) y porque, al menos teóricamente, esta clase de computadoras es capaz de sobrepasar el rendimiento de cualquiera de las otras.

En contraste con los sistemas SM-MIMD, el usuario debe estar consciente de la ubicación de los datos en las memorias locales ya que en caso en el caso en que un procesador necesite utilizar información que no se encuentre en su memoria local, los datos deberán ser transmitidos explícitamente por medio de la red de interconexión.[4]



En consecuencia, un sistema DM-MIMD presenta un acesso a memoria de dos niveles, al primero se accede a una gran velocidad, y corresponde a la memoria local de cada procesador; mientras que el segundo es, al compararlo con el primero, varias veces más lento. Aventajando a los sistemas SM-MIMD, podemos observar que cuando la tarea ejecutada en cada procesador utiliza únicamente datos de la memoria local se puede alcanzar un rendimiento mayor. Por otra parte, la necesidad de comunicarse con el resto de los procesadores para intercambiar datos requiere de una red de interconexión eficiente.

---

[3]En contraste con la solución de basada en un crossbar switch (de $O(n^2)$), el esquema se destaca al incrementar la cantidad de procesadores, ya que la cantidad de conexiones es $O(n \log_2 n)$.

[4]El enfoque desde el punto de vista del usuario puede variar por medio de la implementación (existente tanto en hardware como en software) de un sistema de *memoria compartida virtual*, que presente al usuario un espacio de direcciones global único, comprendiendo las memorias locales de cada uno de los procesadores conectados.

Una dificultad exclusiva de estos sistemas es el desajuste entre los tiempos de comunicación y la velocidad de los procesadores para el caso en que estos últimos sean mejorados sin incrementar a la vez la performance de la red de interconexión. En algunas situaciones, esto puede convertir un problema limitado por la capacidad computacional en uno limitado por la capacidad comunicacional.

Como en el caso de los sistemas SM-MIMD, nuevamente es importante considerar la topología y velocidad de los caminos que recorren los datos. Aunque existen una gran cantidad de estructuras de interconexión posibles, sólo unas pocas son populares en la práctica. Uno de ellos es el denominado *hipercubo*, donde en un sistema con 2^d procesadores, exite un camino entre cualquiera de ellos de, a lo sumo, d nodos.[5] Otra gran cantidad de sistemas emplean un *mesh* de 2d o 3d, debido a que una gran cantidad de problemas pueden ser trasladados eficientemente a esta topología.

Finalmente, una gran proporción de sistemas utilizan *crossbar*s. En el caso de poseer una pequeña cantidad de procesadores, este puede ser directo (o de sólo un nivel). Mientras que para conectar una mayor cantidad de nodos, se establecen múltiples niveles, donde los *crossbars* del nivel 1 están conectados a *crossbars* de nivel 2, en lugar de estar directamente conectados a otros nodos. De este modo, se pueden conectar muchos procesadores en pocos niveles.

1.2.6 Pile-of-PCs

Becker [25] observa que en la última década el tamaño de los circuitos integrados se ha reducido en cada dimensión en un orden de magnitud, obteniendo por resultado un incremento de tres ordenes de magnitud en la cantidad de dispositivos por chip, y una mejora similar en la velocidad total del chip. Además, Becker resalta que, en cinco años, mientras que la performance de los procesadores de workstations se incrementó aproximadamente un 50% cada año, los procesadores personales han superado un factor de dos en cada uno de los últimos cuatro.

Tal ritmo de evolución en el mercado de computadoras personales ha dado por resultado que el nivel de performance de sus procesadores se superponga con el de las workstations, mientras al mismo tiempo bajan sus precios debido al volumen de producción de los componentes para ese mercado.

Aparece, entonces, la oportunidad de alcanzar un poder de cálculo antes sólo disponible en máquinas especializadas a partir del desarrollo de una red o, de manera más precisa, un cluster de computadoras personales. Más aún, los sensiblemente inferiores costos de éstas últimas configuraciones permite que ellos puedan ser adquiridos por el investigador individual (o grupo de investigación).

Podemos describir esta arquitectura como una extensión del concepto de DM-MIMD donde, en lugar de integrar procesadores en una o más computadoras, estaciones de trabajo o computadoras personales son conectadas por medio de una red de interconexión (como

[5] En un hipercubo es posible simular cualquier otra topología (como un *árbol*, *anillo*, o *mesh* de 2d o 3d).

Ethernet o FDDI) para realizar concurrentemente tareas correspondientes a un mismo programa. Esto no es conceptualmente diferente a DM-MIMD, sin embargo, la comunicación entre los procesadores es generalmente varios órdenes de magnitud más lenta.

El término *Pile-of-PCs* es utilizado para describir a un cluster de PCs aplicados en forma conjunta para resolver un problema particular. Es conceptualmente similar a un COW (Cluster Of Workstations) o un NOW (Network Of Workstations), pero en cambio se enfatiza, con el objetivo de alcanzar una mejor relación costo/performance:

1. El uso de componentes *commodity* del mercado masivo.
2. Procesadores dedicados (en lugar de utilizar ciclos ociosos de un grupo de workstations).
3. Un área de red privada para el sistema.

El enfoque de *Pile-of-PCs* aprovecha los componentes que responden a standards aceptados ampliamente por la industria y se beneficia de los precios resultantes de una dura competencia y la producción en masa. Otra ventaja comparativa es que no existe un único proveedor que posee los derechos sobre el producto, ya que mucho de ellos proveen subsistemas esencialmente idénticos, como motherboards, controladores de periféricos, dispositivos de entrada/salida, entre otros. Subsistemas que, a su vez, proveen interfaces standard como el bus PCI, las interfaces IDE y SCSI, y Ethernet entre las de comunicaciones.

Además, permite realizar un seguimiento de la tecnología, ya que se puede disponer de las últimas actualizaciones en cada componente y de los mejores precios, que cambian en forma constante varias veces en el año. Lo que lleva a una ventaja adicional, la de implementar una configuración específica de acuerdo a las necesidades del usuario. En contraposición con el enfoque de un proveedor que dispone de una lista de opciones que puede no haber evolucionado en una dirección adecuada para el usuario, la flexibilidad de un *Pile-of-PCs* permite enfocar y desarrollar en el tiempo la configuración adquirida.

1.2.7 Beowulf

Se denomina *Beowulf* a una computadora paralela elaborada con componentes de bajo costo y moderada performance disponibles en el mercado de computadoras personales, junto a los standards *de facto* en hardware y software para pasaje de mensajes.

Su origen se encuentra en el proyecto del mismo nombre comenzado en 1994 con el apoyo del proyecto "HPCC (High Performance Computing and Communications) Earth and Space sciences" de la NASA, cuyo objetivo fue investigar el potencial de los clusters de PCs para realizar tareas computacionales importantes más allá de la capacidad de las workstations contemporáneas pero no a mayor costo. El primer logro del proyecto se alcanzó en noviembre de 1996, cuando se anunció ([22]) que un sistema Beowulf había superado en performance sostenida un Gigaflops[6] en una aplicación de la ciencia espacial con un costo menor a los 50.000 dólares.

[6]Flop: operación aritmética de punto flotante.

La posibilidad de incorporar una gran cantidad de computadoras para elaborar un cluster de alta performance ha permitido que computadoras paralelas basadas en Beowulf hayan logrado entrar en el ranking de las 500 supercomputadoras más rápidas del mundo[7].

Para definir un Beowulf debemos, según [25], agregar al modelo de *Pile-of-PCs*, descripto en la sección anterior, las siguientes condiciones:

1. No usar componentes específicos para el sistema.
2. Fácil replicación a partir de múltiples proveedores.
3. Subsistema de entrada/salida escalable.
4. Software de base disponible en forma gratuita.
5. Uso de herramientas de programación distribuida gratuitos con cambios mínimos.
6. Devolver el diseño y las mejoras a la comunidad.

Un Beowulf aprovecha los sistemas de software disponibles actualmente, generalmente gratuitos y a la vez tan sofisticados, robustos y eficientes como los comerciales. El software deriva de esfuerzos de toda la comunidad en sistemas operativos, lenguajes y compiladores, y bibliotecas de programación paralela; con un nivel comparable en casi todos los casos al que ofrecen los proveedores comerciales.

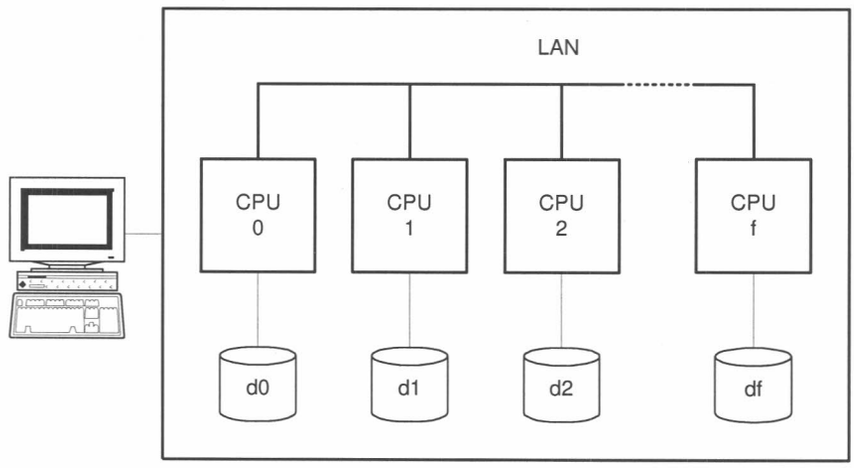


Figura 1.6: Arquitectura de workstation paralela Beowulf

El sistema operativo más usado en estos proyectos es Linux, un sistema Unix/Posix disponible tanto en la red en forma gratuita como en distribuciones comerciales con servicio

[7] ver `http://www.top500.org`

de soporte, una completa implementación de X Windows, y compiladores standard para la mayor parte de los lenguajes de programación. Además, las dos principales bibliotecas de pasaje de mensajes (ver sección 1.4), PVM y MPI, están disponibles para estos sistemas y son ampliamente utilizadas por toda la comunidad.

1.3 Performance

1.3.1 Introducción

El desarrollo de un programa paralelo tiene como objetivo no sólo optimizar una única métrica, como el tiempo de ejecución. Un buen diseño debe además optimizar una función dependiente del problema a resolver donde se tienen en cuenta el tiempo de ejecución, los requerimientos de memoria, los costos de implementación y mantenimiento de la la solución, entre otros. Estas optimizaciones en el diseño conllevan una tensión entre simplicidad, performance, portabilidad y otros factores.

Por lo tanto, poder decidir entre distintas alternativas requiere una comprensión de sus costos. En las siguientes secciones se presentarán las herramientas fundamentales para estudiar el comportamiento de un programa paralelo, comparando su eficiencia con programas seriales que resuelvan el mismo problema. Estas nos permitirán encontrar modelos del desempeño de una solución particular *antes* de implementarla.

1.3.2 Performance de programas seriales - Análisis de la complejidad

Para poder definir fórmulas que nos permitan evaluar la performance de un programa paralelo debemos extender aquellas que corresponden al análisis de la complejidad computacional de un programa serial.

Una forma particular de encarar su estudio es concentrarse en la performance del *peor caso* para poder determinar la dependencia, ignorando factores constantes, entre el tiempo de ejecución y el tamaño del problema. Una *función de complejidad temporal* para un algoritmo es una función que depende del tamaño del problema y especifica el máximo tiempo requerido por el algoritmo para resolver cualquier instancia del problema con ese tamaño. En otras palabras, la complejidad temporal mide la tasa de crecimiento para la solución de un problema a medida que su tamaño aumenta.

Se dice que un algoritmo corre en tiempo $O(f(n))$ si para ciertos números c y n_0, el tiempo que toma el algoritmo es como máximo $cf(n)$ para todo $n \geq n_0$. De acuerdo a la definición, la complejidad de un algoritmo es una cota superior en el tiempo de ejecución del algoritmo para valores suficientemente grandes de n. Por lo tanto, esta medida de complejidad nos muestra la tasa de crecimiento asintótico del tiempo de ejecución.

1.3.3 Análisis de performance de un programa paralelo

Resulta inmediato observar que todo intento de estudiar la performance de un programa paralelo debe depender, además del tamaño del problema (como para un programa serial), de la cantidad de procesadores utilizados para resolverlo, por lo que denominamos $T(n, p)$, a la función de dos variables que denota la performance del programa.

Para poder comparar el rendimiento del programa paralelo en relación con uno serial se suele utilizar dos medidas conocidas como: *speedup* y *eficiencia*. El *speedup* es la razón entre el tiempo de ejecución del programa paralelo y aquel requerido por una solución serial al mismo problema. Por lo tanto, sea $T_1(n)$ el tiempo de ejecución de la solución serial y $T_p(n,p)$ el tiempo de ejecución de la solución paralela utilizando p procesadores, el *speedup* del programa paralelo es:

$$S(n,p) = \frac{T_1(n)}{Tp(n,p)}$$

La *eficiencia* es una medida de la utilización de los procesadores en un programa paralelo en comparación con el programa serial, y se define:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_1(n)}{pT_p(n,p)}$$

De acuerdo a lo que se ha visto anteriormente, en general $0 < S(n,p) \leq p$, entonces resulta que $0 < E(n,p) \leq 1$. Por lo tanto, si $E(n,p) = 1$, el programa posee un *speedup lineal*, mientras que si $E(n,p) \leq 1/p$, el programa posee *slowdown*.[8]

Overhead

El *trabajo* realizado por un algoritmo secuencial es el tiempo que requiere su ejecución: $W_1(n) = T_1(n)$. En el caso de un algoritmo paralelo, es la suma del trabajo de cada uno de los procesadores: $W_p(n,p) = \sum_{i=1}^{p} W_i(n,p)$. Como los tiempos de espera en cada procesador forman parte de su trabajo, esto implica que $W_i(n,p) = T_p(n,p)$, y entonces:

$$W_p(n,p) = \sum_{i=1}^{p} W_i(n,p) = \sum_{i=1}^{p} T_p(n,p) = pT_p(n,p).$$

Al buscar una solución paralela a un problema computacional encontramos que, respecto a una versión secuencial, distintos factores actúan como sobrecarga en el tiempo de ejecución del algoritmo, entre los que podemos destacar:

- Períodos donde no todos los procesadores pueden estar trabajando en la resolución del problema (como en partes inherentemente seriales del problema).
- Cálculos adicionales requeridos en la versión paralela respecto de la secuencial.
- Tiempo de comunicación para la transmisión de mensajes.

Definimos entonces el *overhead* de un algoritmo paralelo como el trabajo adicional que realiza respecto a una solución secuencial:

$$T_O(n,p) = W_p(n,p) - W_1(n) = pT_p(n,p) - T_1(n)$$

[8]Existen casos en que $E(n,p) > 1$ (es decir, $S(n,p) > p$), lo que denominamos *speedup superlineal*. Generalmente se debe al uso de un algoritmo secuencial subóptimo o alguna característica de la arquitectura paralela que favorece el desempeño de la solución sobre la misma (por ejemplo, disponer de una mayor cantidad de memoria en cada procesador permite minimizar la transferencia de datos a disco y aumentar la performance global de un algoritmo).

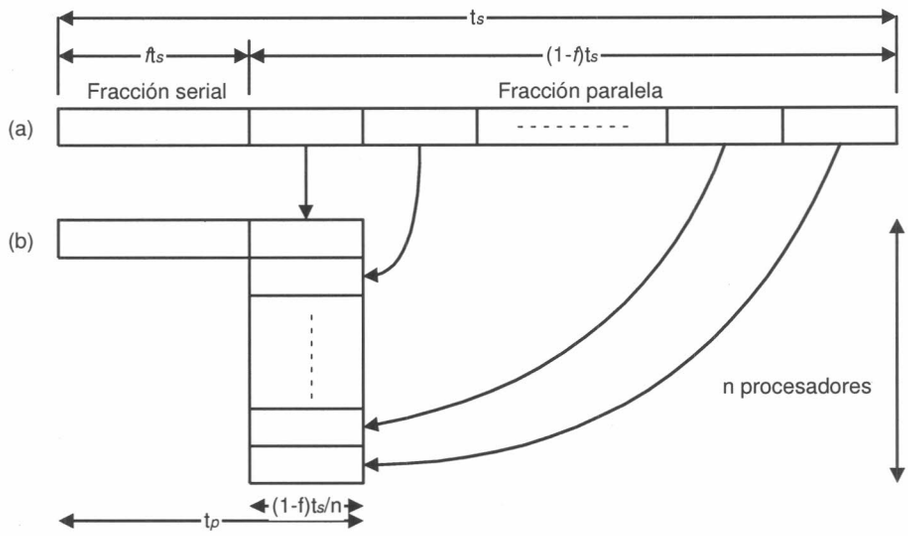


Figura 1.7: Ley de Amdahl, se ilustra el efecto de las fracciones: secuencial y paralela. (a) Tiempo de ejecución secuencial, (b) Tiempo de ejecución paralelo.

1.3.4 Ley de Amdahl

Al analizar todo algoritmo paralelo podemos encontrar que al menos una parte del mismo es secuencial. Ésta, según [1], limita el *speedup* que puede obtenerse mediante la utilización de una computadora paralela. La *ley de Amdahl* se puede expresar de la siguiente manera, si la fracción del problema que es inherentemente secuencial es f, entonces el máximo speedup que se puede obtener en una computadora paralela es $1/f$. En la fig. 1.7 se muestra que la fracción paralela de un problema puede realizarse por varios procesadores, y que la fracción secuencial f limita el tiempo requerido para resolver el problema aún cuando podamos utilizar más procesadores.

Es decir, el tiempo de ejecución de la solución paralela será: $T_p(n,p) = fT_1(n) + (1 - f)T_1(n)/p$, y por lo tanto el factor de *speedup* es:

$$S(n,p) = \frac{T_1(n)}{Tp(n,p)} = \frac{T_1(n)}{fT_1(n)+(1-f)T_1(n)/p} = \\ = \frac{1}{f+(1-f)/p} = \frac{p}{1+(p-1)f}$$

Luego, aún con una cantidad infinita de procesadores el máximo *speedup* posible está limitado por $1/f$:[9]

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

[9]A su vez, la eficiencia de un programa paralelo se acercará a cero a medida que se incorporen nuevos procesadores.

Por lo tanto, de acuerdo a esto, si el problema es sólo 5% serial, el máximo *speedup* que podremos obtener en nuestra solución paralela es 20, independientemente de la cantidad de procesadores que utilizemos. En un caso extremo, con un problema sólo 1% serial, aún utilizando 10.000 procesadores, el máximo speedup obtenible es 100, que corresponde a una eficiencia de 0,01.

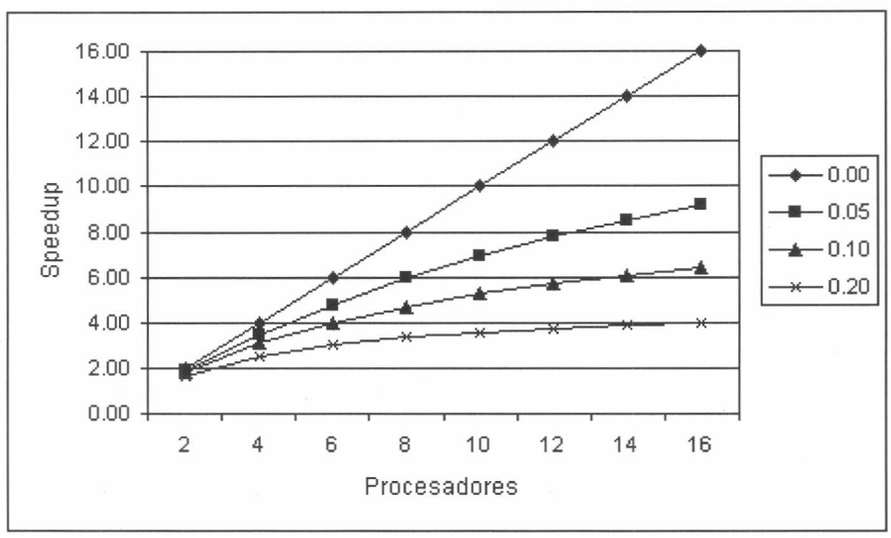


Figura 1.8: Ley de Amdahl. Speedup en función de la cantidad de procesadores, para distintos valores de fracción secuencial.

Aunque desde su publicación se pensó que sus conclusiones limitarían la utilidad de las computadoras paralelas, fue reevaluada a partir de los trabajos de [13] y [12] donde se sugiere, como se verá al introducir la idea de *escalabilidad* de un programa paralelo, que los problemas a resolver deben ser más grandes a la vez que incorporamos más procesadores a su resolución.

Se puede aplicar la ley de Amdahl al caso de paralelizar un programa en múltiples etapas. En este enfoque para el desarrollo de una aplicación paralela, se mide el rendimiento de distintas partes de un programa secuencial para detectar aquellas que demandan más capacidad de procesamiento. El costo computacional de la fracción secuencial remanente (que no es paralelizada) permite prever un límite inferior respecto al tiempo de ejecución del programa paralelo a desarrollar.

1.3.5 Escalabilidad

El término *escalabilidad* es impreciso, ya que se le ha asociado más de un significado. En [29] se sugieren dos categorías, la primera hace referencia a que se suele utilizar para indicar

que un diseño de hardware particular permite que al incrementar el tamaño del sistema, se obtenga una mejora en su performance. Denomina a este caso como *escalabilidad de arquitectura*. Por otra parte, también se puede usar para decir que un algoritmo paralelo puede manejar una mayor cantidad de datos con un incremento bajo y acotado en los pasos computacionales, y se lo llama, *escalabilidad algorítmica*.

Idealmente toda computadora paralela debería tener escalabilidad de arquitectura, sin embargo esto depende del diseño del sistema. En general, a medida que agregamos procesadores a un sistema, la red de interconexión debe ser extendida. Y, como consecuencia de estos cambios, mayores demoras en los tiempos de comunicación entre ellos disminuirán la eficiencia y, por ende, su escalabilidad. Por ejemplo, en el caso de un cluster Beowulf que inicialmente interconectaba las workstations disponibles por medio de un único switch, si al incorporar nuevas máquinas se supera la cantidad de conexiones disponibles, será necesario implementar un sistema de switches de múltiples niveles (como en el caso de los sistemas DM-MIMD con crossbars de 2 niveles presentados en la sección 1.2.5 - *Computadoras paralelas - Sistemas MIMD de memoria distribuida*). Llegado un punto, los tiempos de comunicación entre máquinas ubicadas a la mayor distancia entre niveles deteriora la performance global del sistema, mientras que el costo de la interconexión completa de cada una de las workstations será generalmente inviable.

1.3.6 Ley de Gustafson

En [13] y [12], Gustafson presenta un argumento basado en el concepto de escalabilidad para mostrar que la ley de Amdahl no es tan significativa como era esperado, en limitar el speedup potencial de una solución paralela. En su trabajo, parte de la observación que una computadora paralela más grande nos permitirá afrontar un problema de mayor tamaño en un tiempo razonable[10]. Es decir, en la práctica, el tamaño del problema no es independiente de la cantidad de procesadores utilizados para resolverlo. Por lo tanto, sugiere reemplazar esta restricción por el tiempo de ejecución, que se mantendrá fijo mientras se incrementa el tamaño del problema y los procesadores de la computadora paralela.

Manteniendo constante el tiempo de ejecución, entonces, el factor de speedup resultará diferente al que define Amdahl y es denominado *factor de speedup escalado*.

Siguiendo la notación utilizada por Gustafson, sea s la parte serial de nuestro algoritmo y p la que corresponde a la paralelizada. Si fijamos como constante el tiempo de ejecución en un procesador en $s + p$ y, para simplificar la explicación, asumimos que $s + p = 1$, entonces la ley de Amdahl establece (ahora n es la cantidad de procesadores a utilizar):

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

Para utilizar el factor de speedup escalado de Gustafson, el tiempo de ejecución en paralelo es constante 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

[10]Se da, entonces, la combinación de las dos categorías de escalabilidad presentadas en la sección anterior.

paralela, el mismo que utilizamos en la ejecución serial original, y para simplificar nuevamente establecemos que $s+p = 1$. Entonces, el tiempo que requiere la ejecución en sólo una máquina para resolver el mismo problema es $s + np$, ya que la parte paralela debe ejecutarse en forma secuencial. El speedup escalado será:

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

En lugar de una curva de rápido descenso como en el caso de la Amdahl, la observación de Gustafson es que el factor de speedup escalado como función de la parte serial s, es una línea de pendiente (negativa) $1 - n$. Bajo las diferentes suposiciones de cada caso, en un problema con una fracción serial del 5% resuelto con 20 procesadores, obtenemos un speedup escalado de $0,05 + 0,95 \cdot 20 = 19,05$ mientras que el speedup de acuerdo a la ley de Amdahl será de $10,26$.

Gustafson cita en [12] casos con speedups de 1021, 1020 y 1016 alcanzados en la práctica con un sistema de 1024 procesadores en problemas de simulación numérica.

Eficiencia y escalabilidad

Analizaremos ahora cómo se comporta la eficiencia de un algoritmo paralelo, al aumentar el tamaño del problema y agregar más procesadores para resolver un problema computacional. Según se ha definido, la eficiencia es:

$$E(n, p) = \frac{T_1(n)}{pT_p(n,p)}$$

Luego, visto de acuerdo al *overhead*, como $T_O(n, p) = pT_p(n, p) - T_1(n)$:

$$E(n, p) = \frac{T_1(n)}{T_O(n,p)+T_1(n)} = \frac{1}{T_O(n,p)/T_1(n)+1}$$

De este modo, la eficiencia de un algoritmo queda determinado por su *overhead* respecto de la versión serial.

Para determinar la escalabilidad de un programa paralelo deberemos encontrar una tasa de crecimiento para el tamaño del problema (n) en función de la cantidad de procesadores (p), que mantenga constante la eficiencia de la solución. Realizaremos esta tarea al analizar nuestra implementación paralela de la simulación del problema de electrodeposición, con lo que podremos prever el rendimiento de la solución en el caso de incorporar nuevas computadoras al Beowulf.

1.4 Message passing (Pasaje de mensajes)

Para implementar un algoritmo paralelo en una máquina MIMD de memoria distribuida se suele utilizar el paradigma de *message passing* (o *pasaje de mensajes)*. El concepto básico de este paradigma consiste en que un grupo de procesos se comunican entre si por medio del intercambio de mensajes para realizar una tarea determinada.

Los procesos que componen la implementación paralela de un problema computacional pueden ser creados en forma estática o dinámica. En general, en el primero de los dos enfoques se utiliza un único programa que se ejecuta en cada proceso (SPMD: *single-program multiple-data*), y estos son creados al momento de comenzar la ejecución del programa; mientras que en el método de creación dinámica de procesos un programa principal se encarga de crear a procesos que llevarán a cabo parte del trabajo computacional del problema (MPMD: *multiple-program multiple-data*), cada uno de dichos procesos puede crear otros, y también terminarlos, de acuerdo a sus requerimientos durante la ejecución.

Actualmente se destacan dos especificaciones standarizadas (con implementaciones en una gran variedad de sistemas operativos y arquitecturas de procesadores) que reunen la mayor cantidad de desarrollos dentro del paradigma de *message passing*, estos son MPI (*Message Passing Interface*) [27] y PVM (*Parallel Virtual Machine*) [9]. El primero de ellos responde al modelo de creación de procesos en forma estática o SPMD, mientras que el segundo es un ejemplo de un sistema del modelo MPMD. En [10] y [11] se comparan ambas especificaciones.

PVM se destaca por permitir trabajar en ambientes heterogéneos, ya que provee la posibilidad de interoperar entre máquinas que corresponden a distintas arquitecturas.[11] Además, dispone de funcionalidad para desarrollar aplicaciones tolerantes a fallos, tanto de máquinas como de procesos.

El grupo encargado de desarrollar el standard MPI, por su parte, se enfocó en proveer una biblioteca que pudiera ser implementada en el hardware provisto por las distintas empresas participantes de la especificación y que permitiera alcanzar la más alta performance disponible en cada una de sus arquitecturas. Por lo tanto, MPI no requiere que las implementaciones sean interoperables ni tampoco asume el costo del ambiente de una máquina virtual que maneja procesos dinámicamente.[12] Por el contrario, provee una mayor variedad de opciones para realizar comunicaciones punto-a-punto y grupales, que permiten explotar las características del hardware disponible y a la vez brindan mayor flexibilidad en el desarrollo de algoritmos paralelos.

Además, con MPI se puede especificar una topologia de comunicación lógica, que facilita la implementación de algoritmos paralelos existentes para arquitecturas diferentes a la de la máquina paralela a utilizar.

En nuestro trabajo utilizamos MPI debido, principalmente, a su eficiencia, y también

[11] Lo que induce en costos relativos a la conversión de los datos enviados entre procesos corriendo en arquitecturas diferentes.

[12] Aunque en la especificación MPI-2 se proveen rutinas para iniciar procesos, aunque no tan flexibles como las disponibles en PVM.

por la diversidad de opciones en la comunicación entre procesos. Los beneficios de utilizar PVM no se aplican en nuestro caso, dado que nuestro trabajo se desarrolla sobre un cluster de procesadores de una misma arquitectura.

En las próximas secciones, entonces, utilizaremos las rutinas de MPI para mostrar las operaciones básicas disponibles en el paradigma de *message passing*.

1.4.1 Transmisión de datos punto-a-punto

El envío y recepción de mensajes para realizar el intercambio de datos entre procesos consta principalmente de un par de funciones dentro del standard MPI. La función para enviar un mensaje es:

```
MPI_Send(buffer, count, datatype, dest, tag, comm)
```

Y la función para recibir un mensaje es:

```
MPI_Recv(buffer, count, datatype, source, tag, comm, status)
```

Los parámetros indican que en `buffer` se transmiten `count` elementos del tipo de datos `datatype`, desde el nodo `source` al nodo `dest`.[13]

Para lograr una mayor flexibilidad y eficiencia en los programas paralelos, se disponen de una variedad de mecanismos para las funciones de envío/recepción de datos.

La transmisión de datos en forma *sincrónica* es utilizada por funciones que retornan sólo cuando la transferencia ha sido completada. Estas funciones no necesitan disponer de buffers de almacenamiento debido a que el usuario no puede modificar los datos a enviar mientras se realiza la transferencia debido a que no dispone del control de la ejecución del programa. Una rutina de envío de datos sincrónica esperará hasta que el receptor esté disponible, mientras que una rutina de recepción de datos sincrónica sólo retornará cuando los datos esperados arriben. Por lo tanto, la utilización de rutinas sincrónicas para el envío y recepción de un mensaje produce la sincronización de los procesos involucrados, lo que se denomina *rendezvous*.

Al utilizar funciones de envío o recepción de datos que no esperan la finalización de sus tareas podemos realizar las operaciones de transmisión en forma *asincrónica*. Al enviar datos de este modo, el control será devuelto una vez que se registre la tarea con el mensaje a enviar, pero no se deberá esperar a que el receptor este disponible y se complete la transferencia. De manera similar, la recepción asincrónica de mensajes permite realizar tareas mientras se esperan los datos (requiere además de la utilización de otras funciones para determinar si la transferencia ha finalizado).

En MPI se define además la transmisión de datos en forma *bloqueante* con la utilización de un buffer al que se copian los datos a transferir en la operación de envío. En el caso de utilizar rutinas *no bloqueantes*, es el usuario quien debe garantizar que los datos a enviar no serán modificados hasta que se finalice la transmisión.

[13] El parámetro `tag` permite al usuario diferenciar diversas clases de mensajes transmitidos entre un mismo par de procesos.

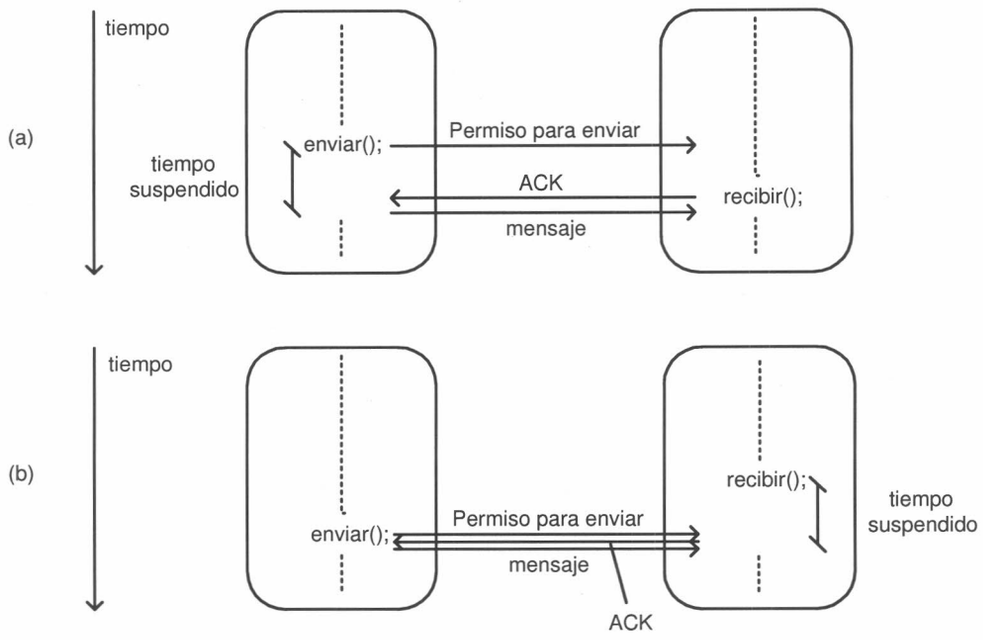


Figura 1.9: Transmisión sincrónica de datos. En (a) la operación `enviar()` se ejecuta antes que `recibir()`, mientras que en (b) ocurre lo contrario.

El costo de comunicación

Se ha mencionado, al describir las distintas variedades de arquitecturas paralelas, que el tiempo requerido por la comunicación entre los procesadores de un sistema de memoria distribuida es marcadamente superior al que corresponde a las operaciones de cálculo (hasta en varios órdenes de magnitud). Por lo tanto, para obtener aproximaciones razonables al tiempo de ejecución de un algoritmo paralelo se debe considerar el costo de comunicación en forma independiente[14], es decir:

$$T(n,p) = T_{calc}(n,p) + T_{comm}(n,p)$$

A su vez, el análisis del tiempo requerido para realizar las operaciones de intercambio de datos entre procesadores depende del tipo a que correspondan. El caso más simple y, a la vez, más utilizado corresponde al envío y recepción de datos entre dos procesos cualesquiera. La ejecución de la tarea se puede dividir en tres fases:

1. *Inicialización*: donde se suele copiar el mensaje a un área controlada por el sistema,

[14]De la misma forma se debe considerar al costo de las operaciones de entrada/salida. Sin embargo, como en nuestro programa esos tiempos son mínimos (debido a que los resultados se guardan a intervalos muy espaciados de cálculo), no se incluirá dicho factor en el análisis de la performance del mismo.

en la que se almacenan el identificador de los procesadores de origen y destino junto con los datos a transmitir y, en algunos casos, información adicional.[15]

2. *Comunicación*: en la que se realiza la transmisión efectiva de los datos entre los procesadores.

3. *Finalización*: es análoga a la primera, donde se copian los datos del área de memoria del sistema en donde se recibieron los datos (en la fase de *Comunicación*) a un área del usuario para su posterior utilización.[16]

Entonces, si denominamos al tiempo de inicialización como t_s, en el que se tiene en cuenta además el tiempo requerido por la fase de finalización[17], y a t_c como el tiempo que toma transmitir cada unidad de datos de un procesador a otro, el costo correspondiente a enviar un mensaje que comprende k unidades de datos entre dos procesadores será:

$$t_s + kt_c$$

Generalmente se llama *latencia* a t_s y al recíproco de t_c se puede hallar descripto como *ancho de banda*. Según [23] y [6] es difícil encontrar estimaciones confiables tanto de t_s como de t_c, dependiendo los resultados de cada sistema en particular. Sin embargo, realizando una amplia generalización, en la mayoría de los casos t_c está a alrededor de un orden de magnitud del costo de una operación aritmética, t_a, y t_s es entre uno y tres ordenes de magnitud superior a t_c.

Asintóticamente, para valores grandes de k, sólo el valor de t_c es importante, pero como generalmente t_s es superior en más de un orden de magnitud a t_c, la latencia predominará en aplicaciones que transmiten mensajes de tamaño reducido.

En la siguiente tabla se muestran valores (en microsegundos) presentados en [23], correspondientes a una amplia variedad de computadoras paralelas, que permiten verificar las diferencias entre t_s y t_c según la tecnología de interconexión de los procesadores.[18]

---

[15] En MPI, por ejemplo, el *tag* para diferenciar distintas sesiones de transmisión de datos entre procesos.

[16] Esta fase es opcional, ya que en algunos casos la recepción de los datos enviados por un procesador pueden ser recibidos directamente en el área de memoria reservada a esos efectos por el usuario.

[17] Ya que la complejidad de ambas operaciones es semejante. Por lo tanto, la diferencia será tan sólo de un factor constante.

[18] En la práctica, el tiempo de transmisión como función del tamaño del mensaje no será estrictamente una recta (como supone la fórmula) sino que presenta irregularidades debido a detalles de los protocolos de comunicación y de la estrategia de manejo de buffers utilizada en las bibliotecas de comunicación. Sin embargo, la fórmula provee una representación razonablemente precisa, especialmente para los mensajes de mayor tamaño.

Computadora	Sistema operativo	t_s	t_c
Cray T3D (PVM)	MAX 1.2.0.2	21	0.30
Cray T3D (SM)	MAX 1.2.0.2	3	0.063
Intel Paragon	OSF 1.0.4	29	0.052
Intel iPSC/860	NX 3.3.2	65	2.7
Intel iPSC/2	NX 3.3.2	370	2.9
IBM SP-1	MPL	270	1.1
IBM SP-2	MPI	35	0.23
Meiko CS2 (SM)	Solaris 2.3	11	0.20
Meiko CS2	Solaris 2.3	83	0.19
nCUBE 2	Vertex 3.2	170	4.7
TMC CM-5	CMMD 2.0	95	0.89
Ethernet	TCP/IP	500	8.9

1.4.2 Empaquetado (ó "packing")

Los métodos de transmisión de datos presentados en la sección anterior permiten enviar o recibir una secuencia de elementos idénticos en posiciones contiguas de memoria. En algunos casos es necesario enviar datos que no cumplen con alguna de estas dos características (o ambas a la vez), en una única transmisión para amortizar el costo fijo correspondiente a la latencia.

MPI provee esta funcionalidad por medio de dos mecanismos:

1. El usuario puede definir tipos de datos derivados, que especifican una distribución de datos en memoria más general. Estos tipos de datos pueden ser luego utilizados en las funciones de comunicación en lugar de los tipos predefinidos.

2. Un proceso puede empaquetar en forma contigua áreas de datos no contiguos, para luego enviar un único conjunto de datos. Luego, el proceso receptor procede a desempaquetar los datos y distribuirlos a las posiciones de memoria que correspondan.

Las técnicas de empaquetado de datos serán utilizadas en el desarrollo de nuestro trabajo para minimizar el costo de comunicación entre los procesos, ya que nos permitirán enviar una menor cantidad de mensajes entre los procesos participantes de la solución, al consolidar la información a intercambiar correspondiente a los múltiples arreglos bidimensionales de datos utilizados.

1.4.3 Transmisión grupal (broadcast, scatter y gather)

Cuando se requiere el envío de información a más de un procesador se dispone de una variedad de funciones de comunicación global, entre las que se destacan:

Broadcast En una operación de *broadcast*, un proceso envia un conjunto de datos al resto de los interesados. La función provista por MPI es:

`MPI_Bcast(buffer, count, datatype, root`[19]`, comm`[20]`)`

Como la operación no será realizada hasta que todos los procesos que intervienen en la comunicación ejecuten la llamada a la función `MPI_Bcast`, este mecanismo actúa también sincronizando a estos procesos.

Scatter Se utiliza la función de *scatter* para enviar cada dato de un conjunto dado a un proceso distinto. De esta manera, el elemento i-ésimo del conjunto de datos será transmitido al proceso i-ésimo. La función provista por MPI es:

`MPI_Scatter(s_buf, s_count, s_type, r_buf, r_count, r_type, root, comm)`[21]

Por medio de este mecanismo se puede, por ejemplo, distribuir la información de configuración al momento de inicialización del programa a cada uno de los procesos.

Gather La operación de *gather* permite que un proceso reuna la información de un conjunto de procesos. Actuando en forma inversa que *scatter*, los datos del proceso i-ésimo son recibidos por el proceso `root` y ubicados en la i-ésima posición. La función provista por MPI es:

`MPI_Gather(s_buf, s_count, s_type, r_buf, r_count, r_type, root, comm)`

Este mecanismo suele ser utilizado luego de realizar cálculos sobre un conjunto de datos para guardar los resultados obtenidos.

Otras funciones provistas por MPI son: `MPI_Alltoall`, que permite enviar datos de todos los procesos a todos los procesos; y `MPI_Scan`, para realizar el cálculo de prefijos entre los procesos involucrados.

Reducciones globales MPI posee funciones que combinan la transmisión entre un conjunto de procesos: *broadcast*, *scatter* y *gather*, con una operación lógica o aritmética a efectos de realizar cálculos globales, por medio de las funciones: `MPI_Allreduce`, `MPI_Reduce_scatter` y `MPI_Reduce`, respectivamente.

Aunque las mismas operaciones se puede llevar a cabo enviando todos los datos a un único procesador que se encargue de ejecutar la tarea y luego distribuya los resultados, en casi todos los casos es posible utilizar algoritmos más eficientes (minimizando el tiempo de ejecución de la tarea).

---

[19] Identifica al proceso que se encargará de transmitir los datos que el resto de los procesos espera.

[20] Este parámetro identifica al grupo, previamente definido, de procesos que participan de la comunicación. MPI provee, con `MPI_COMM_WORLD`, un identificador de todos los procesos que participan en el programa.

[21] Los parámetros con prefijo `s_` determinan el conjunto de datos que será enviado en la operación (estos parámetros sólo son utilizados en el proceso `root`, e ignorados en el resto de los participantes), mientras que aquellos que comienzan con `r_` definen el conjunto que guardará los datos recibidos.

Entre las operaciones previstas en el standard de MPI se encuentran[22]:

Valor máximo	Suma	AND: lógico y bit-a-bit
Valor máximo y ubicación	Producto	OR: lógico y bit-a-bit
Valor mínimo		XOR: lógico y bit-a-bit
Valor mínimo y ubicación		

Performance de operaciones globales en MPI

El análisis de la performance de las operaciones globales (como `MPI_Reduce`, `MPI_Gather`, etc.) no es simple debido a que cada implementación puede decidir qué algoritmo utilizar para realizar su trabajo. Sin embargo, en general se puede esperar que éstas funciones hagan uso de eficientes algoritmos[23], obteniendo resultados similares a los de la siguiente tabla (donde N es la cantidad de palabras de datos a transmitir y P es el número de procesadores utilizados)[24]:

Operación	**Costo (para N chico)**	**Costo (para N grande)**
`MPI_Barrier`	$t_s \log P$	$t_s \log P$
`MPI_Bcast`	$\log P(t_s + t_c N)$	$2(t_s \log P + t_c N)$
`MPI_Scatter`	$t_s \log P + t_c N$	$t_s \log P + t_c N$
`MPI_Gather`	$t_s \log P + t_c N$	$t_s \log P + t_c N$
`MPI_Reduce`	$\log P(t_s + (t_c + t_{op})N)$	$2t_s \log P + (2t_c + t_{op})N$
`MPI_Allreduce`	$\log P(t_s + (t_c + t_{op})N)$	$2t_s \log P + (2t_c + t_{op})N$

[22] Además se dispone de un mecanismo extensible que permite que el usuario pueda implementar nuevas operaciones para ser utilizadas con la sintaxis standard.

[23] En [8] se describen diseños basados en hipercubos.

[24] Debe tenerse en cuenta que en algunas arquitecturas, limitaciones en ancho de banda pueden generar incrementos los tiempos, especialmente en el caso de los mensajes de gran tamaño.

1.5 Diseño de algoritmos paralelos

1.5.1 Introducción

En esta sección se describe una metodología presentada por [8] para el proceso de diseño de una solución paralela a un problema computacional, que será luego utilizada para el problema de electrodeposición. La metodología se caracteriza por enfocarse inicialmente en aquellos problemas que son independientes de la computadora paralela disponible, demorando hasta las últimas etapas del proceso de diseño los detalles de implementación específicos del sistema utilizado.

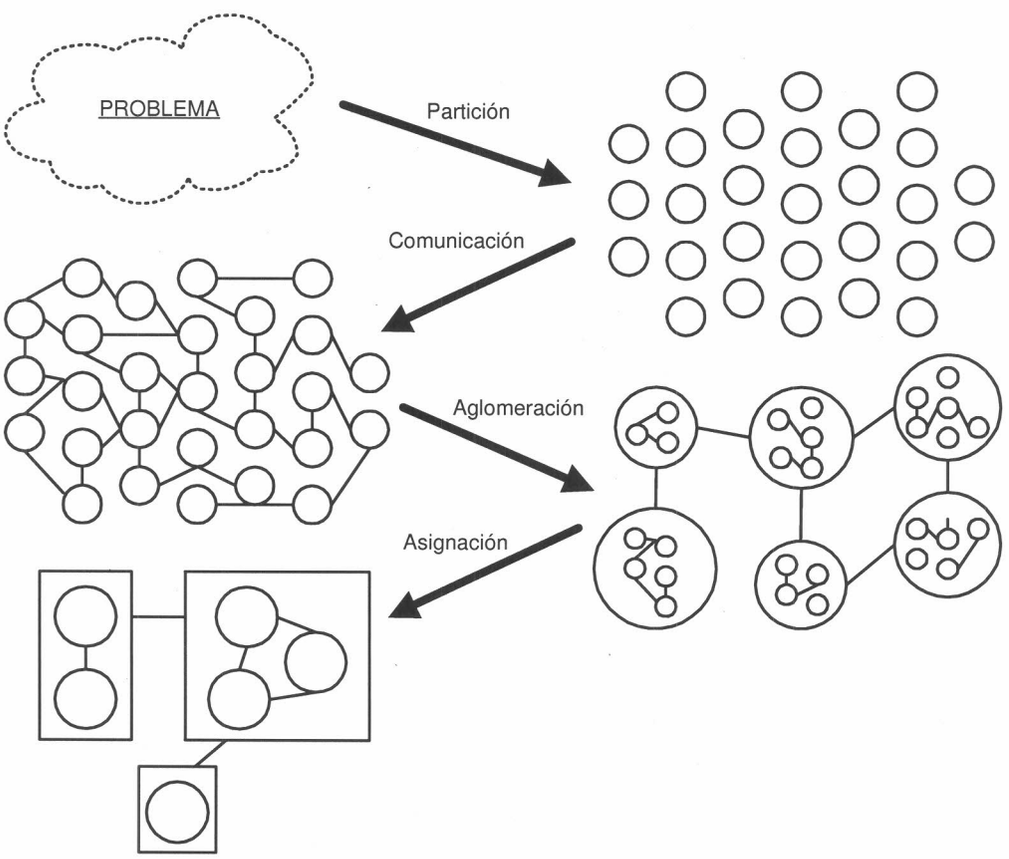


Figura 1.10: La metodología de diseño. A partir de la especificación de un problema, desarrollamos su partición, determinamos los requerimientos de comunicación, aglomeramos las tareas, y finalmente asignamos las tareas a los procesadores.

La metodología, representada en la Fig. 1.10, está compuesta por cuatro etapas:

1. *Partición*: Las tareas de cálculo que han de realizarse son divididas en otras de menor

tamaño. Se evita considerar cuestiones prácticas como la cantidad de procesadores disponibles, focalizando la atención en reconocer oportunidades de ejecución paralela.

2. *Comunicación*: Se determinan las comunicaciones requeridas para poder llevar a cabo las tareas, y se definen estructuras de comunicación y algoritmos para ejecutarlas.

3. *Aglomeración*: Las tareas y estructuras de comunicación definidas en las dos estapas anteriores son evaluadas en relación a los requerimientos de performance y los costos de implementación. Si es necesario, pueden agruparse tareas en unidades mayores para mejorar la performance o reducir los costos de desarrollo.

4. *Asignación (mapping)*: Cada tarea es asignada a un procesador intentando satisfacer los objetivos contrapuestos de maximizar la utilización de los procesadores y de minimizar los costos de comunicación.[25]

En las dos primeras etapas, la atención se concentra en descubrir algoritmos que permitan que la aplicación brinde el mayor grado posible de concurrencia y escalabilidad. Mientras que en las dos restantes, se traslada el énfasis al trabajo local correspondiente a cada procesador desde el punto de vista de la performance.

1.5.2 Partición

La obtención de las tareas a realizar por cada uno de los procesadores de la máquina paralela, cualquiera sea su arquitectura, requiere la división del problema en otros que les serán encomendados. Existen principalmente dos estrategias para llevar a cabo el particionamiento del trabajo:

1. *Paralelismo de datos:* En el enfoque de descomposición de datos, primero buscamos descomponer los datos asociados al problema. Si es posible, dividimos los datos en pequeñas partes de aproximadamente igual tamaño. Luego, particionamos el trabajo computacional a realizar, asociando cada operación a los datos sobre los que opera. La partición genera una cantidad de tareas, que comprenden un conjunto de datos y otro de operaciones sobre los datos. Las operaciones pueden requerir datos de diversas tareas, introduciendo una comunicación para mover los datos entre ellas.

   El esquema de *descomposición de dominios* se encuentra dentro del primer enfoque y busca descomponer los datos asociados al problema en partes más pequeñas de aproximadamente el mismo tamaño (para de este modo balancear la carga asociada a cada uno de los procesadores[26]).

[25] El *mapping* se puede realizar en forma estática o dinámica (por medio de algoritmos de balanceo de carga).

[26] Asumiendo un ambiente de procesadores homogéneo.

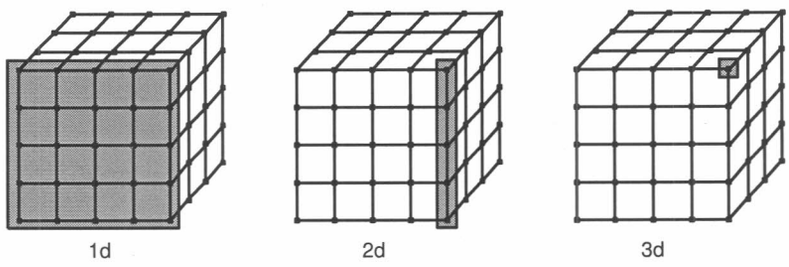


Figura 1.11: La figura ilustra la descomposición en dominios de un problema que utiliza una grilla tridimensional. Sobre cada punto de la grilla se realizan cálculos en forma repetida. En las primeras etapas del diseño favorecemos la descomposición más agresiva posible, que en este caso consiste en asignar a cada punto de la grilla una tarea. Cada tarea debe mantener los valores asociados con ese punto de la grilla y es responsable de los cálculos requeridos para modificarlos.

2. *Paralelismo de control:* La descomposición funcional representa una visión diferente sobre la resolución del problema. Inicialmente se enfoca en el trabajo computacional y luego en los datos que deben ser manipulados en las operaciones. Si se tiene éxito al dividir la computación en tareas disjuntas, se puede proceder a examinar los requerimientos de datos; que pueden ser disjuntos, lo que finaliza el trabajo de particionamiento; o pueden tener superposiciones, por lo que se necesitará comunicación para evitar la replicación de los datos (esta es una señal de que quizás sea necesario utilizar un enfoque de descomposición de datos).

Ambos criterios no son incompatibles sino que corresponden a puntos de vista complementarios y que pueden ser llevados a la práctica en algoritmos que los combinen.

1.5.3 Comunicación

Como se ha dicho en la sección anterior, aunque se busca que las tareas generadas en la etapa de partición se ejecuten en forma concurrente, en general no lo harán de modo independiente. Las tareas de computación a realizar requerirán datos asociados con otras tareas. Por lo tanto, los datos deberán ser transferidos entre las tareas para permitir que la computación prosiga. El flujo de dicha información se especifica en esta fase de diseño, la de *comunicación.*

Para identificar las necesidades comunicacionales de la partición se deben buscar pares de tareas en los que una de ellas, el *consumidor*, requiera información de la otra, el *productor.* Para cada una de estas relaciones serán definidos los mensajes necesarios para llevar a cabo el intercambio.

Los patrones que generalmente caracterizan los requerimientos de comunicación de un

problema son categorizados en [8] de acuerdo a cuatro categorías independientes. Los formas de comunicación serán entonces: local/global, estructurada/no estructurada, estática/dinámica, y sincrónica/asincrónica.

- En la comunicación local, cada tarea se comunica con un grupo reducido de otras tareas (o "vecinos"); en constraste, una comunicación global requiere que cada tarea se comunique con muchas otras.
- En la comunicación estructurada, una tarea y sus vecinos forman una estructura regular, como un árbol o grilla; mientras que las redes de comunicación no estructuradas pueden formar grafos arbitrarios.[27]
- En la comunicación estática, la identidad de las parejas de comunicación no cambia a lo largo del tiempo; cuando en el caso de la comunicación dinámica, las estructuras pueden ser determinadas por datos calculados en tiempo de ejecución y pueden ser altamente variables.
- En la comunicación sincrónica, los productores y consumidores se ejecutan en forma coordinada, con parejas de productores/consumidores cooperando en operaciones de transferencia de datos; en contraste, la comunicación asincrónica puede requerir que un consumidor obtenga datos sin cooperación del productor.

1.5.4 Aglomeración

En las primeras dos etapas del proceso de diseño, se realizó la divisón de las tareas computacionales a realizar (*partición*) y se introdujó la comunicación para proveer los datos requeridos por cada tarea. El algoritmo resultante es aún abstracto en tanto que no se ha perfeccionado para alcanzar una ejecución eficiente en una computadora particular.

En esta tercera etapa, *aglomeración*, pasamos de lo abstracto a lo concreto. Reevaluamos decisiones tomadas en los pasos anteriores con una visión dedicada a obtener un algoritmo que ejecute en forma eficiente en una computadora paralela específica.[28]

Existen tres objetivos en conflicto entre si que guían nuestras decisiones correspondientes a la aglomeración:

1. Reducir los costos de comunicación incrementando computación y la granularidad de la comunicación. Aunque en la etapa de partición nuestro enfoque haya sido alcanzar la mayor cantidad de tareas posibles, el costo de comunicación requerido para que cada una de estas tareas se pueda ejecutar suele producir un importante impacto en la performance de un algoritmo paralelo. Por lo tanto, es necesario en este momento

[27] La comunicación no estructurada complica las tareas de aglomeración y asignación. En particular, algoritmos sofisticados pueden ser requeridos para determinar una estrategia de aglomeración que a la vez cree tareas de aproximadamente igual tamaño, y minimice los requerimientos de comunicación al generar el menor número de ejes vinculando tareas.

[28] El número de tareas resultantes de la etapa de aglomeración, aunque reducido, puede aún ser mayor que el número de procesadores. En este caso, nuestro diseño sigue siendo en cierta medida abstracto, ya que la asignación de las tareas a los procesadores no se ha resuelto.

del diseño, determinar la agrupación de tareas de modo tal que se minimice el costo comunicacional del algoritmo, ya sea por medio de la reducción en la cantidad de mensajes enviados o en el tamaño de los mismos.

2. Retener flexibilidad con respecto a la escalabilidad y decisiones de asignación. Es posible que al aglomerar tareas se limite innecesariamente la escalabilidad de un algoritmo, y que luego al portar el programa a una computadora paralela mayor la performance se vea afectada. Es por eso que es necesario preservar un balance entre la cantidad de tareas para ajustar empíricamente el algoritmo al asignar las tareas en una computadora específica.

3. Reducir los costos de ingeniería de software. Hasta ahora se ha asumido que la elección de una estrategia de aglomeración se determina únicamente por el objetivo de mejorar la eficiencia y flexibilidad de un algoritmo paralelo. Una preocupación adicional, que puede ser particularmente importante al paralelizar código secuencial existente, es el costo de desarrollo asociado con las diferentes estrategias de partición. Desde esta perspectiva, las estrategias más interesantes pueden ser aquellas que eviten cambios de código extensos.

1.5.5 Asignación (mapping)

En la cuarta y última etapa del proceso de diseño de un algoritmo paralelo, debemos especificar dónde debe ejecutar cada tarea. Nuestro objetivo al desarrollar un algoritmo de asignación es normalmente minimizar el tiempo total de ejecución[29], y usamos dos estrategias para alcanzarlo:

1. Ubicamos las tareas que pueden ser ejecutadas en forma concurrente en diferentes procesadores.

2. Ubicamos las tareas que se comunican en forma frecuente en el mismo procesador, para aumentar la localidad.

En muchos algoritmos desarrollados usando técnicas de descomposición de dominios aparecen un número fijo de tareas de igual tamaño y comunicación local y global estructurada. En dichos casos, una asignación eficiente es simple, se debe asignar cada tarea de modo de minimizar la comunicación entre procesos, y también podemos alglomerar las tareas asignadas al mismo procesador, si no se lo ha hecho anteriormente.

Además, en muchos casos es necesario utilizar técnicas de balanceo de la carga de modo de asignar las tareas, tanto al comenzar la ejecución del algoritmo paralelo como durante su ejecución, a cada procesador en la forma más pareja posible para así maximizar la performance global del algoritmo (al utilizar el mayor rendimiento de cada procesador disponible).

[29]Es necesario destacar que en nuestro caso el problema de asignación es NP-hard.

Capítulo 2

Desarrollo

2.1 Introducción

En este capítulo se presentará el desarrollo de una solución paralela para la simulación del problema de electrodeposición. Comenzaremos mostrando el modelo matemático del ECD y describiremos el algoritmo secuencial para simularlo, para luego aplicar la metodología presentada en la sección 1.5 - *Diseño de algoritmos paralelos* para elaborar la solución que será utilizada en nuestro Beowulf.

Además, serán explicadas y analizadas distintas estrategias que fueron implementadas[1] para alcanzar una mejor performance en nuestra computadora paralela, a saber:

1. Descomposición de dominios:
   (a) Descomposición en 1d
   (b) Descomposición en 2d
2. Intercambio de datos entre dominios:
   (a) Múltiples mensajes (un mensaje de sincronización de datos por cada función a resolver)
   (b) Mensaje único (empaquetado de los datos de todas las funciones con datos a sincronizar)
3. Balanceo estático de la carga (de acuerdo a la performance de cada computadora)

[1]El diseño de la solución paralela desarrollada para incorporar las estrategias de descomposición de dominios e intercambio de datos es descripto en el apéndice B.

2.2 Modelo teórico del transporte iónico

El escenario físico descripto en la sección 1.1 puede ser estudiado con un modelo macroscópico global partiendo de primeros principios, que incluye las ecuaciones de Nernst-Planck para el transporte iónico, la ecuación de Poisson para el potencial eléctrico, y las ecuaciones de Navier-Stokes para el fluido.

En la aproximación denominada modelo de vista lateral las ecuaciones devienen:

$$\frac{\partial C}{\partial t} = \frac{1}{Pe_C}\nabla^2 C + M_C[\frac{\partial}{\partial y}(C\frac{\partial \phi}{\partial y}) + \frac{\partial}{\partial z}(C\frac{\partial \phi}{\partial z})] - \frac{\partial}{\partial y}(vC) - \frac{\partial}{\partial z}(wC)$$

$$\frac{\partial A}{\partial t} = \frac{1}{Pe_A}\nabla^2 A + M_A\left[\frac{\partial}{\partial y}\left(A\frac{\partial \phi}{\partial y}\right) + \frac{\partial}{\partial z}\left(A\frac{\partial \phi}{\partial z}\right)\right] - \frac{\partial}{\partial y}(vA) - \frac{\partial}{\partial z}(wA)$$

$$\nabla^2 \phi = -Po_C C + Po_A A$$

$$\frac{\partial \omega}{\partial t} + v\frac{\partial \omega}{\partial y} + w\frac{\partial \omega}{\partial z} = \frac{1}{Re}\nabla^2 \omega - Gg_C\frac{\partial C}{\partial y} - Gg_A\frac{\partial A}{\partial y}$$

$$\nabla^2 \psi = -\omega$$

donde C, A, ψ, ϕ, y ω representan la concentración de cationes, la concentración de aniones, la función de corriente, el potencial electrostático y la vorticidad en el plano vertical, respectivamente.

Las condiciones de borde para el problema son: para (x, y) en el cátodo,

$$\phi(x, y) = -\frac{kT}{z_C e \phi_0}\ln[z_c C(x, y)]$$

$$\frac{\partial C}{\partial n}(x, y) = 0$$

donde k es la constante de Boltzmann, y T es la temperatura absoluta; para (x, y) en el ánodo,

$$\phi(x, y) = 1 - \frac{kT}{z_C e \phi_0}\ln[z_c C(x, y)]$$

$$C(x, y) = A(x, y)$$

para (x, y) en las caras laterales son $\partial C/\partial n = \partial A/\partial n = 0$; y en cada borde sólido la función de corriente satisface $\psi = \partial\psi/\partial n = 0$.

Los números adimensionales utilizados son:

el número de migración,

$$M_C = \frac{\mu_C \phi_0}{x_0 u_0},\ M_A = \frac{\mu_A \phi_0}{x_0 u_0};$$

el número de Peclet,

$$Pe_C = \frac{x_o u_o}{D_C},\ Pe_A = \frac{x_o u_o}{D_A};$$

el número de Poisson,

$$Po_C = \frac{x_o^2 C_o z_C e}{\epsilon \phi_0},\ Po_A = \frac{x_o^2 C_o z_A e}{\epsilon \phi_0};$$

el número de Reynolds,

$$Re = \frac{x_o u_o}{v};$$

el número de Grashof gravitatorio,

$$Gg_C = \frac{x_o C_o g \alpha}{u_0^2},\ Gg_A = \frac{x_o C_o g \beta}{u_0^2};$$

2.2.1 Modelo computacional

El modelo computacional resuelve el sistema, en cada paso de tiempo, en un dominio fijo, en una malla 2d uniforme (equiespaciada en ambas direcciones) usando diferencias finitas y métodos iterativos standard (ver [16]). Su solución se obtiene mediante el sistema de ecuaciones en diferencias:

$$W_k^{n+1} = \sum_j a_j W_j^n$$

donde j representa el vecino más cercano de la ubicación k, la sumatoria se aplica sobre el rango de todos los vecinos más cercanos, W_k es una función vectorial, cuyos componentes son las concentraciones C y A, el potencial eletrostático ϕ, la función de vorticidad ω, y la función de corriente ψ, y a_j es la matriz diagonal cuyos elementos contienen coeficientes no-lineales de las ecuaciones discretizadas.

Por brevedad, presentamos a continuación la aproximación unidimensional de las ecuaciones de Nernst-Planck y el campo eléctrico. Las ecuaciones de Navier-Stokes para el fluido tienen un tratamiento similar y pueden ser consultadas en [18]. Un esquema implícito conservativo del sistema resulta en:

$$\frac{C_i^{n+1} - C_i^n}{k} = Pe_C \left(\frac{C_{i+1}^{n+1} + C_{i-1}^{n+1} - 2C_i^{n+1}}{h^2} \right) + M_C \left(\frac{C_{i+1/2}^{n+1} \left(\phi_{i+1}^{n+1} - \phi_i^{n+1} \right) + C_{i-1/2}^{n+1} \left(\phi_i^{n+1} - \phi_{i-1}^{n+1} \right)}{h^2} \right)$$

$$\frac{A_i^{n+1} - A_i^n}{k} = Pe_A \left(\frac{A_{i+1}^{n+1} + A_{i-1}^{n+1} - 2A_i^{n+1}}{h^2} \right) + M_A \left(\frac{A_{i+1/2}^{n+1} \left(\phi_{i+1}^{n+1} - \phi_i^{n+1} \right) + A_{i-1/2}^{n+1} \left(\phi_i^{n+1} - \phi_{i-1}^{n+1} \right)}{h^2} \right)$$

$$\phi_i^{n+1} = \frac{\phi_{i+1}^{n+1} + \phi_{i-1}^{n+1} - h^2 f^*}{2}$$

donde $f^* = -Po_C C_i^{n+1} + Po_A A_i^{n+1}$, y n es el paso de tiempo. Los valores de concentración en nodos con subíndice fraccional son computados como el promedio de las concentraciones vecinas en los nodos enteros. Las condiciones de borde se aproximan de manera similar.

En el método iterativo para resolver el sistema algebraico, se alcanza la convergencia en cada paso de tiempo cuando el residuo satisface la siguiente condición:

$$res = \max_i \left[w_i^{n+1,l+1} - w_i^{n+1,l} \right] < 10^{-5};\ w = A, C, \phi$$

donde l es el número de iteración. La convergencia del proceso evolucionario hacia el estado estacional se alcanza si:

$$res^* = \max_i \left[w_i^{n+1,l+1} - w_i^{n,l} \right] < 10^{-7};\ w = A, C, \phi$$

2.2.2 Simulación numérica secuencial

Con anterioridad a este trabajo, en [19] se implementó una simulación numérica en forma secuencial. Como los métodos numéricos empleados no son estables utilizando los parámetros físicos correspondientes al problema, el esquema requiere la utilización de mallas mucho más finas que las que se pueden procesar en sólo una máquina. Para enfrentar este problema, en la solución secuencial se trabajó con simulaciones en una escala diferente a la de los experimentos y, por lo tanto, el análisis y comparación de los resultados fue más restringida.

En una visión simplificada del algoritmo secuencial, este se reduce a la ejecución de los siguientes pasos:

1. Inicialización de los arreglos con las condiciones iniciales.
2. **Mientras** no se alcance el estado estacionario ó la cantidad de pasos de tiempo especificados por el usuario.
    2.1. **Mientras** no se alcance la convergencia en un paso de tiempo.
        2.1.1. Calcular la nueva aproximación.
        2.1.2. Actualizar el residuo que mide la convergencia.

2.3 Diseño del algoritmo paralelo

Para realizar la implementación de la simulación numérica del proceso físico en una máquina paralela utilizamos la metodología descripta anteriormente (en la sección 1.5 - *Diseño de algoritmos paralelos*). En la primera etapa, de *partición* del problema, nos inclinamos por utilizar la estrategia de *paralelismo de datos*, para descomponer el problema en subdominios. En cada subdominio habrá cinco funciones incógnita definidas sobre los nodos de su correspondiente malla uniforme.

En el problema secuencial, la aproximación por diferencias finitas del problema diferencial sobre una malla uniforme, resulta en un sistema algebraico de $5mn$ incógnitas que se resuelve en forma iterativa. Por lo tanto, en cada iteración (paso 2.1.1 del algoritmo secuencial) se realizan $5mn$ tareas. Una tarea adicional corresponde al paso 2.1.2 del algoritmo secuencial, donde se debe calcular el residuo que mide la convergencia de la nueva solución.

En la segunda etapa, determinamos los requerimientos de comunicación de cada una de las tareas. Para calcular la nueva solución en cada nodo por diferencias finitas se requieren los valores de la solución previa en los nodos adyacentes, como se indica en la Fig. 2.1.

Además, como el sistema de ecuaciones liga las distintas funciones incógnita entre sí, también será necesario que los datos correspondientes a las mismas posiciones de una arreglo sean accesibles en otra. En el diagrama de la Fig. 2.2 se muestran las dependencias entre las distintas funciones para su resolución.

Podemos describir el tipo de comunicación requerido para realizar las tareas de cálculo en cada paso del algoritmo de acuerdo con las categorías presentadas en la sección 1.5.3 - *Diseño de algoritmos paralelos - Comunicación.* La comunicación es:

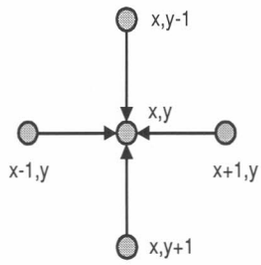


Figura 2.1: Los valores en nodos adyacentes a x, y son requeridos para poder calcular el nuevo valor de la solución.

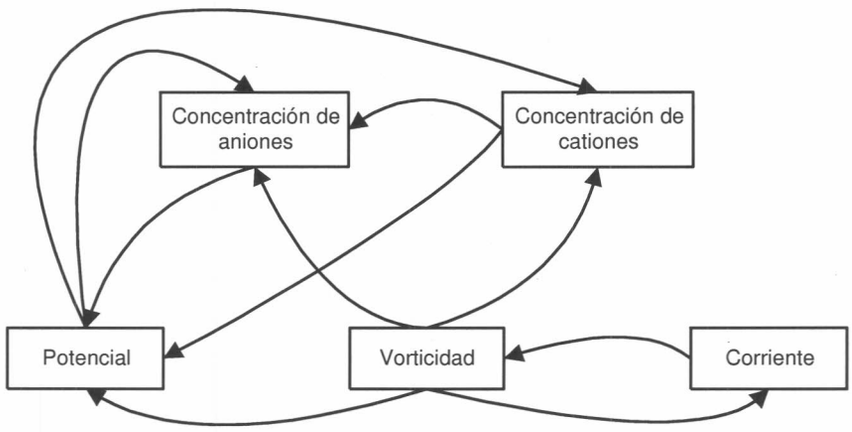


Figura 2.2: Dependencias entre las funciones incógnita que componen el problema para obtener la nueva solución en la iteración siguiente.

1. *local* ya que para realizar una de las tareas sólo es necesario acceder a las tareas adyacentes[2],

2. *estructurada* porque las tareas de nuestra partición corresponden a arreglos bidimensionales (cuya estructura es regular),

3. *estática* porque durante la ejecución del algoritmo no variarán las dependencias de una tarea, y

4. *sincrónica* dado que para realizar el siguiente paso de cálculo cada tarea requiere los

[2]Aunque en el caso de la tarea que actualiza el residuo de convergencia se requiere comunicación *global*, ya que este valor depende de la máxima diferencia entre dos iteraciones sucesivas de cada elemento de los arreglos.

datos del paso anterior de los procesadores adyacentes.

Dado que hemos determinado el intercambio de información necesario para poder ejecutar las tareas, debemos ahora decidir como agruparlas para que, en el próximo y último paso del método de diseño, podamos realizar la asignación a los procesadores disponibles.

En un principio podemos descartar la opción de agrupar todas las tareas correspondientes a cada función incógnita en forma separada porque, contrariamente a nuestros objetivos para esta etapa, se restringen definitivamente las posibilidades de asignación y, por lo tanto, la flexibilidad del diseño, dado que tan sólo dispondremos en este caso de cinco grupos para asignar en la etapa final (es decir, se limita la posibilidad de agregar nuevos procesadores para mejorar la performance global). Además, aún cuando la cantidad de procesadores disponibles fuera inferior a cinco, las dependencias de cada función para poder evaluar cada una de las tareas, haría necesaria la transmisión (en cada iteración) de todos los datos de los arreglos hacia otro, con el consiguiente impacto en la performance global del programa.

El enfoque utilizado en nuestro trabajo consiste en agrupar las tareas correspondientes a una misma posición en los arreglos correspondientes a cada función del problema. De este modo, el grupo de tareas de la posición (i, j) estará compuesto por las tareas, en la misma posición, de las funciones de: potencial, concentración de aniones, concentración de cationes, corriente y vorticidad.

Por lo tanto, cumpliendo los primeros dos objetivos de la etapa de aglomeración, por un lado disponemos de un grupo de tareas que en caso de no ser asignadas a un mismo procesador harían necesario el intercambio de datos entre dichos procesadores para poder ser ejecutadas, con lo que reducimos los requerimientos de comunicación. Mientras que por otro, aunque nuestro objetivo será ubicar los grupos de tareas correspondientes a posiciones adyacentes en un mismo procesador, mantenemos flexibilidad para que en la etapa de asignación podamos decidir en qué forma, para que la mejor de un conjunto de estrategias nos brinde una mayor performance.

Finalmente, en la etapa de asignación debemos decidir en qué procesador se ejecutará cada grupo de tareas. Buscamos aprovechar el máximo rendimiento de los procesadores disponibles, y al mismo tiempo queremos mantener el mayor grado de localidad en los grupos de tareas, para evitar costos de comunicación entre los procesadores que limiten el *speedup* del sistema.

En el desarrollo de nuestro trabajo implementamos múltiples variantes de asignación que, en casi todos los casos, fueron combinadas para luego determinar cuál es la que permite obtener mejor performance en la resolución del problema. En las próximas secciones, luego de presentar una versión simplificada del algoritmo paralelo, estudiaremos cada una de las estrategias utilizadas.

2.3.1 Simulación numérica paralela

Los pasos afectados al realizar la paralelización del método son 1, 2.1.1 y 2.1.2. En el paso 1, la inicialización de los arreglos locales a cada procesador debe tener en cuenta el

desplazamiento correspondiente a la ubicación global de su subarreglo para poder inicializarla con los valores correctos. En el 2.1.1, además, para poder llevar a cabo el cálculo en forma correcta en el arreglo global, se deben sincronizar los datos entre los distintos dominios asignados por medio de las regiones de intercambio de datos. Finalmente, en 2.1.2, para actualizar el residuo de convergencia se debe determinar el máximo global de cada uno de los arreglos, a tal efecto, cada uno de los procesadores debe calcular el máximo correspondiente a su área local para luego poder llevar a cabo una reducción global que permita establecer los residuos globales y distribuirlos a todos los procesadores (por medio de las funciones presentadas en la sección 1.4.3 - *Message Passing - Transmisión grupal*).

De esta forma, el método de cálculo en la máquina paralela realiza los siguientes pasos:

1. Inicialización de los arreglos con las condiciones iniciales, *teniendo en cuenta la ubicación global de cada uno de los procesadores.*
2. **Mientras** no se alcance el estado estacionario ó la cantidad de pasos de tiempo especificados por el usuario.
   - 2.1. **Mientras** no se alcance la convergencia en un paso de tiempo.
     - 2.1.1. Calcular la nueva aproximación, *teniendo en cuenta la ubicación global.*
     - *2.1.2. Realizar la sincronización de los datos de cada arreglo.*
     - *2.1.3. Determinar residuos que miden localmente la convergencia.*
     - *2.1.4. Realizar reducciones globales para obtener los residuos de convergencia a nivel global.*
     - 2.1.5. Actualizar el residuo que mide la convergencia.

2.4 Descomposición de dominios

La técnica de descomposición de dominios se ha aplicado en la solución de una gran variedad de problemas en computadoras paralelas, como en la solución de problemas basados en la resolución de PDEs ([2], [3]) y el procesamiento digital de imágenes ([20], [15]).

Podemos determinar dos clases de problemas sobre los que se han realizado diversos trabajos sobre descomposición de dominios, de acuerdo a la presencia o no de estructura en el problema a resolver. En el caso de problemas que carecen de una estructura regular (o *no estructurados*), el enfoque más utilizado consiste en la aplicación de la técnica de *recursive bisection* ([26], [5], [30]). Por otra parte, para los problemas con estructura y, en particular, con dominios rectangulares, [4] y [31] han estudiado el problema de partición óptima para su asignación en computadoras paralelas, conocido como el problema de perímetro mínimo (NP-hard) aplicando algoritmos genéticos.

En nuestro problema, el dominio a descomponer corresponde al conjunto de los arreglos asociados a cada una de las funciones que serán resueltas por diferencias finitas. Debido a la limitada cantidad de procesadores en nuestro cluster, la implementación de nuestra solución asigna únicamente dominios rectangulares. Utilizar un método de descomposición

con dominios irregulares, aumentaría la complejidad de los mecanismos de sincronización de datos y, a la vez, al generar una mayor cantidad de segmentos que limitan un par de procesadores, también incrementaría la cantidad de mensajes a enviar entre los procesadores. Como se verá en discusiones posteriores, el mayor tiempo requerido para enviar más mensajes de menor tamaño compensaría, al menos en parte, las ventajas de una descomposición más cercana a la óptima (es decir, que minimiza la cantidad de datos transferidos).

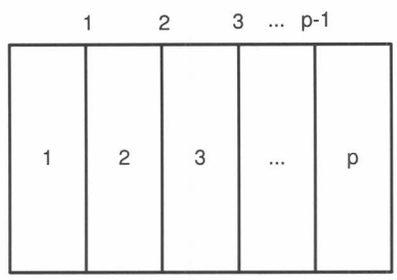


Figura 2.3: Descomposición en 1d (se observan las $p-1$ columnas utilizadas para asignar un dominio a cada uno de los p procesadores disponibles).

Las estrategias de descomposición que se estudiaron corresponden a la división en 1d y 2d del área de cada arreglo. En la primera de ellas (ver fig. 2.3), se procede a definir dominios rectangulares a partir de la selección de $p-1$ columnas[3], quedando limitada el área correspondiente a una de las máquinas del cluster por dicha columna y aquella con la que se realizó la asignación al procesador anterior (en el caso del primer y último procesador, los límites están dados por la primera y última columna del arreglo bidimensional, respectivamente).

En la división en 2d (ver fig. 2.4) se divide, además, a los arreglos bidimensionales por medio de cortes por filas. Dado que nuestra implementación sólo manejará dominios rectangulares, una descomposición en 2d en x,y (columnas y filas de dominios resultantes, respectivamente) debe obedecer la restricción $p = xy$.

La descomposición en 2d introduce un nuevo costo a tener en cuenta al establecer cuál es la más eficiente de las estrategias, que consiste en tener que copiar a buffers intermedios los datos ubicados sobre las filas utilizadas para delimitar los distintos dominios, debido a que dichos datos no se encuentran ubicados secuencialmente en la representación en memoria de los arreglos (por lo tanto, se requiere espacio en memoria adicional para $2m(x-1)$ datos). Aún cuando el tiempo de copiado de estos datos no sea significativo, ya que la transmisión de los datos ulterior de estos datos será relizada en un medio al menos un orden de magnitud más lento que la memoria principal de cada procesador, no se puede dejar de considerar el costo de reservar espacio en memoria para implementar esta estrategia.

[3]Puede también realizarse por filas, pero la elección no es indistinta ya que la organización de los datos en memoria afecta a la performance de la transmisión. Es decir, si los datos están organizados secuencialmente por columnas, no es necesario copiarlos a un buffer intermedio al enviarlos a otro procesador.

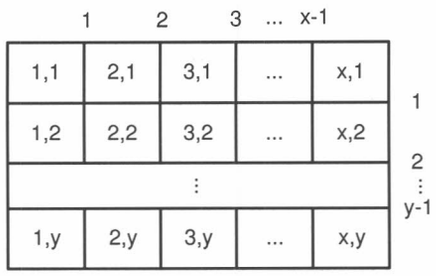


Figura 2.4: Descomposición en 2d (se observan las $x-1$ columnas y las $y-1$ filas utilizadas para asignar un dominio a cada uno de los p procesadores disponibles).

2.4.1 Evaluación de las técnicas

Analizamos ahora la cantidad de datos que se deben transmitir en cada iteración con cada una de las estrategias. Si disponemos de p procesadores a los que debemos asignar áreas de un arreglo bidimensional de m columnas por n filas, en el caso de la descomposición en 1d en cada una de las columnas que limitan las áreas entre procesadores adyacentes se enviarán n celdas de datos hacia el otro procesador (o $2n$ celdas de datos por iteración, n hacia cada lado). Por lo tanto, como en la descomposición se toman $p-1$ columnas, la cantidad total de celdas de datos a transmitir es $2n(p-1)$.

Por su parte, para una descomposición en 2d (con x e y como se los definió anteriormente), la cantidad de celdas a transmitir en cada iteración es $2m(y-1)+2n(x-1)$. Es decir, se deben transmitir las $y-1$ filas y las $x-1$ columnas de datos utilizadas en la asignación de dominios.

Para poder determinar que esquema de descomposición es conveniente para cada problema podemos ver cuando:

$$
\begin{gathered}
2n(p-1) < 2m(y-1) + 2n(x-1) \text{ con } p = xy \\
n(p-1) < m(y-1) + n(x-1) \\
n(p-1) - n(x-1) < m(y-1) \\
n(p-x) < m(y-1) \\
n(p-x) < m(\tfrac{p}{x} - 1) = m\tfrac{(p-x)}{x} \\
n < \tfrac{m}{x}
\end{gathered}
$$

Entonces, la cantidad de filas debe ser menor que la cantidad de columnas divididas por la cantidad de columnas de dominios a asignar.

En la fig. 2.5, se muestra la cantidad de datos a transmitir en diversas configuraciones con 16 procesadores, a medida que se aumentan la cantidad de columnas, con las filas fijas en 30. En el caso de la descomposición unidimensional, como la cantidad de filas no cambia en cada

configuración, los requerimientos de comunicación se mantienen constantes. En el resto de los casos, correspondientes a descomposiciones bidimensionales, el tiempo de comunicación se verá incrementado, con una mayor pendiente a medida que se agregan columnas. Sin embargo, debe apreciarse que en muchas de las configuraciones evaluadas, mientras no se cumple la relación antes expuesta, se puede obtener una menor transferencia de datos entre los procesadores por medio de una descomposición bidimensional.

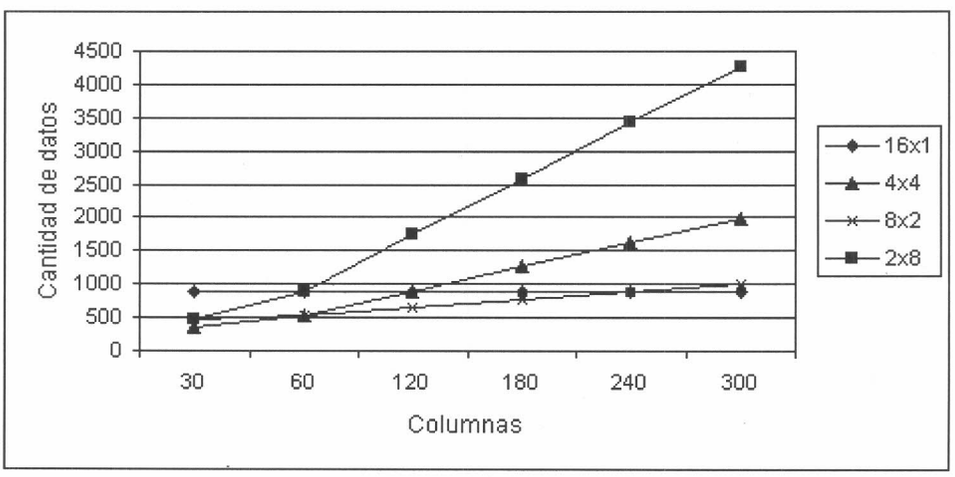


Figura 2.5: Cantidad de datos a transmitir en descomposiciones en 1d y 2d con 16 procesadores.

Como se menciona en la sección anterior, debemos además considerar para cada una de las estrategias utilizadas sus requerimientos de memoria para buffers intermedios. Mientras que la descomposición unidimensional no requiere espacio adicional para la transmisión/recepción de datos, la estrategia bidimensional requiere buffers para almacenar los datos de cada una de las filas utilizadas para delimitar los dominios.

En la siguiente tabla se muestran la cantidad de datos que deberán ser copiados a buffers intermedios en las descomposiciones analizadas en el ejemplo anterior. Se observa que a medida que se utilizan más filas para determinar los dominios de la descomposición, aumentan también los requerimientos de memoria.

Columnas	**16x1**	**8x2**	**4x4**	**2x8**
30	0	60	90	240
60	0	120	180	480
120	0	240	360	960
180	0	360	540	1440
240	0	480	720	1920
300	0	600	900	2400

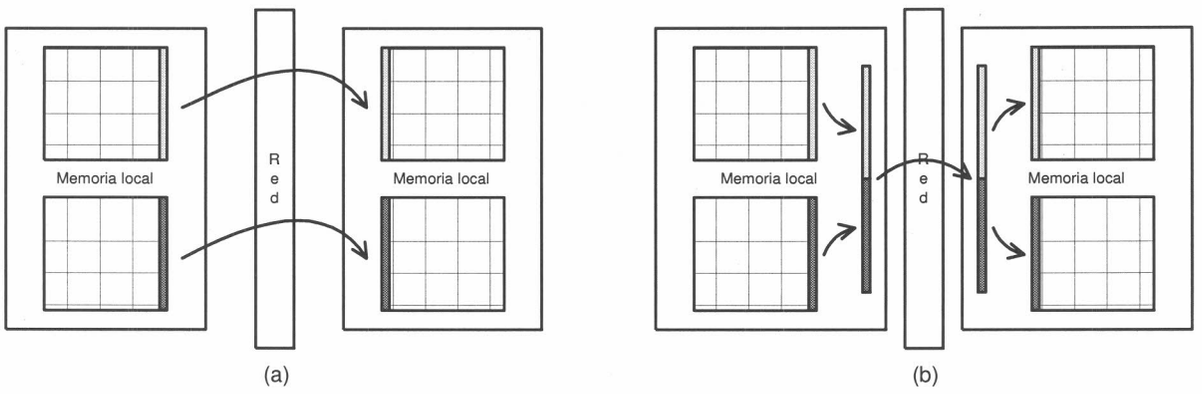


Figura 2.6: Estrategias de intercambio de datos. (a) Múltiples mensajes, (b) Mensaje único (con empaquetado en buffer intermedio).

2.5 Intercambio de datos entre los dominios

En cada una de las iteraciones en que se realizan diferencias finitas para resolver cada una de las funciones que componen el problema se modifican las celdas de los dominios asignados de acuerdo a las estrategias descriptas en la sección anterior. En ambos casos, las áreas adyacentes a otros procesadores deben intercambiar datos para proseguir con la siguiente iteración. Para cada arreglo, entonces, cada uno de los procesadores debe enviar datos correspondientes a columnas o, en el caso de la descomposición en 2d, partes de columnas y de filas que limitan con otro procesador.

Como en nuestro problema debemos sincronizar datos correspondientes a más de un arreglo bidimensional, en cada iteración es necesario mandar múltiples mensajes a cada procesador adyacente. Anteriormente se ha mostrado (en la sección 1.4.1 - *El costo de comunicación*) que el costo de enviar un mensaje a otro procesador tiene un componente fijo correspondiente a la inicialización de la transmisión de los datos (*latencia*) y otro que depende la cantidad de datos a enviar. Entonces, como por cada mensaje enviado se incurre en el costo de inicialización, se decidió implementar y evaluar la copia en un buffer intermedio de todos los datos a transmitir a cada procesador en una interación, para luego enviar sólo un mensaje por procesador adyacente.

Este nuevo paso previo a la transmisión, que puede también considerarse como parte de la fase de inicialización de la transmisión de un mensaje, incurre en un costo que no se presenta en el esquema original, y es por dicha razón que no podemos establecer a priori cuál de las dos estrategias otorga el mejor rendimiento en nuestra simulación.[4]

Quedan entonces definidas dos estrategias de intercambio de datos con cada procesador, que se complementan con las descomposiciones presentadas en la sección anterior:

[4]La estrategia de descomposición en 2d, como se ha explicado, ya incurre en costo similar para la sincronización de las partes de filas adyacentes a otro procesador.

1. Múltiples mensajes por procesador adyacente.

2. Único mensaje por procesador adyacente.

La implementación de la segunda de las estrategias se realizó utilizando los tipos derivados de datos provistos por MPI (que se presentaron en la sección 1.4.2 - *Message passing - Empaquetado*). Como el tamaño de los mensajes y las áreas de datos usadas permanecen invariables durante la ejecución del programa, las funciones de empaquetado de datos no son prácticas ya que en cada mensaje se hace necesario reconstruir el formato del mensaje, debiendo solicitar cada vez un área de datos donde copiar los datos en forma intermedia. Con los tipos derivados de datos, realizamos este trabajo únicamente en la inicialización del programa (antes de comenzar a realizar las iteraciones de cálculo) y, al enviar un nuevo mensaje, sólo debemos copiar los datos al área de datos intermedio.

2.6 Balanceo estático de la carga de los procesadores en un Beowulf heterogéneo

Al enumerar los beneficios de armar un Beowulf para utilizar en la simulación de problemas complejos destacamos la posibilidad de expandirlo con facilidad de acuerdo a nuestras necesidades y posibilidades, debido a que estaba integrado por componentes de computadoras personales standard. Durante el transcurso de este trabajo nuestro cluster fue modificado en varias oportunidades, incorporando nuevas computadoras, distintas a las que lo integraban previamente (la evolución de la configuración de nuestro Beowulf se presenta en la sección A.1.1).

Las diferencias por arquitectura de procesador y en velocidad de reloj hacen que el rendimiento de las computadoras del cluster sea dispar. Entonces es necesario incorporar a la etapa de asignación de dominios, una lógica de distribución o *balanceo* de la carga para que cada uno trabaje sobre una cantidad de datos proporcional a su capacidad de cálculo.

El esquema de balanceo de carga sólo fue implementado para combinarlo con las estrategias de descomposición unidimensional. En el caso de la descomposición bidimensional, descomponer los dominios de cálculo teniendo en cuenta la capacidad de cada procesador requiere la utilización de una descomposición con áreas no rectangulares o en caso de realizar una implementación más simple, necesita transmitir datos a más de un procesador en una misma dirección (por ejemplo, puede existir más de un procesador en contacto con el borde inferior del dominio asignado). Se consideró, entonces, que este nivel de complejidad excedía los beneficios obtenibles porque, como ya hemos visto al comparar la cantidad de datos a transmitir en cada caso, la descomposición bidimensional ya incurre en un mayor costo debido a la relación entre la cantidad de filas y columnas del problema a resolver.

Para medir la performance real de cada una de las distintas computadoras que integran el cluster, realizamos un conjunto de benchmarks destinados a obtener un indicador numérico de su rendimiento al ejecutar operaciones de cálculo de punto flotante (que comprenden la

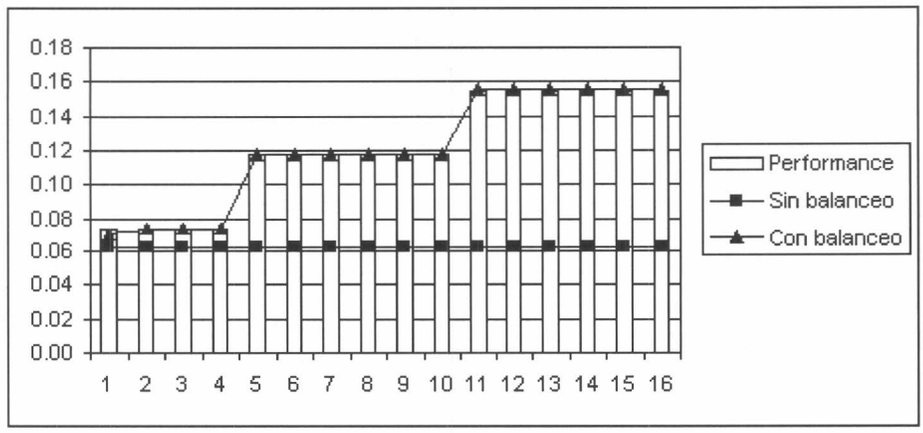


Figura 2.7: Asignación de dominios por procesador en las estrategias sin y con balanceo.

mayor parte de las operaciones realizadas por la simulación en cada una de las iteraciones) para asignarle un dominio de datos de la misma proporción. En la siguiente tabla se presentan los resultados, a partir del valor de referencia 100, asignado a la computadora de menor performance.

Computadora	**Performance**
Intel Pentium III 733mhz	100
AMD Athlon 950mhz	160
AMD Athlon 1,2ghz	210

En la fig. 2.7 se puede observar junto con el rendimiento de cada uno de las procesadores, la proporción del problema asignada a cada uno de ellos de acuerdo con la estrategia original, que asume implícitamente la existencia de una computadora paralela compuesta por procesadores homogéneos, y con la que realiza una distribución de la carga en forma balanceada.

En el primero de los casos, la distribución muestra que los primeros procesadores (los de menor performance) deberán realizar más trabajo que aquel que habría de corresponderles en forma proporcional, mientras que el caso contrario ocurre con los procesadores más veloces de nuestro Beowulf.

La consecuencia más importante en cuanto al desempeño que puede obtenerse en el cluster se desprende de observar que como todos los procesadores deberán realizar una misma cantidad de operaciones de cálculo en cada iteración, evidentemente los procesadores más lentos requerirán más tiempo. Luego, como cada iteración se lanza luego de sincronizar los datos con los procesadores adyacentes, los más rápidos deberán esperar, sin poder ejecutar ninguna tarea, a que los más lentos transmitan sus resultados. El tiempo ocioso de los procesadores más veloces, disminuye la eficiencia global de la computadora paralela, por la pérdida de valiosos ciclos de cálculo.

Al distribuir la carga se puede esperar que este problema sea eliminado porque los procesadores habrán de terminar sus iteraciones de cálculo casi al mismo tiempo.

2.7 Análisis de complejidad

2.7.1 Introducción

En la sección 1.3 - *Performance*, ya hemos explicado que el análisis de complejidad nos permitirá evaluar el desempeño de nuestro programa paralelo en relación a su versión secuencial. En el desarrollo, además, nos brinda formas objetivas de comparar las distintas estrategias de asignación que hemos implementado. Finalmente, un conocimiento preciso del comportamiento de nuestro programa paralelo nos dejará analizar la posibilidad de incorporar nuevas computadoras a nuestro Beowulf, estimando el rendimiento a obtenerse con una nueva configuración.

2.7.2 Algoritmo secuencial

El tiempo de ejecución de la simulación numérica del problema físico depende de la convergencia del proceso iterativo, que está determinado, además del tamaño de la grilla utilizada, de los valores correspondientes a las constantes utilizadas en las ecuaciones. Como la convergencia de la solución paralela del mismo método de cálculo que utiliza la versión secuencial no padece variaciones significativas, el estudio de la complejidad del algoritmo será realizado sobre una iteración de cálculo (pero a igual densidad de malla).

Comenzando con la implementación secuencial de la simulación numérica, el tiempo requerido por una iteración quedará determinado sólo por el tiempo de cálculo (ya que no realiza comunicación con otro proceso), y es $T_1(m,n) = m \cdot n \cdot k_{calc}$. No es necesario realizar un estudio más detallado de las operaciones de cálculo realizadas para resolver cada una de las ecuaciones que componen el problema, porque serán las mismas las realizadas por la versión paralela. A su vez, esta simplificación nos permitirá concentrarnos en los aspectos que gobiernan el estudio de escalabilidad de un programa paralelo.

2.7.3 Algoritmo paralelo

La complejidad computacional de cada iteración correspondiente a la solución paralela desarrollada en este trabajo puede definirse de acuerdo a las etapas que debe cumplir el algoritmo (cálculo, sincronización entre procesos, y reducción global para actualizar el residuo de convergencia):

$$T_p(m,n,p) = T_{calc}(m,n,p) + T_{sinc}(m,n,p) + T_{red}(m,n,p)$$

Descartando el tiempo de inicialización agregado únicamente para la solución paralela, podemos asumir que el tiempo de cálculo al utilizar p procesadores deberá disminuir proporcionalmente, es decir:

$$T_p(m,n,p) = \frac{T_1(m,n)}{p} + T_{sinc}(m,n,p) + T_{red}(m,n,p) = \\ = \frac{m \cdot n \cdot k_{calc}}{p} + T_{sinc}(m,n,p) + T_{red}(m,n,p)$$

En este último paso, se asume implícitamente que la computadora paralela dispone de un ambiente de procesadores homogéneos. Para el caso en que necesitemos evaluar la performance de nuestro Beowulf, integrado por computadoras de distinta performance, el valor de k_{calc} deberá tomarse como el promedio de dicho valor para cada una de las computadoras que integran el cluster. De este modo, estaremos evaluando el comportamiento de un *virtual* cluster homogéneo.

El tiempo correspondiente a la reducción para determinar el residuo de convergencia global en cada iteración permanece sin cambios al variar las estrategias de sincronización de datos entre los procesos. De acuerdo a la fórmula presentada para la operación `MPI_Allreduce` en la sección 1.4.3 - *Performance de operaciones globales en MPI*, el tiempo requerido por la reducción será:

$$T_{red}(m,n,p) = \log p \cdot k_{red}$$

Asumimos que k_{red} será un valor constante debido a que en las distintas configuraciones a evaluar, tanto la operación de reducción global a utilizar y la cantidad de datos sobre los que se realizará la operación es la misma. Por lo tanto, sólo nos interesa conocer el comportamiento en la medida que variamos la cantidad de procesadores disponibles.

Finalmente, consideramos el tiempo de sincronización para cada una de las estrategias implementadas. Debe notarse que en el análisis de complejidad para medir la performance del algortimo paralelo, debemos estudiar los requerimientos de transferencia de datos para alguno de los procesadores internos de la descomposición. Es decir, para uno de los procesadores que debe en cada iteración transferir la mayor cantidad de datos con los procesadores adyacentes, ya que estos serán los que limiten la performance global de la solución ya que los procesadores (aún aquellos que deban transferir menos información, como los que están ubicados en los bordes) realizan las iteraciones de cálculo de manera sincrónica (tienen que esperar los datos del procesador adyacente para pasar a la siguiente iteración).

Como cada una de las estrategias de descomposición de dominios se combinarán con los esquemas de intercambio de datos, obtenemos fórmulas distintas para cada variante. En el primer caso, correspondiente a la descomposición unidimensional el tiempo de sincronización será:

$$\begin{array}{ll} \text{MMxP} & T_{sinc(d1,mmp)}(m,n,p) = 2k_{mat}(t_s + t_c n) = 2 \cdot 5(t_s + t_c n) \\ \text{1MxP} & T_{sinc(d1,1mp)}(m,n,p) = 2(t_s + k_{mat} t_c n) = 2t_s + 2 \cdot 5 t_c n \end{array}$$

Debe notarse que la performance correspondiente a ambas variantes no depende de la cantidad de procesadores utilizados, sino solamente de la dimensión del problema a resolver (o, de manera específica, de la cantidad de celdas de datos que se transmiten al sincronizar las columnas que limitan los dominios).

Como se desprende de las fórmulas presentadas, un caso particular en el que la descomposición unidimensional aventaja a la bidimensional ocurre cuando es necesario realizar corridas incrementando sólo la cantidad de columnas del problema a resolver. En ese caso, se mantendrá constante la cantidad de datos a transmitir (ya que la cantidad de filas, n, no varía) y, por tanto, también el tiempo de sincronización entre procesos.

Mientras que las dos variantes en la implementación de la descomposición bidimensional serán:

$$\begin{array}{ll} \text{MMxP} & T_{sinc(d2,mmp)}(m,n,p) = 2k_{mat}(2t_s + t_c\frac{m}{x} + t_c\frac{n}{y}) = 2 \cdot 5(2t_s + t_c\frac{m}{x} + t_c\frac{n}{y}) \\ \text{1MxP} & T_{sinc(d2,1mp)}(m,n,p) = 2(2t_s + k_{mat}(t_c\frac{m}{x} + t_c\frac{n}{y})) = 4t_s + 2 \cdot 5(t_c\frac{m}{x} + t_c\frac{n}{y})) \end{array}$$

2.7.4 Overhead

Al introducir el concepto de *overhead* de un programa paralelo respecto de su versión secuencial definimos tres categorías principales que suelen incrementar el trabajo de la versión paralela. En nuestro caso, el *tiempo de comunicación para la transmisión de mensajes* corresponde al tiempo de sincronización entre procesos (T_{sinc}); mientras que los *cálculos adicionales requeridos en la versión paralela respecto de la secuencial* son requeridos por la reducción global que determina los residuos de convergencia en cada iteración de cálculo (T_{red}).

Luego, aplicando a nuestra implementación la definición de *overhead* de una solución paralela, $T_O(m,n,p) = pT_p(m,n,p) - T_1(m,n)$, obtenemos:

$$\begin{gathered} T_O(m,n,p) = pT_p(m,n,p) - T_1(m,n) = \\ = p(T_{calc}(m,n,p) + T_{sinc}(m,n,p) + T_{red}(m,n,p)) - T_1(m,n) = \\ = p(\tfrac{T_1(m,n)}{p}) + p(T_{sinc}(m,n,p) + T_{red}(m,n,p)) - T_1(m,n) \\ = p(T_{sinc}(m,n,p) + T_{red}(m,n,p)) \end{gathered}$$

Por lo tanto, el *overhead* de nuestro algoritmo paralelo depende de la cantidad de procesadores utilizados y de la suma del tiempo de sincronización y reducción requerido. Al comparar, entonces, el trabajo adicional de cada una de las variantes implementadas, sólo tendremos que analizar la complejidad de la estrategia de sincronización utilizada, ya que el resto de los términos permanece inalterado en cada caso.

Para determinar la escalabilidad de nuestra solución ante nuevas configuraciones del cluster (resultantes de la incorporación de nuevas computadoras), utilizaremos el análisis de complejidad del *overhead* en la fórmula para calcular la eficiencia de un algoritmo presentada en 1.3.6.

Capítulo 3

Resultados

3.1 Introducción

En este capítulo se discuten los resultados obtenidos en este trabajo de la simulación numérica de la deposición electroquímica (ECD) con la implementación paralela descripta anteriormente.

En primer lugar observaremos las simulaciones del modelo del plano vertical, correspondiente a la vista lateral de la celda en el experimento físico, modelo que describe el caso limitante dominado por la gravitoconvección. Estos resultados fueron presentados anteriormente en [17].

Luego estudiaremos el desempeño del algoritmo paralelo elaborado, analizando cada una de las estrategias implementadas para la descomposición y asignación de dominios. Con este estudio encontraremos la configuración que nos brinda una mejor performance (y por ende, un mayor aprovechamiento de los recursos disponibles) en la simulación numérica.

Finalmente, veremos la eficiencia y escalabilidad de la configuración encontrada, de acuerdo al modelo elaborado en el análisis de complejidad de nuestro algoritmo.

3.2 Simulación numérica

A continuación se presentan los resultados de la simulación numérica paralela del modelo de la vista lateral. Para tener una idea clara del problema, la fig. 3.1 muestra, en un ensayo experimental de ECD, una vista lateral de la evolución espacio-temporal de los rollos catódicos y anódicos. Esta vista fue tomada por superposición temporal de trayectorias de partículas. Allí se ve claramente como los rollos catódicos y anódicos invaden la celda desde ambos extremos.

Las celdas experimentales tienen en promedio una relación entre longitud-altura de 30 que se mantienen en la simulación con una grilla de 1080x36. Las simulaciones describen un proceso evolucionario sin crecimiento hasta que el estado fijo es obtenido (en algunos casos). Los números adimensionales utilizados son (sin agregar glicerina): $Pe = 12$, $Mc = 1,0$,

$Ma = 1,52$, $Po = 600$, $Re = 1,2 \cdot 10^{-2}$ y $Gr_A = Gr_C = 1,0 \cdot 10^4$.

La fig. 3.2 presenta el modelo correspondiente a la vista lateral en la simulación de la evolución del vórtice (que muestra el mismo comportamiento que la secuencia de imágenes obtenidas experimentalmente, en la fig. 3.1), visualizado con curvas de nivel de la función de corriente, para el caso en que no se agregó glicerina (viscosidad de agua pura). Se observa en forma clara la existencia y evolución en espacio-tiempo de los vórtices catódicos y anódicos. La región generada por la convección debida a las fuerzas de empuje crece con el tiempo, invadiendo la celda desde ambos extremos. Los vórtices continúan creciendo hasta chocar. Esto modifica significativamente el patrón del flujo y la concentración iónica. Entre los vórtices no hay movimiento previamente a la colisión, luego el fluido es inmerso en un único vórtice.

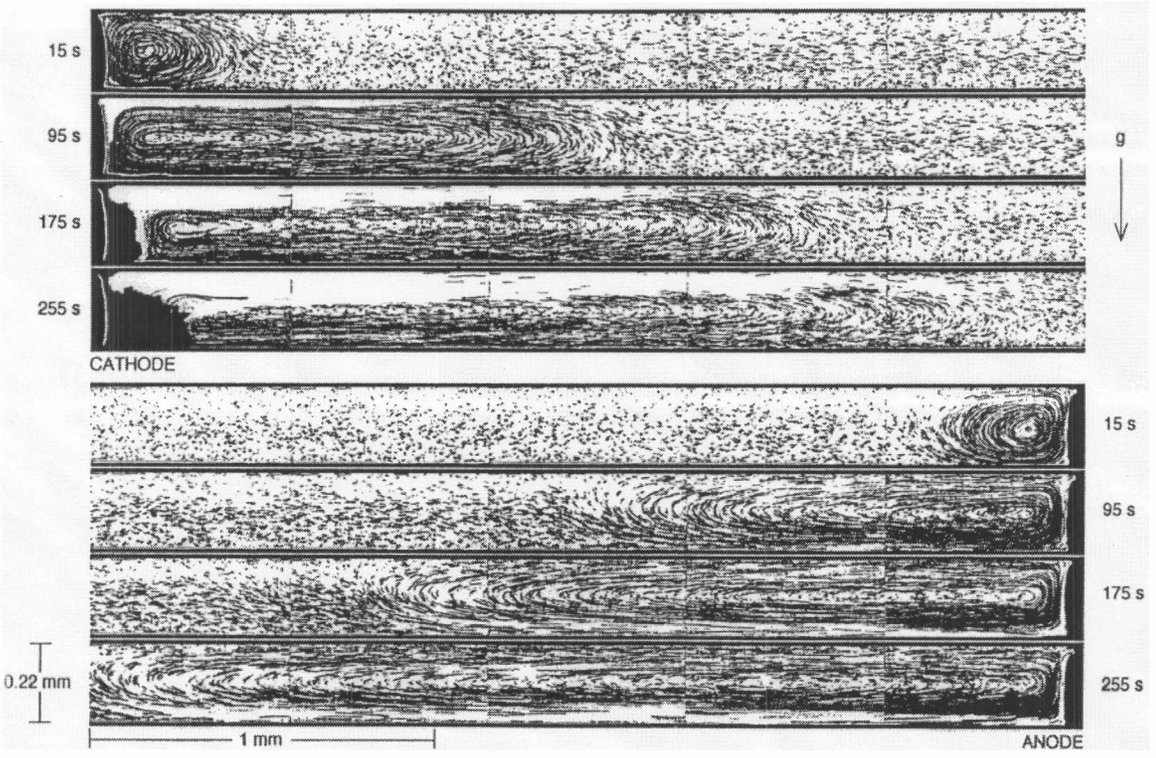


Figura 3.1: Vista lateral del experimento de ECD.

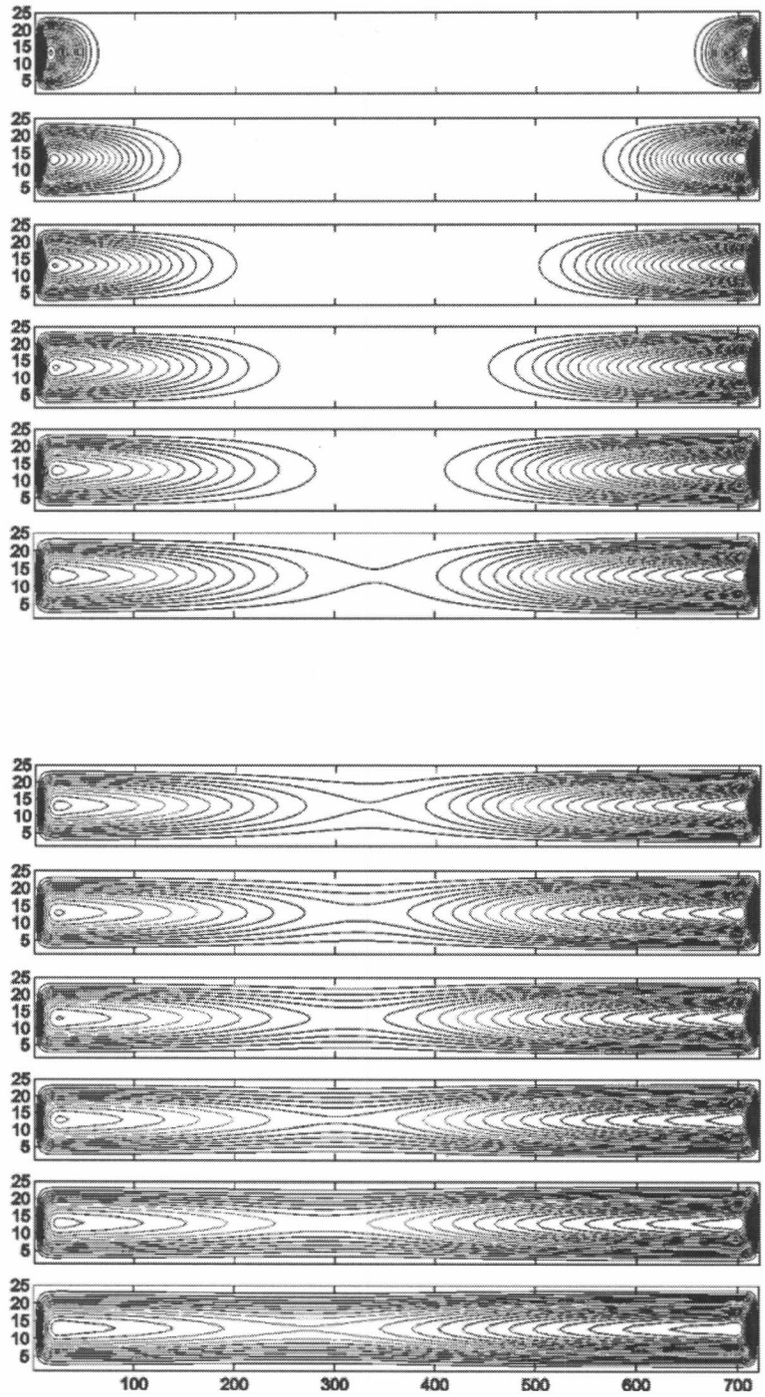


Figura 3.2: Vista lateral de simulación de la evolución, colisión y fusión del vórtice. Tiempos adimensionales de arriba a abajo: 10, 60, 120, 180, 240, 300, 360, 420, 480, 540, 600 y 660.

La fig. 3.3 muestra la evolución en espacio-tiempo de las curvas de nivel de la concentración de cationes adimensionalizada. Las concentraciones son afectadas por la gravitoconvección, y los resultados numéricos muestran que los frentes de concentración y convección evolucionan en forma conjunta.

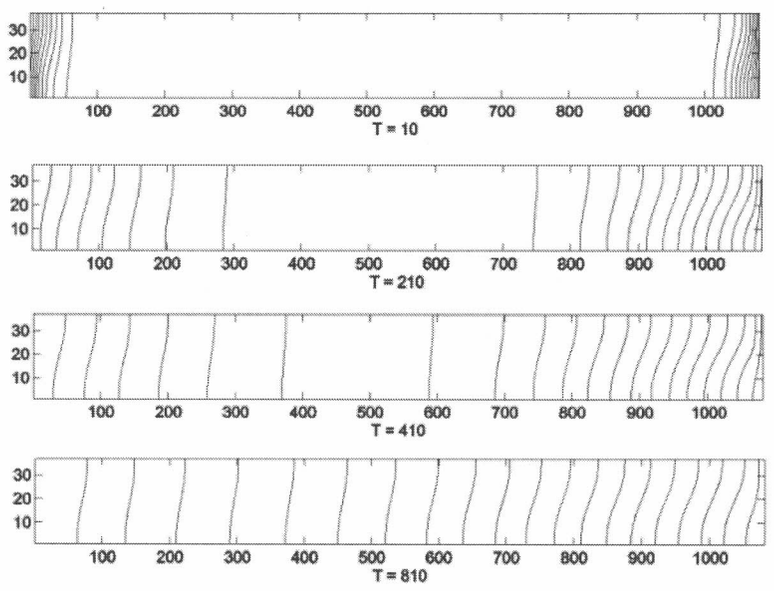


Figura 3.3: Curvas de nivel de la concentración de cationes para diferentes tiempos sin agregar glicerina.

En la fig. 3.4 se observa, para el instante $t = 300$, las curvas de nivel para la concentración de cationes adimensionalizada para diferentes viscosidades (0, 10, 20, 30 y 40 por ciento de glicerina). La figura revela que el frente de la concentración es retardado cuando la viscosidad aumenta, como se observa en los experimentos.

En la fig. 3.5 vemos las curvas de nivel, en el instante $t = 300$, de la función de corriente para diferentes viscosidades (0, 10, 20, 30 y 40 por ciento de glicerina), que muestra la influencia de la viscosidad en el retardo del avance del frente vorticoso.

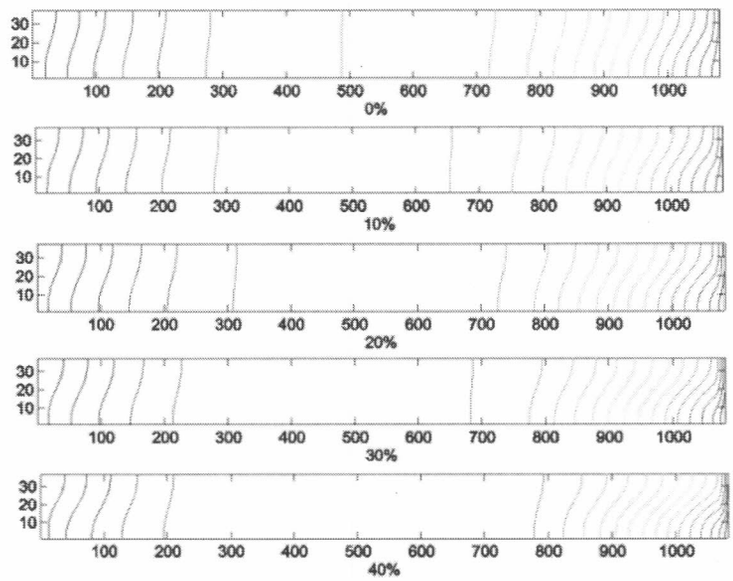


Figura 3.4: Curvas de nivel de la concentración de cationes en $t = 300$ para diferentes concentraciones de glicerina.

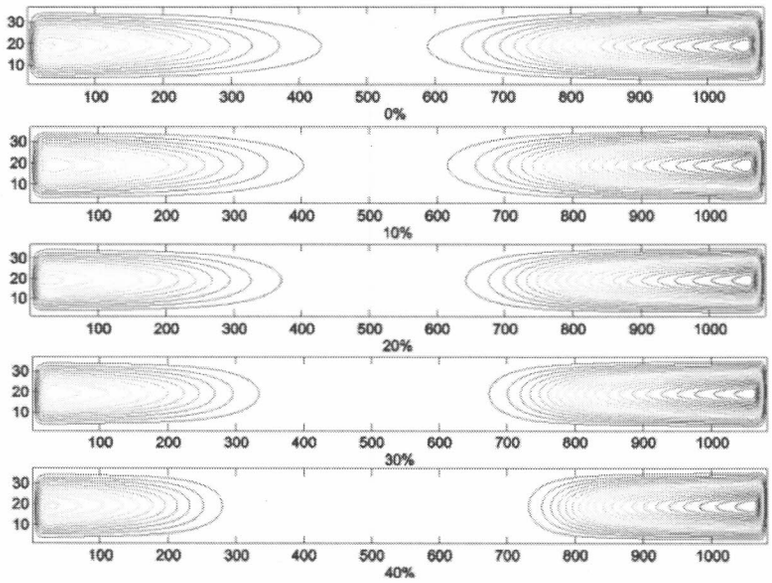


Figura 3.5: Capturas en $t = 300$ de la evolución del vórtice para diferentes viscosidades.

3.3 Performance

Las distintas estrategias diseñadas en el desarrollo de la solución paralela fueron evaluadas en nuestro cluster para poder determinar la configuración más eficiente, que sería utilizada en las corridas para simular el proceso de electrodeposición.

En primer lugar, compararemos las estrategias de descomposición unidimensional y bidimensional, sin incorporar a la evaluación las estrategias de intercambio de datos (en ambos casos se utilizará el método de múltiples mensajes por nodo). Tampoco se utilizará en la descomposición unidimensional el balanceo estático de la carga, porque esto supone poner en desventaja a la estrategia de descomposición bidimensional, que no tiene en cuenta la capacidad de cálculo de cada procesador en la asignación de los dominios. De este modo, en estas pruebas la performance global del Beowulf estará limitada por las computadoras de menor rendimiento. Los resultados serían semejantes a los que se pueden obtener en un cluster homogéneo, donde todas las computadoras fueran iguales a la de menor capacidad.

Para evaluar ambas estrategias se realizaron corridas para cada una de las configuraciones con un problema de tamaño fijo, de 720x30 (con una relación entre la cantidad de filas y columnas no muy diferente a la que se utilizaría en la simulación del proceso físico, de 1080x36). Las corridas se repitieron incrementando la cantidad de procesadores utilizados, desde 2 a 16.

Como la descomposición bidimensional permite realizar más de una descomposición para una misma cantidad de procesadores, tomaremos la configuración de mejor performance. En la siguiente tabla, se muestran junto a estos resultados los requerimientos de transferencia de datos en cada iteración en cada caso.

p	x	y	Tiempo	Datos a transferir
4	**2**	**2**	**86,58**	**750**
6	**3**	**2**	**59,61**	**510**
6	2	3	72,64	740
8	**4**	**2**	**45,46**	**390**
8	2	4	59,81	735
10	**5**	**2**	**37,00**	**318**
10	2	5	55,29	732
12	**6**	**2**	**31,81**	**270**
12	2	6	52,47	730
12	4	3	33,77	380
12	3	4	40,26	495
14	**7**	**2**	**27,69**	**236**
14	2	7	47,22	729
16	**8**	**2**	**25,39**	**210**
16	2	8	41,68	375
16	4	4	26,99	375

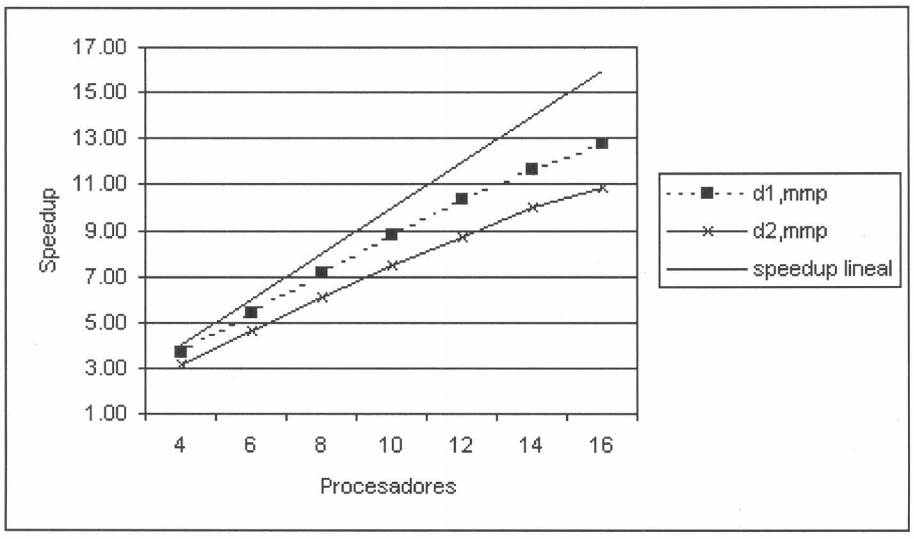


Figura 3.6: Comparación de la evolución del *speedup* para las estrategias de descomposición unidimensional y bidimensional.

Se puede apreciar claramente que, dada la relación entre la cantidad de columnas y filas de cada uno de los arreglos, las configuraciones con menor cantidad de filas en los subdominos (y) se ven favorecidas como consecuencia de una menor necesidad de transferencia de datos en cada iteración y, por ende, un menor tiempo de sincronización.

Seleccionando las corridas de mejor performance en la descomposición bidimensional para una misma cantidad de procesadores, en la fig. 3.6 se presentan los speedups obtenidos con cada una de las estrategias de descomposición de dominios.

Dado que, como se ha visto en 2.4.1 - *Descomposición de dominios - Evaluación de las técnicas*, la descomposición bidimensional no podrá obtener en nuestro problema mejor performance que la unidimensional, en los siguientes estudios sólo analizaremos el comportamiento de esta última estrategia.

Estudiaremos ahora el comportamiento de las estrategias de intercambio de datos implementadas:

1. Múltiples mensajes por procesador adyacente.

2. Único mensaje por procesador adyacente.

En la fig. 3.7 vemos la evolución del speedup en cada caso para un problema de tamaño fijo (180x30). Como se había previsto en el desarrollo del algoritmo paralelo, el esquema que utiliza sólo un mensaje por procesador adyacente en cada iteración, obtiene consistentemente un menor tiempo de sincronización de datos.

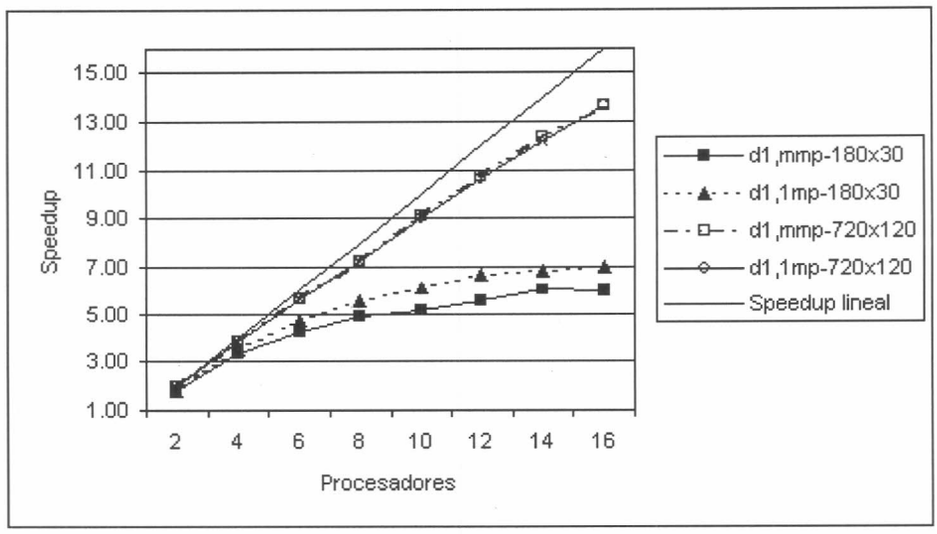


Figura 3.7: Comparación de la evolución del *speedup* para las estrategias de intercambio de datos (múltiples mensajes por procesador y mensaje único por procesador).

Estos resultados comprueban que el beneficio de limitar la cantidad de mensajes para evitar incurrir en el costo de inicialización o latencia de cada mensaje al enviarlos por separado, supera al tiempo en que se copian los datos al buffer intermedio en la memoria local.

Además, se puede apreciar que para trabajar un problema de dimensiones tan reducidas, tal como lo predice la ley de Amdahl, incorporar una mayor cantidad de procesadores no aumenta sustancialmente el speedup (incluso al utilizar 16 procesadores con la estrategia de múltiples mensajes, se verifica un deterioro en la performance al compararlo con una configuración integrada por 14 computadoras).

Sin embargo, vemos en el mismo gráfico los tiempos de ejecución al resolver el problema trabajando con arreglos de 720x120, 16 veces mayor que el tamaño original de la prueba. Como se puede observar, la evolución del speedup para las dos estrategias de intercambio de datos es casi idéntica (las líneas aparecen superpuestas en el gráfico). Aunque los tiempos de sincronización para el esquema que utiliza sólo un mensaje son siempre inferiores, esto ocurre porque la relación entre el tiempo de cálculo y el de sincronización es marcadamente superior en el nuevo caso. El análisis de complejidad presentado en el desarrollo del algoritmo nos indica que, mientras el tiempo de cálculo se incrementará en $O(mn)$ (en nuestro caso, $O(mn) = O(n^2)$), el intercambio de datos para la descomposición en 1d lo hará en $O(n)$.

En la fig. 3.8 se muestra, para corridas realizadas con 16 procesadores, la proporción correspondiente a las distintas fases de una iteración de nuestro algoritmo a medida que incrementamos el tamaño de problema a resolver. El porcentaje del tiempo que la simulación transcurre en la comunicación entre los procesos (tanto por sincronización de columnas o reducciones globales) decrece en comparación con el tiempo de cálculo. Por lo tanto, dis-

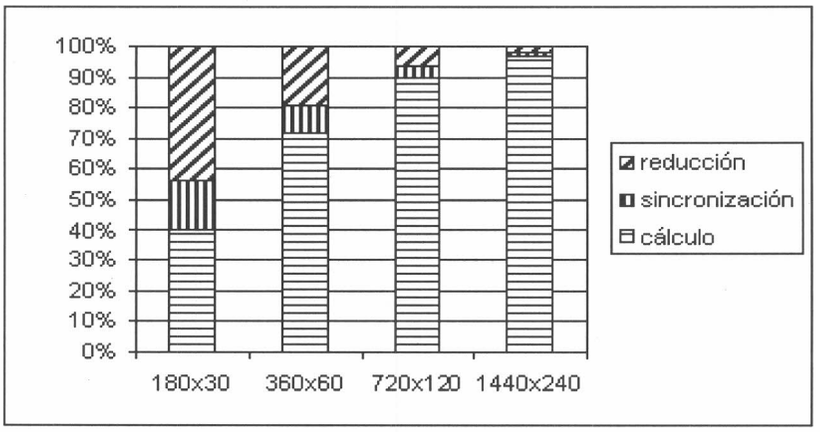


Figura 3.8: Proporción del tiempo de corrida utilizado en cada una de las fases de la simulación paralela.

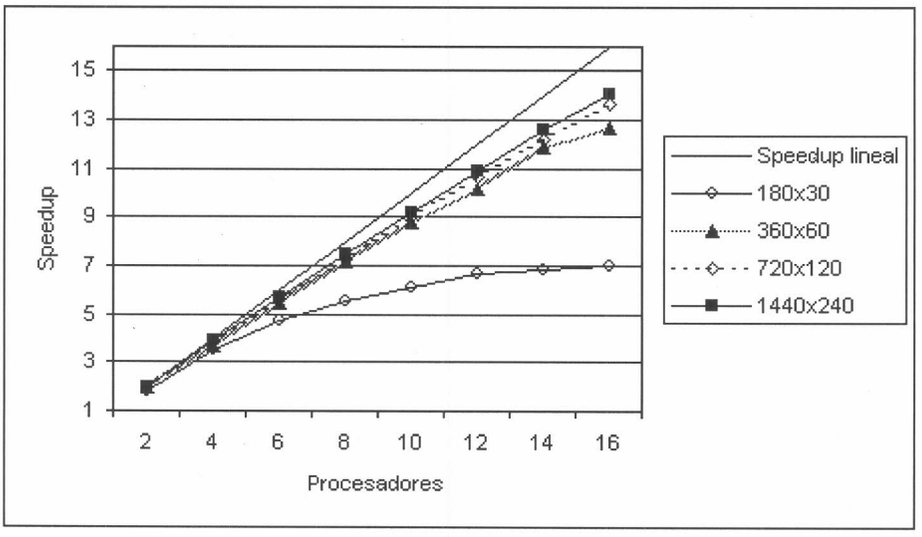


Figura 3.9: Evolución del speedup en descomposición unidimensional con un mensaje por procesador al aumentar el tamaño del problema.

minuye también el *overhead* de la solución paralela y el *speedup* se acerca a su valor óptimo (o *speedup lineal*).

De los resultados presentados hasta el momento, podemos concluir que la configuración que nos brinda mayor performance al trabajar en nuestro problema corresponde a la descomposición unidimensional con intercambio de un sólo mensaje por procesador adyacente. Con las próximas pruebas, observaremos su comportamiento a medida que aumentamos el tamaño del problema a resolver, duplicando en cada caso ambas dimensiones de las mallas utilizadas.

La fig. 3.9 confirma la observación de Gustafson (ver 1.3.6) sobre el criterio de escalabilidad, si incrementamos el tamaño del problema a la vez que agregamos procesadores para resolverlo, logramos alcanzar un nivel de speedup casi lineal (manteniendo una eficiencia alrededor de 0,9). Por el contrario, en el primero de los casos, de 180x30, se puede apreciar el marcado deterioro del speedup al utilizar más de seis procesadores (la eficiencia cae desde 0,79 con seis procesadores hasta 0,44 con 16).

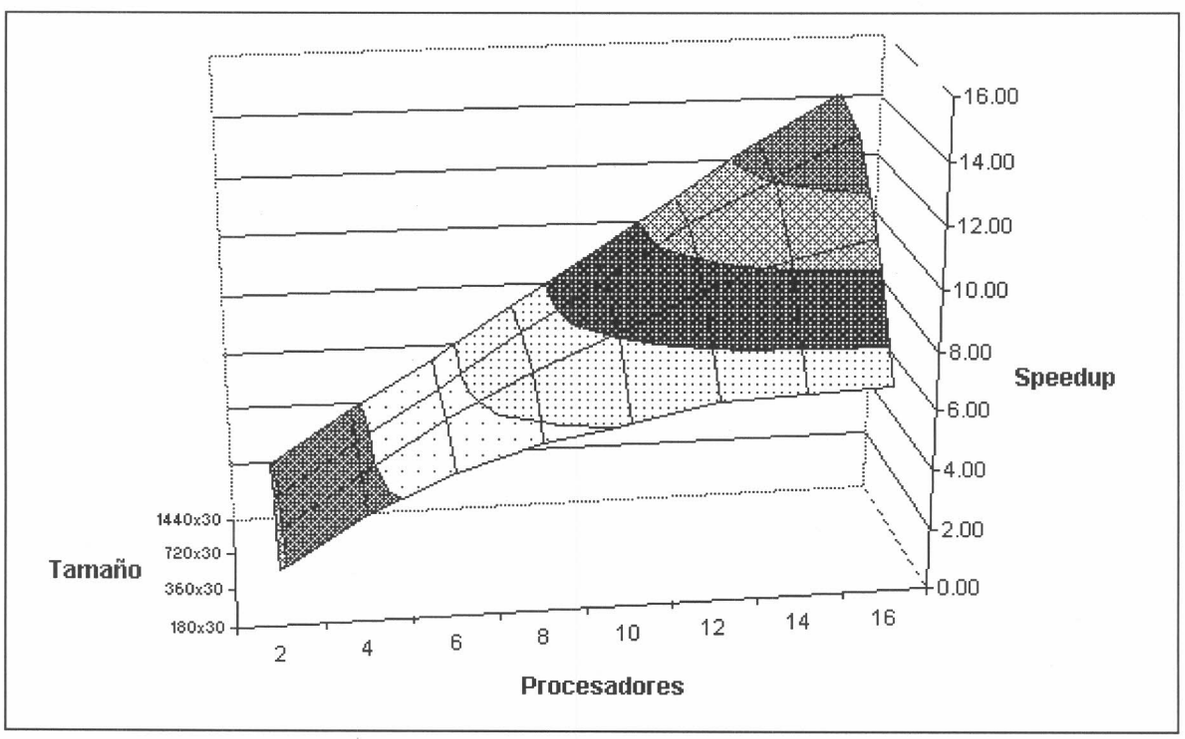


Figura 3.10: Evolución del speedup en función del tamaño del problema y la cantidad de procesadores utilizados.

A modo de resumen, podemos observar en la fig. 3.10 el speedup obtenido como función del tamaño del problema y de la cantidad de procesadores utilizados. Dada una cantidad fija de procesadores, vemos que el speedup del algoritmo paralelo aumenta al incrementar

el tamaño del problema. Para 16 procesadores la diferencia se manifiesta claramente, pero aún para dos procesadores el speedup se incrementa desde 1,83 (para 180x30) hasta 1,97 (en 1440x30), obteniendo prácticamente speedup lineal. Por otra parte, para un problema de tamaño fijo, la eficiencia de la solución paralela decrece a medida que se incrementan la cantidad de procesadores. Este deterioro es marcado para el problema de menor tamaño (de 180x30) porque, como hemos visto, la proporción del tiempo requerido por el intercambio de datos es mayor.

Finalmente, incorporamos el balanceo de carga en la asignación de los dominios a cada uno de los procesadores del cluster. En la fig. 3.11 se puede observar el rendimiento de ambas estrategias, y como la descomposición con balanceo obtiene tiempos inferiores en todos los casos[1]. En promedio, la mejora es de un 25% en el tiempo de ejecución, pero al incorporar las últimas seis computadoras (las de mayor capacidad de cálculo), la mejora supera el 30%.

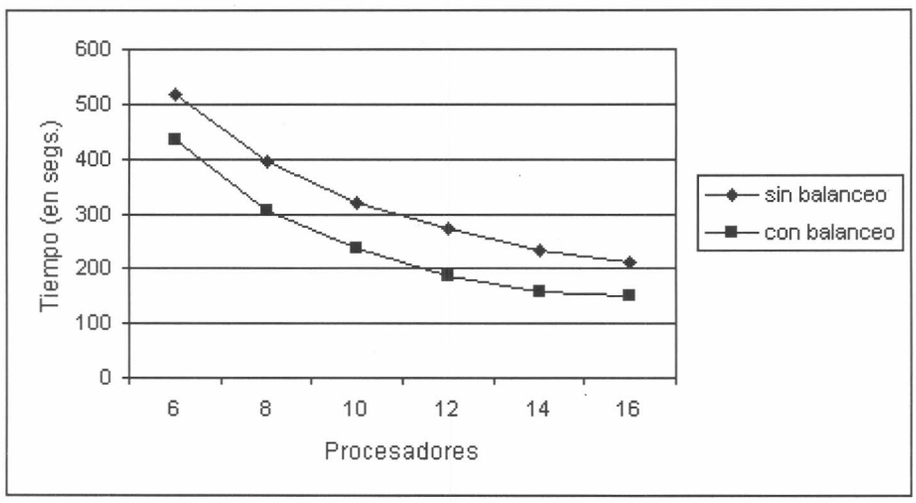


Figura 3.11: Tiempos de ejecución en una simulación de 1440x240 con y sin balanceo de carga.

3.4 Estimación de tiempos de corrida

Al realizar el análisis de complejidad del algoritmo paralelo de las diferentes variantes implementadas en 2.7.3, se determinaron varias constantes que dependen de las características de la computadora paralela utilizada para realizar las simulaciones numéricas. Las mediciones de performance efectuadas para determinar dichos valores en nuestro Bewoulf arrojaron los siguientes tiempos (en segundos):

[1]Las pruebas se realizan a partir de seis procesadores porque los tiempos son semejantes con los primeros cuatro, ya que estas computadoras son iguales y, por lo tanto, ambas estrategias realizan la misma asignación.

k^a_{calc}	4,27E-06	Tiempo de cálculo de las funciones por cada celda (Pentium III 733mhz).
k^b_{calc}	2,46E-06	Tiempo de cálculo de las funciones por cada celda (Athlon 950mhz).
k^c_{calc}	1,91E-06	Tiempo de cálculo de las funciones por cada celda (Athlon 1200mhz).
t_s	9,40E-05	Tiempo de latencia correspondiente a la inicialización en la transmisión de un mensaje.
t_c	8,46E-07	Tiempo requerido para la transmisión de un byte.
k_{red}	6,50E-04	Constante utilizada para determinar el costo de realizar una reducción global.

Comparando el tiempo de transmisión de un byte con el presentado para Ethernet en 1.4.1 - *El costo de comunicación*, donde $t_c = 8,90E-06$, observamos que la configuración de nuestro Beowulf, equipada con un switch Fast Ethernet, muestra el esperado incremento de 10x que resume la caracterizca diferencia entre ambas tecnologías.[2]

3.4.1 Resultados generales

Una vez determinados estos valores, realizamos corridas utilizando distinta cantidad de procesadores y variando el tamaño del problema para comparar los tiempos de ejecución en cada configuración con el valor estimado por nuestro análisis.

Se ejecutaron entonces 528 pruebas, que resultan de la combinación de las siguientes variables:

1. Cantidad de procesadores: 1, 2, 4, 6, 8, 10, 12, 14 y 16.
2. Cantidad de columnas: 180, 360, 720, 1440.
3. Cantidad de filas: 30, 60, 120, 240.
4. Estrategia de intercambio de datos: Múltiples mensajes por procesador y mensaje único por procesador.
5. Distribución balanceada de la carga.

Las estimaciones previstas determinaron valores con un error promedio de 7%. En las próximas dos secciones se presentan resultados en estimaciones con 16 procesadores, discriminando el tiempo estimado para cada una de las fases que componen la ejecución del algoritmo paralelo: cálculo, sincronización de datos y reducción global (para determinar los residuos de convergencia).

[2]En el caso del tiempo de latencia, no se produce un aumento de performance de la misma magnitud. Pero, nuevamente, dicho valor será poco representativo en el tiempo de comunicación total en la medida que el tamaño de los mensajes aumente.

3.4.2 Corridas sin balanceo

En las siguientes tablas se muestran los tiempos estimados y reales, y el error de aproximación resultante, a medida que se incrementa el tamaño del problema a resolver.[3] En primer lugar se presentan los resultados de las corridas con la estrategia de múltiples mensajes por procesador, que para todas las configuraciones observadas obtuvo un error promedio de 6% de estimación.

Dim.	Iter.	T_{calc}	T_{sinc}	T_{red}	Tiempo est.	Tiempo real	Error
180x30	3	4,32	3,58	4,70	12,60	10,98	13%
360x60	3	17,28	4,34	4,70	26,32	23,58	10%
720x120	3	69,12	5.87	4,70	79,68	80,19	1%
1440x240	2	184,32	5.94	3,13	193,39	209,34	8%

La siguiente tabla muestra los resultados de corridas con las configuraciones del caso anterior, pero utilizando la estrategia de intercambio de datos con un único mensaje por procesador. Los resultados obtenidos obtuvieron un error promedio de 5%.

Dim.	Iter.	T_{calc}	T_{sinc}	T_{red}	Tiempo est.	Tiempo real	Error
180x30	3	4,32	1,33	4,70	10,34	9,29	10%
360x60	3	17,28	2,09	4,70	24,06	21,86	9%
720x120	3	69,12	3,61	4,70	77,43	80,48	4%
1440x240	2	184,32	4,44	3,13	191,89	210,90	10%

3.4.3 Corridas con balanceo

Para poder estimar los tiempos de cálculo correspondientes a las corridas con balanceo se escala el tiempo de cálculo de las máquinas de menor performance del cluster (que, según se explicó en 2.6, limitan el rendimiento del resto cuando no se distribuye la carga en forma proporcional a la capacidad de cada computadora) al promedio de performance de las máquinas que integran las pruebas. Es decir, si se utilizan las cuatro computadoras originales del cluster, el factor de escala es 1, porque las cuatro computadoras tienen el mismo rendimiento. Si se incorporan los siguientes dos nodos (con procesadores AMD Athlon de 950mhz) para hacer corridas con seis procesadores, el factor resultante será $1,2$, ya que las máquinas más rápidas disminuirán el trabajo de las computadoras más lentas y, por lo tanto, cada iteración será terminada en menos tiempo. Finalmente, utilizando todas las computadoras de nuestro Beowulf, el factor para escalar el tiempo de cálculo será de $1,64$.

En la próxima tabla veremos nuevamente resultados de corridas utilizando la estrategia de múltiples mensajes por procesador, pero ahora se utilizará la distribución balanceada de la carga. Las estimaciones para estas pruebas obtuvieron valores con un error promedio de 8% respecto al tiempo de ejecución observado en la práctica.

[3] Como todas las corridas presentadas se realizan con 16 procesadores, el tiempo de reducción es igual para una misma cantidad de iteraciones, aún cuando aumente el tamaño del problema.

Dim.	Iter.	T_{calc}	T_{sinc}	T_{red}	Tiempo est.	Tiempo real	Error
180x30	3	3,18	3,58	4,70	11,45	9,03	21%
360x60	3	12,71	4,34	4,70	21,74	17,15	21%
720x120	3	50,82	5.87	4,70	61,39	51,00	17%
1440x240	2	135,53	5.94	3,13	144,60	151,42	5%

Por último, la siguiente tabla muestra los resultados para las pruebas con la estrategia de intercambio de datos con un único mensaje por procesador, con un error promedio de 7%.

Dim.	Iter.	T_{calc}	T_{sinc}	T_{red}	Tiempo est.	Tiempo real	Error
180x30	3	3,18	1,33	4,70	9,20	8,14	11%
360x60	3	12,71	2,09	4,70	19,49	16,18	17%
720x120	3	50,82	3,61	4,70	59,13	51,32	13%
1440x240	2	135,53	4,44	3,13	143,10	151,82	6%

3.5 Escalabilidad

El estudio de la escalabilidad de la solución paralela nos permite determinar para una eficiencia dada, la relación entre la cantidad de procesadores utilizados y el tamaño del problema a resolver para mantener la eficiencia preestablecida. Para esto, aplicamos la fórmula de *overhead* encontrada para nuestra solución en 2.7.4, a la definición de eficiencia, como se presentó en 1.3.6, manteniendo la relación entre la cantidad de columnas y filas utilizada en nuestro problema (de 30 columnas por fila).

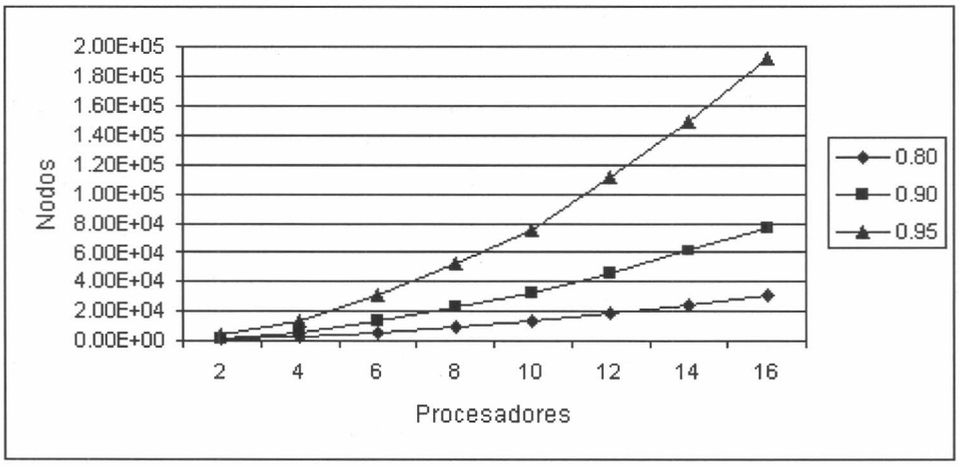


Figura 3.12: Tamaño de problema (mn) a resolver como función de la cantidad de procesadores utilizados para una eficiencia dada.

En la fig. 3.12 se muestran los resultados para distintos valores de eficiencia: 80%, 90% y 95%. Como es esperable, a medida que la eficiencia se acerca a 1,[4] el tamaño del problema a resolver crece en forma más rápida.

Según estos resultados, para obtener una eficiencia de 80% con 16 procesadores debemos trabajar en un problema de al menos 960x32, dimensiones similares al utilizado en nuestras simulaciones (de 1080x36).

3.6 Tiempos de corrida

Para finalizar el estudio de performance de nuestro algoritmo paralelo, comparamos el tiempo requerido por las simulaciones realizadas para el trabajo [17][5], con el que hubiera sido requerido por el algoritmo secuencial con distintas configuraciones (en orden: el procesador más lento de nuestro Beowulf, el procesador *virtual* de performance promedio, y el procesador más rápido del cluster).

El tiempo de ejecución paralela se obtiene utilizando la descomposición unidimensional e intercambio de datos con un único mensaje por procesador. Los tiempos de las corridas en cada caso son:

Tipo simulación	**Configuración**	**Tiempo**
Secuencial	Intel Pentium III 733mhz	4 días 18:15
Secuencial	Promedio Beowulf	2 días 21:47
Secuencial	AMD Athlon 1200mhz	2 días 06:25
Paralelo	Sin balanceo	0 días 09:22
Paralelo	Con balanceo	0 días 05:43

Se puede observar que, aún con la mejor computadora secuencial disponible, se requieren más de dos días adicionales de trabajo con respecto a la mejor corrida en paralelo en nuestro Beowulf. El speedup del algoritmo paralelo con balanceo respecto al procesador de performance promedio es superior a 12.

También es apreciable la mejora en el rendimiento obtenida por medio de la implementación de la asignación de dominios proporcional a la capacidad de cada procesador del cluster (que apenas supera el 60% del tiempo utilizado por la versión sin balanceo).

La magnitud de la diferencia de performance a favor de la solución paralela nos permite en la práctica acelerar el estudio del modelo matemático, al poder ejecutar una mayor cantidad de simulaciones en un mismo período de tiempo.

---

[4]Es decir, se aprovecha al 100% los procesadores disponibles, lo que supone que la solución paralela no incurre en ningún *overhead*.

[5]Se realizaron cinco corridas con un tamaño de problema de 1080x36. Una de las corridas hasta $t = 810$ y las cuatro restantes, con las distintas proporciones de glicerina, hasta $t = 410$.

Capítulo 4

Conclusiones

Utilizando un conjunto interconectado de computadoras personales de bajo costo, mediante una implementación paralela de la solución numérica del proceso físico, hemos logrado realizar simulaciones más próximas al experimento real que las obtenidas con una implementación secuencial. Las corridas, además, requieren un tiempo razonable de ejecución, que permite la exploración de distintas condiciones del problema (como variar la viscosidad, agregando glicerina).

La configuración de un Beowulf, a partir de componentes fabricados para el mercado de computadoras personales, permitió alcanzar, con un presupuesto reducido, un rendimiento antes sólo disponible en supercomputadoras. Se evaluaron distintas técnicas de descomposición y sincronización de datos para escoger la más conveniente en nuestro caso (dependiente tanto del problema como de la computadora paralela disponible).

La flexibilidad del diseño de la solución implementada (descripto en el apéndice B), nos permitió desarrollar y combinar las distintas estrategias de comunicación sin requerir cambios en el método numérico de cálculo, simplificando (al eliminar escenarios de errores en la implementación) la evolución de nuestro estudio para optimizar la performance de la solución. Al incorporar en una etapa posterior del trabajo nuevas y más potentes computadoras al *cluster* original, este diseño también nos brindó la posibilidad de aprovechar de manera inmediata toda su capacidad de cálculo al minimizar el trabajo requerido para implementar una descomposición balanceada de la carga.

Por último, el análisis de complejidad del algoritmo paralelo nos permite predecir el comportamiento de nuestra implementación en diferentes escenarios, como incorporar nuevas computadoras al Beowulf o incrementar el tamaño de las grillas utilizadas para la simulación. Estos resultados son fácilmente extensibles a otras soluciones secuenciales ya realizadas y, por lo tanto, podemos también estimar los beneficios de portar cada una de ellas a una versión paralela.[1]

[1]Tarea que también se verá simplificada por la reutilización del *framework* de descomposición y sincronización de datos desarrollado en este trabajo.

4.1 Trabajo futuro

Las áreas previstas para trabajos futuros consisten en incorporar a la solución numérica una etapa de agregación, que consiste en simular el crecimiento del depósito sobre el cátodo. Para esto se utiliza un método DBM generalizado ([18], [21], [24]), que consiste en avanzar la interface del crecimiento de un nodo, con probabilidad proporcional al máximo flujo entrante de cationes a la agregación. La implementación de este proceso requerirá realizar nuevas operaciones globales para poder determinar el flujo en la interface en toda la malla y un generador aleatorio paralelo para seleccionar las partículas a agregar mediante la regla estocástica.

Por otra parte, en las áreas donde evoluciona el frente de convección y concentración de cationes, para evitar problemas de inestabilidad numérica se debe utilizar una malla con mayor cantidad de nodos. La implementación actual, sin embargo, utiliza un esquema de grillas equiespaciadas. Entonces, para focalizar los recursos de cálculo en la evolución del frente se pueden utilizar técnicas de multi-grilla, aumentando la precisión de cálculo en dichas áreas, evitando malgastar tiempo de cálculo en zonas más estables. Este esquema requeriría la implementación de un método de redistribución dinámica de la carga, ya que el área donde se requiere mayor precisión de cálculo puede desplazarse al dominio de cálculo asignado a otro de los procesadores.

Asumiendo que la carga había sido distribuida anteriormente en forma balanceada, asignar una mayor cantidad de celdas a uno de los procesadores habrá de disminuir la eficiencia global del cluster, ya que el resto deberá esperar que este complete su iteración antes de proceder a realizar la siguiente (ya que las mismas son ejecutadas en forma sincronizada).

Finalmente, la evolución natural del trabajo consiste en desarrollar la simulación numérica del proceso en 3d. En este caso, debido a que nuestro Beowulf está integrado por computadoras heterogéneas, será necesario implementar técnicas de descomposición de dominios basadas en la técnica de bisección recursiva, como se discutió en 2.4.

Bibliografía

[1] G. Amdahl, *The validity of the single processor approach to achieving large scale computing capabilities*, AFIPS conference proceedings, Spring Joint Computing Conference **30** (1967), 483–485.

[2] X. Cai, W. D. Gropp, D. E. Keyes, R. G. Melvin, and D. P. Young, *Parallel newton-krylov-schwarz algorithms for the transonic full potential equation*, SIAM Journal on Scientific Computing **19** (1998), no. 1, 246–265.

[3] N. Chrisochoides, E. Houstis, and J. Rice, *Mapping algorithms and software environment for data parallel pde iterative solvers*, Journal of Parallel and Distributed Computing **21** (1994), no. 1, 75–95.

[4] I. T. Christou and R. R. Meyer, *Optimal equi-partition of rectangular domains for parallel computation*, Journal of Global Optimization **8** (1996), no. 1, 15–34.

[5] R. Diekmann, D. Meyer, and B. Monien, *Parallel decomposition of unstructured FEM-meshes*, Concurrency: Practice and Experience **10** (1998), no. 1, 53–72.

[6] J. J. Dongarra and T. Dunigan, *Message-passing performance of various computers*, Concurrency: Practice and Experience **9** (1997), no. 10, 915–926.

[7] M.J. Flynn, *Some computer organizations and their effectiveness*, IEEE Transactions on Computing **21** (1972), no. 9, 948–960.

[8] Ian T. Foster, *Designing and building parallel programs : Concepts and tools for parallel software engineering*, Addison-Wesley Pub Co, 1995.

[9] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jaing, and V. Sunderam, *PVM: Parallel Virtual Machine*, MIT Press, Boston, 1994.

[10] G. Geist, J. Kohl, and P. Papadopoulos, *PVM and MPI: A comparison of features*, Calculateurs Paralleles **8** (1996), no. 2.

[11] W. Gropp and E. L. Lusk, *Why are PVM and MPI so different?*, Fourth European PVM - MPI Users' Group Meeting (1997).

[12] J. L. Gustafson, G. R. Montry, and R. E. Benner, *Development of parallel methods for a 1024-processor hypercube*, SIAM Journal of Scientific and Statistical Computing **9** (1988), no. 4, 609–638.

[13] J.L. Gustafson, *Amdahl's law re-evaluated*, Communications of the ACM **31** (1988), no. 5, 532–533.

[14] J. M. Huth, H. L. Swinney, W. McCormick, A. Kuhn, and F. Argoul, *Role of convection in thin layer electrodeposition*, Physical Review E **51** (1995), 3444.

[15] C. Lee, Y. Wang, and T. Yang, *Global optimization for mapping parallel image processing tasks on distributed memory machines*, Journal of Parallel and Distributed Computing **45** (1997), no. 1, 29–45.

[16] G. Marshall, *Solución numérica de ecuaciones diferenciales - Tomo II: Ecuaciones en derivadas parciales*, Reverte, Buenos Aires, 1986.

[17] G. Marshall, S. Dengra, E. Arias, F.V. Molina, M. Vallieres, and G. Gonzalez, *Ion transport modelling in realistic thin-layer ECD for gravitoconvection prevailing regimes*, Symposium K1, Electrochemical Processing in ULSI Fabrication and Electrodeposition of and on Semiconductors IV, 199th Meeting of The Electrochemical Society - Washington, DC, USA, 25 al 30 de marzo de 2001 (en prensa).

[18] G. Marshall and P. Mocskos, *A growth model for ramified electrochemical deposition in the presence of diffusion, migration and electroconvection*, Physical Review E **55** (1997), no. 1, 549–563.

[19] G. Marshall, P. Mocskos, H. L. Swinney, and J. M. Huth, *Buoyancy and electrically driven convection models in thin-layer electrodeposition*, Physical Review E **59** (1999), no. 2, 2157–2167.

[20] A. N. Moga and M. Gabbouj, *Parallel image component labeling with watershed transformations*, IEEE Transactions on Pattern Analysis and Machine Intelligence **19** (1997), no. 5, 441–450.

[21] L. Niemeyer, L. Pietronero, and H. Wiesmann, *Fractal dimension of dieletric breakdown*, Physical Review Letters **52** (1984), 1033–1036.

[22] HPCC Program Office, *NASA HPCC press release*, http://ess.gsfc.nasa.gov/news.sc96.beowulf.html.

[23] P. Pacheco, *Parallel programming with MPI*, Morgan Kaufmann Publishers, 1996.

[24] L. Pietronero, *Theoretical concepts for fractal growth*, Special issue of Physica D **38** (1989), 279.

[25] D. Ridge, D. Becker, P. Merkey, and T. Sterling, *Beowulf: Harnessing the power of parallelism in a pile-of-pcs*, Proceedings, IEEE Aerospace (1997).

[26] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, Computing Systems in Engineering **2** (1991), 135–148.

[27] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The complete reference vol. 1, the MPI core*, MIT Press, Boston, 1998.

[28] A. van der Steen and J. Dongarra, *Overview of high performance computers*, To appear in Handbook of massive data sets, Kluwer Academic Publishers., 2001.

[29] B. Wilkinson and C. Michael Allen, *Parallel programming: Techniques and applications using networked work*, Prentice Hall, 1998.

[30] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency **3** (1991), 457–481.

[31] J. Yackel, R. R. Meyer, and I. T. Christou, *Minimum perimeter domain assignment*, Mathematical Programming **78** (1997), no. 2, 283–303.

Apéndice A

Speedy, nuestro Beowulf

En esta sección describimos las características técnicas de nuestro Beowulf, **Speedy**, tanto de los recursos de hardware que lo integran como el software utilizado como soporte de nuestra implementación paralela de la simulación numérica. El cluster está integrado por 16 computadoras y es heterogéneo, al estar integrado por tres modelos diferentes de procesadores.

Una de las máquinas del cluster se denomina *master* porque es la encargada de lanzar la ejecución de los procesos en el resto de las computadoras del cluster (a las que llamaremos máquinas *slave*), administrar los usuarios y sus permisos en el Beowulf, y mantener la información de todos los usuarios en sus respectivos directorios *home* (que serán accededidos desde las máquinas *slave* durante la ejecución de las simulaciones).

En el esquema de una máquina paralela Beowulf presentado en la fig. 1.6 aparece fuera del cluster una terminal encargada de la administración y utilización de los recursos del cluster. En nuestro caso, esta computadora es *master*, que toma parte también en las simulaciones para aprovechar al máximo los recursos disponibles. Este modo de operación es hoy posible debido a que no se necesitan realizar varias tareas en forma simultánea y, por lo tanto, cuando el cluster no está realizando corridas para resolver nuestro problema, la máquina *master* puede ser utilizada para el análisis de los resultados obtenidos. A partir de la disponibilidad de una nueva estación de trabajo, las tareas correspondientes al *front-end* del cluster dejarán de ser realizadas por *master*.

A.1 Hardware

Las 16 computadoras que integran nuestro cluster se encuentran asignadas según se presenta a continuación:

Cantidad	Nodos	Procesador
4	master,node2...node4	Intel Pentium III 733mhz
6	node5...node10	AMD Athlon 950mhz
6	node11...node16	AMD Athlon 1200mhz

Cada una de las computadoras está posee 64 megabytes de memoria RAM y discos rígidos de 10 gigabytes. Además se dispone de 5 unidades de CDROM para la instalación y configuración de cada máquina.

La interconexión de las máquinas para la transferencia de información se realiza por medio de placas 3Com Fast Ethernet (100 megabits), conectadas a un switch 3Com Office Connect 1600 10/100, con 16 ports que permiten la comunicación individual entre dos nodos (ya que no se comparte el medio de intercambio de datos entre las computadoras conectadas, como es el caso de un hub).

A.1.1 Evolución histórica

En nuestra experiencia, aproximadamente seis meses después de armar nuestro primer cluster con cuatro máquinas, logramos incrementar en seis la cantidad de estaciones de trabajo. Las nuevas máquinas, sin embargo, no correspondían a la especificación original debido a que la constante mejora de los componentes disponibles en el mercado a un precio semejante nos permitió incorporar computadoras de mayor rendimiento. Finalmente, poco tiempo antes de terminar este trabajo, se incorporaron seis nuevas máquinas (nuevamente con distintas características técnicas a las anteriores).

En la siguiente tabla se muestra como se fueron las distintas máquinas que hoy componen nuestro Beowulf[1].

Fecha	Cantidad	Descripción computadora
11/2000	4	Intel Pentium III 733mhz
05/2001	2	Intel Pentium III 800mhz
	2	AMD Athlon 950mhz
07/2001	2	AMD Athlon 950mhz
10/2001	6	AMD Athlon 1,2ghz

A.2 Software

El software instalado en las máquinas consiste en todos los casos del sistema operativo Linux, de la distribución Mandrake v7.1 de Mandrake Soft, con el kernel 2.2.15 compilado con optimizaciones para máquinas de tipo Pentium.

A.2.1 Configuración de *master*

Como se describe anteriormente, una de las máquinas actúa como *master* en el Beowulf y, por lo tanto, la configuración de los paquetes instalados en esta máquina difiere del resto de las que integran el cluster (*slaves*). Estos responden a las necesidades de las siguientes áreas:

[1]Los Intel Pentium III de 800 mhz fueron reemplazados por dos AMD Athlon 950mhz en octubre del 2001.

1. Desarrollo de la implementación paralela:

   (a) Compilador C/C++: gcc v2.95.

   (b) MPI: MPICH v1.2.2 del Departamento de matemática y ciencias de la computación del Laboratorio Nacional de Argonne, Estados Unidos.[2]

2. Configuración de la red del cluster:

   (a) Servidor de NIS: para unificar la administración de usuarios y permisos en *master*, y que será utilizado por las distintas máquinas *slaves* para permitir el acceso y ejecución de programas en estas últimas.

   (b) Servidor de NFS: para poder acceder desde las máquinas *slaves* al directorio *home* correspondiente al usuario que ejecuta un programa con MPI (ya que este debe estar disponible en cada una de las máquinas que trabajarán en el problema en el mismo directorio).

3. Análisis de resultados de simulaciones: Matlab 5.3.0

A.2.2 Configuración de máquinas *slaves*

Las máquinas *slaves* no requieren la instalación de una gran cantidad de paquetes además del soporte básico de Linux para trabajar en red. En particular, utilizan el cliente de NIS para verificar permisos en el acceso de usuarios (o ejecución remota de programas) y el cliente de NFS para montar el directorio de *home* de quien se conecta.

La instalación de las máquinas slave está automatizada a partir de la replicación de la configuración del nodo dos (la primer computadora de tipo *slave* del cluster). Una vez que se seleccionaron los paquetes y detalles de instalación requeridos por la distribución de Linux utilizada, se generó un diskette de instalación automática que repite el mismo proceso en cada una las máquinas.

De esta manera, luego de realizar una pequeña cantidad de modificaciones a la instalación obtenida (para deshabilitar servicios activados por defecto que son innecesarios al participar del Beowulf), cada una de las máquinas *slaves* están configuradas de manera idéntica. Este ambiente simplificado de administración y configuración de las máquinas ha permitido incorporar nuevas computadoras en forma rápida (al minimizar la cantidad de errores humanos en el proceso) y, además, permite descartar variaciones en la performance de alguna de las máquinas debido a la configuración inadecuada de algunos de los servicios.

[2]Ver `http://www-unix.mcs.anl.gov/mpi/mpich`

A.3 Trabajo futuro

En la evolución de nuestro cluster, aunque la configuración de las máquinas permanece en un estado estable, existen dos áreas donde se planea realizar mejoras. En primer lugar, toda vez que una mayor cantidad de usuarios requerirá la utilización del Beowulf (desde investigadores a alumnos), se necesita implementar un *scheduler*, encargado de ejecutar un conjunto de tareas en subgrupos de las computadoras del cluster en horarios previamente asignados y reservados a este fin. Entre los productos actualmente disponibles, un relevamiento de otros Beowulfs presenta como la opción más aceptada a la combinación de *PBS (Portable Batch System)*[3], para manejar la cola de tareas; y el *Maui Scheduler*[4], para optimizar la asignación de las tareas a las computadoras disponibles.

Por otra parte, aunque el esquema de replicación de las máquinas *slave* ha brindado los beneficios que se presentaron en la sección anterior, presenta limitaciones en cuanto a que no es posible realizar cambios en el cluster en forma unificada (como incorporar nuevos paquetes en cada una de las máquinas o modificar la configuración de alguno de los servicios utilizados). Existen actualmente soluciones como *SystemImager*[5] que permiten mantener en *master* una única imagen de la configuración del resto de las máquinas del cluster. Al ser modificada, los cambios producidos son replicados en cada una de las computadoras *slave*.

[3]ver `http://www.openpbs.org`
[4]ver `http://mauischeduler.sourceforge.net/`
[5]ver `http://www.systemimager.org`

Apéndice B

Diseño de la implementación

B.1 Introducción

La implementación de la simulación numérica paralela del proceso de deposición electroquímica fue realizada en C++, utilizando técnicas de orientación a objetos para simplificar y hacer flexible a la solución para poder experimentar con una variedad de estrategias de descomposición y asignación de dominios a cada procesador.

Se realizó, entonces, un mayor trabajo de diseño en la elaboración de una pequeña biblioteca que encapsula el manejo de las estructuras de datos y la sincronización de datos entre los distintos procesadores.

La implementación del método de cálculo se realizó en los siguientes entornos:

1. Linux
    - Mandrake Linux 7.1
    - GCC 2.95
    - MPICH 1.2.2 del Departamento de matemática y ciencias de la computación del Laboratorio Nacional de Argonne, Estados Unidos.
2. Windows NT
    - Microsoft Windows NT Server 4.0 Service Pack 6
    - Microsoft Visual C++ 6.0 Service Pack 5
    - WMPI 1.3 del Grupo de sistemas confiables del Departamento de ingeniería informática de la Universidad de Coimbra, Portugal.[1] [2]

[1] Ver `http://dsg.dei.uc.pt/w32mpi`

[2] A partir de la versión 1.5, disponible de Critical Software (`http://www.criticalsoftware.com/wmpi/home/index`

Durante el desarrollo se mantuvo una única versión del código fuente (sin utilizar mecanismos de compilación condicional para salvar diferencias entre ambas plataformas), demostrando los beneficios la utilización de herramientas standard, ya sea el lenguaje C++ como la biblioteca MPI.

B.2 Objetivos

Entre los objetivos que se establecieron inicialmente para el diseño se encuentran:

1. Minimizar la dependencia entre el algoritmo de solución numérica y el de distribución de la carga y los datos en cada procesador.

2. Simplificar la utilización del esquema de distribución de datos para permitir una fácil transición del algoritmo serial a la máquina paralela.

3. Dotar al esquema de distribución de flexibilidad para poder implementar y evaluar distintas estrategias de distribución de datos para seleccionar aquella que brinde una mejor performance.

4. Permitir la ejecución del programa tanto en una máquina paralela como en una serial sin realizar cambios en la implementación. De este modo, se simplifica el desarrollo y verificación de la solución numérica, ya que la depuración de un programa en una máquina serial se puede realizar con herramientas convencionales.

B.3 Descripción del diseño de la implementación

En la fig. B.1 se puede observar la estructura estática de clases que componen el algoritmo paralelo implementado. El grupo de clases que permiten definir cada una de las funciones del sistema de ecuaciones a resolver en la simulación numérica del problema son:

1. *proceso*: Un objeto de esta instancia se encarga de coordinar al resto de los integrantes para realizar la simulación. Posee un conjunto de instancias de los distintos *metodos* implementados y un *distribuidor* (seleccionado en la inicialización y que permite variar la estrategia de intercambio de mensajes). Se encarga de invocar a las funciones encargadas de la inicialización de cada uno de los *metodos*, y luego realiza el ciclo detallado en la sección 2.2.2 - *Simulación numérica secuencial*, donde se realiza la simulación numérica del proceso mientras este no converja ó se alcance el límite preestablecido de iteraciones a ejecutar.

2. *metodo_base*: Define el grupo de rutinas que debe implementar cada uno de los métodos utilizados en la simulación, entre los que se destacan la inicialización y actualización de los datos de cada uno de sus elementos. Se encarga de implementar casi todas

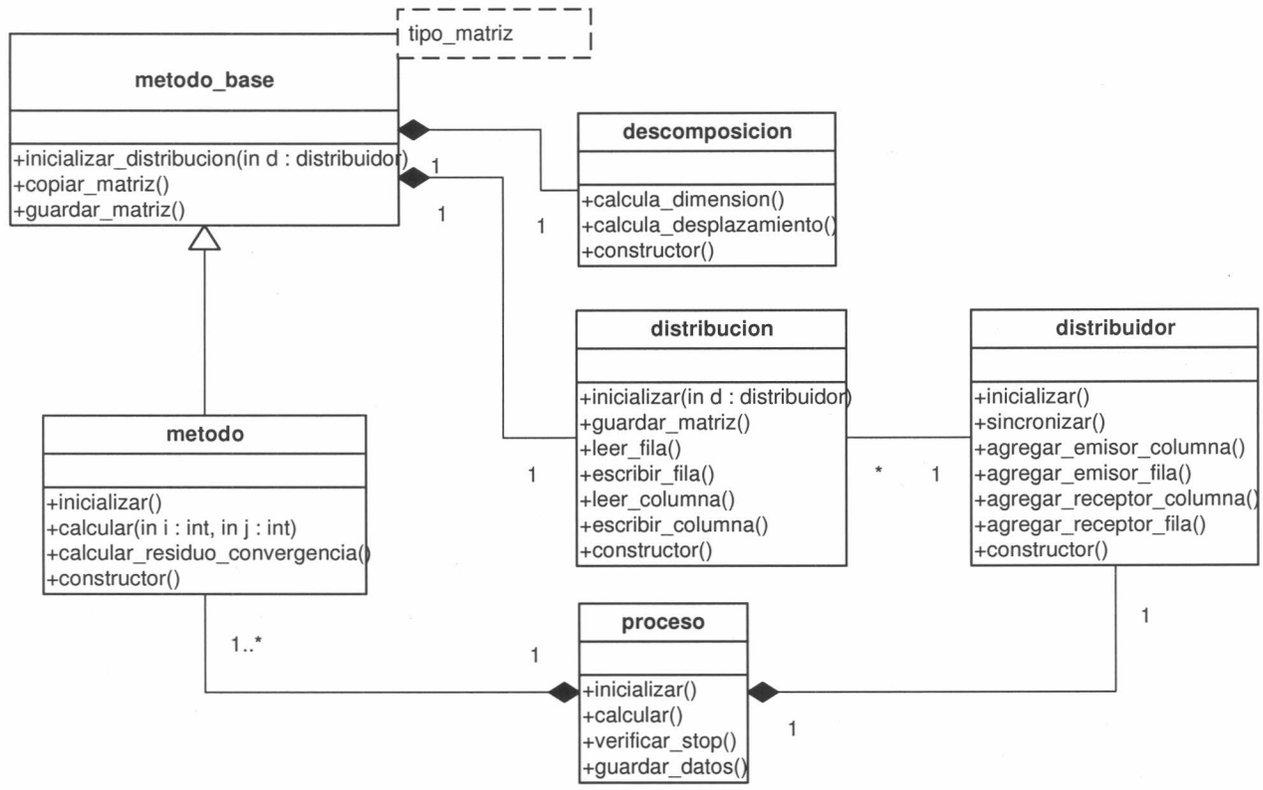


Figura B.1: Diagrama de clases que se utilizan en el algoritmo paralelo.

las tareas que debe realizar un *metodo* para poder ser utilizado dentro del esquema diseñado. Cada *metodo*, entonces, debe heredar la implementación de esta clase y proveer en general poco más que dos rutinas, uno para inicializar el arreglo asociado y otro para operar en cada iteración.

3. *metodo*: Aunque en la implementación final de la simulación numérica no existe ninguna clase llamada *metodo*, existe todo un grupo de clases con este prefijo que corresponden a cada una de las funciones resueltas en la simulación[3]. De esta forma, es posible aislar la implementación de cada una de ellas e incluso utilizar distintos métodos de cálculo en cada caso. Además, el código que se agrega para la solución de cada nueva función sólo se dedica a resolverla, ya que dispone automáticamente de toda la infraestructura necesaria para el manejo de los arreglos utilizados y su sincronización con los otros procesadores.

[3]En nuestro caso: `metodo_concentracion_cationes`, `metodo_concentracion_aniones`, `metodo_potencial_electrico`, `metodo_funcion_de_vorticidad`, `metodo_funcion_de_corriente`.

Las clases que se enumeran a continuación se utilizan para implementar la distintas estrategias de descomposición de dominios e intercambio de datos definidas en el desarrollo del algoritmo paralelo:

1. *metodo_grupo*: Representa un conjunto de objetos que implementan la interface definida en *metodo_base_interface*, y permite ejecutar acciones comunes por medio de una única invocación grupal (que luego es extendida a cada uno de los integrantes).

2. *descomposicion*: Asigna los dominios de cálculo correspondiente a cada uno de los procesadores. Las clases implementadas para realizar las estrategias previstas son: `descomposicion_1d`, `descomposicion_1d_balanceo` y `descomposicion_2d`.

3. *distribucion*: Ejecuta las tareas de sincronización de los datos entre los distintos procesadores de acuerdo al tipo estrategia de intercambio de información utilizada. Existe dos implementaciones de esta interface: `distribucion_1d`[4] y `distribucion_2d`.

4. *distribuidor:* En general se utiliza sólo una instancia de la clase para poder llevar a cabo las tareas de distribución de los datos[5]. Cada uno de los métodos implementados establece en la inicialización del proceso, por medio de instancias de objetos del tipo *distribucion*, que operaciones de transmisión y/o recepción de datos realizará en cada iteración. Luego, el distribuidor utiliza la información almacenada para llevar a cabo las operaciones de intercambio de datos según la estrategia que se haya seleccionado[6]. Las dos estrategias presentadas en la sección 2.5 se implementaron por medio de las clases: `distribuidor_multiples_mensajes` y `distribuidor_mensaje_unico`.

De acuerdo a la división de tareas asignadas a cada uno de los grupos de clases que acabamos de describir, al iniciar la ejecución de la simulación numérica se puede establecer, por medio de parámetros al programa, la combinación de las distintas estrategias a utilizar en la corrida.

Por otra parte, se pueden incorporar nuevas estrategias de descomposición de dominios e intercambio de datos (con nuevas clases que deriven de *descomposicion* y *distribucion*, respectivamente), sin requerir modificaciones en el código correspondiente al método numérico. En el caso de nuestro trabajo, al incorporar nuevas computadoras a nuestro Beowulf de mayor capacidad de cálculo, se implementó una nueva variante de descomposición de dominios balanceada (como se describe en la sección 2.6).

---

[4]No existe una distribución particular para el caso unidimensional con balanceo porque se transmiten la misma cantidad de datos (sólo varían las columnas que se utilizan en los cortes de los dominios).

[5]En forma más específica, se suele utilizar para sincronizar la información entre procesos.

[6]Realizamos este proceso llamando a la función `sincronizar()` luego de cada iteración de cálculo (en el paso 2.1.2 del algoritmo paralelo presentado en la sección 2.3.1).

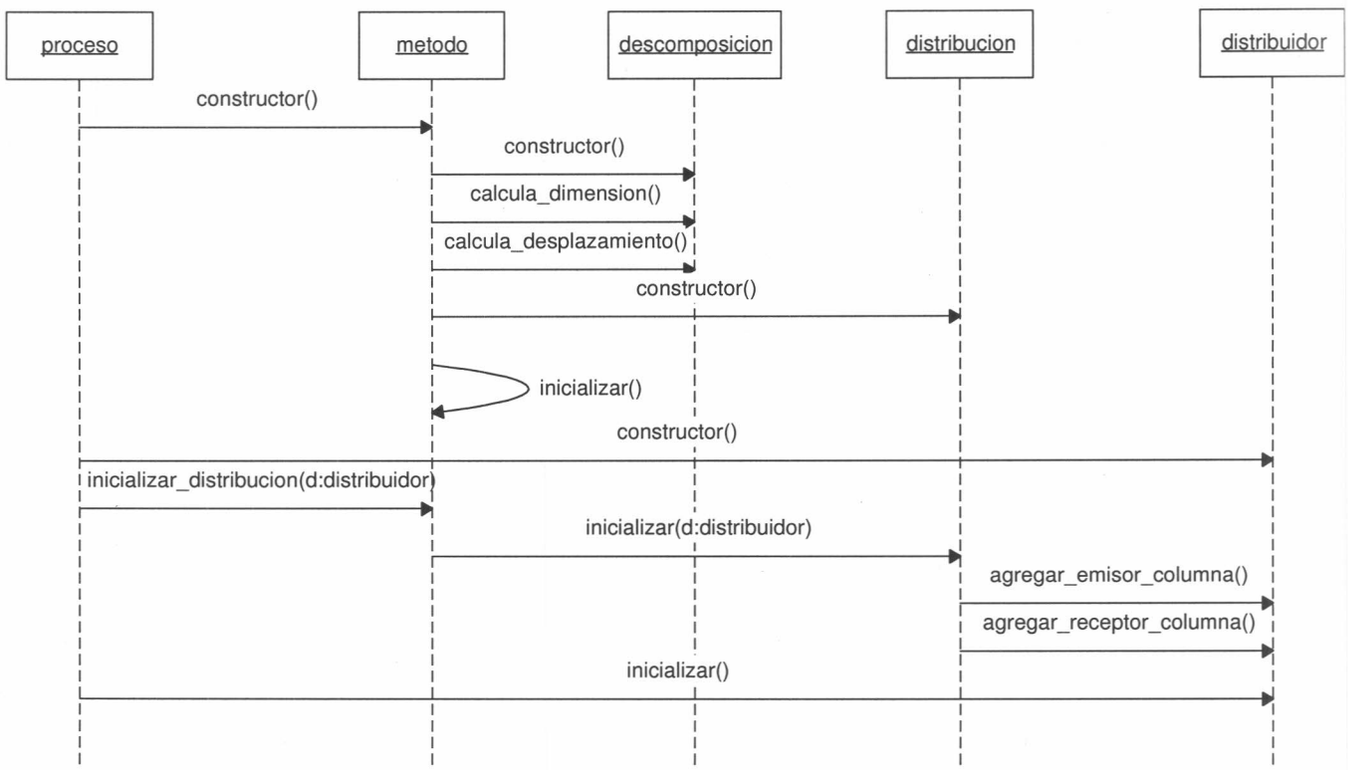


Figura B.2: Secuencia de inicialización del algoritmo paralelo (en una versión simplificada de una descomposición unidimensional).

B.3.1 Inicialización del programa

En esta sección se explica el procedimiento de inicialización del algoritmo paralelo. En la fig. B.2 se observa la secuencia de pasos que realiza el programa una vez que se leen los parámetros del usuario (tamaño de la grilla a utilizar, números adimensionales, cantidad de procesadores a utilizar, estrategia de descomposición y asignación de dominios, y estrategia de intercambio de datos).

Según hemos explicado, el programa dispone de una instancia de la clase *proceso* que se encarga de coordinar la ejecución de la simulación secuencial. Durante su inicialización, crea las instancias de cada uno de los *metodos* (en la fig. B.2 se presenta la inicialización de sólo un método porque el procedimiento es idéntico en el resto de los casos) que componen el sistema de ecuaciones.

Cada uno de los *metodos* debe, a su vez, reservar espacio en memoria para el dominio sobre el que le corresponda trabajar. Para esto, obtiene un objeto de una clase

derivada de *descomposicion* (determinada por el usuario al seleccionar la estrategia en los parámetros del programa), que se encarga de calcular el dominio del procesador al invocarse la función `calcula_dimension()`[7]. Luego procede a inicializar el dominio asignado al ejecutar `inicializar()`, función que provee el usuario y que establece el valor inicial en cada una de las posiciones de la grilla.

El último paso, previo a la ejecución del ciclo en que se realizan los cálculos correspondientes a la simulación, consiste en configurar una instancia de una clase derivada *distribuidor* (nuevamente, seleccionada de acuerdo a la estrategia elegida por el usuario), que se encargará de sincronizar los datos entre los dominios de cada procesador al finalizar cada iteración de cálculo.

Una vez que *proceso* obtiene el *distribuidor*, lo pasa a cada uno de los *metodos* invocando la función `inicializar_distribucion()`, que delega el trabajo a la *distribucion* que obtuvo anteriormente. La *distribucion*, finalmente, configura las operaciones de transmisión y recepción de filas/columnas que se debe realizar para sincronizar los datos en cada iteración.

Debe notarse que, al separar la funcionalidad de *distribuidor* y *distribucion*, por la cual el primero sólo se encarga de llevar a cabo las operaciones que configura el segundo, podemos combinar las distintas variantes de descomposición implementadas (en las clases derivadas de *descomposicion* y *distribucion*) con las estrategias de intercambio de datos (en las clases derivadas de *distribuidor*), sin tener que codificar cada una de ellas en forma explícita. De esta manera, se simplifica y agiliza la posibilidad de implementar nuevas variantes, tanto de descomposición como de intercambio de datos.

Terminando el proceso de inicialización, *proceso* llama a la función `inicializar()` de su *distribuidor*, para que, si los necesita (como en el caso de la estrategia de intercambio de datos con mensaje único por procesador), reserve los buffers que utilizará luego.

[7]Como se puede observar en la secuencia, también se llama a la función `calcula_desplazamiento()`, que permite a cada *metodo* determinar la ubicación del dominio asignado en la grilla global virtual, que componen todos los procesadores. Esta información es utilizada, por ejemplo, para detectar los bordes de la grilla global.