UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

DEPARTAMENTO DE SEÑALES Y COMUNICACIONES

TESIS DOCTORAL

MIDDLEWARE DE EQUILIBRIO DE CARGA PARA COMPUTACIÓN PARALELA SOBRE ENTORNOS HETEROGÉNEOS LAN-WLAN

David de la Cruz Sánchez Rodríguez

Julio 2006

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Departamento: SEÑALES Y COMUNICACIONES

Programa de Doctorado: TECNOLOGÍAS DE LAS TELECOMUNICACIONES

Título de la Tesis Doctoral

MIDDLEWARE DE EQUILIBRIO DE CARGA PARA COMPUTACIÓN PARALELA SOBRE ENTORNOS HETEROGÉNEOS LAN-WLAN

Tesis Doctoral presentada por D. DAVID DE LA CRUZ SÁNCHEZ RODRÍGUEZ

Dirigida por la Dra. Dª. ELSA Mª MACÍAS LÓPEZ

La Directora, **El Doctorando,**

Las Palmas de Gran Canaria, a 18 de julio de 2006

A África y a Adexe

Agradecimientos

Cuando cada día lo das todo por tu hijo, y sientes que tus padres hacen lo mismo por tí, lo menos que puedes hacer es decirles, una vez más: gracias papá, gracias mamá.

A mi familia, desde la que siempre ofrece *el ciento por uno* hasta la más pequeñita; sin ellos no sería lo mismo.

A mi segunda familia, por acogerme como uno más.

A Elsa y a Álvaro por impulsar la realización de este trabajo.

A todos los que han compartido un segundo de su tiempo conmigo, y especialmente a Matías, África, Víctor y Alejandro.

Este trabajo ha sido parcialmente subvencionado por el Ministerio de Educación, Cultura y Deporte y el Fondo Europeo de Desarrollo Regional (FEDER) a través del proyecto TSI2005-07764-C02-01: *"Aplicaciones multimedia antitabáquica sobre dispositivos inalámbricos y web en psicología clínica"*; por la Consejería de Educación, Cultura y Deportes y el FEDER a través del proyecto PI042004/164: *"Control eficiente de las desconexiones en redes inalámbricas para streaming de video y su aplicación a una empresa"*; y por la Fundación Universitaria de Las Palmas, Lopesán S.A. y Unelco S.A. a través del programa INNOVA: *"Diseño de un middleware para la programación eficiente de entornos de comunicación y computación heterogéneos"*.

Resumen

Las redes inalámbricas de área local basadas en el estándar IEEE 802.11 han tenido un auge espectacular en los últimos años, proporcionando nuevos servicios como: la movilidad de usuarios, la localización en interiores, la gestión de recursos sin cables, etc. Este desarrollo ha demostrado que una de sus posibilidades de uso es la computación paralela-distribuida. En concreto, en un trabajo previo se ha demostrado que un entorno de computación LAN-WLAN puede ser usado para ejecutar aplicaciones que siguen el paradigma de programación paralela Maestro/Esclavo, en el cual los cálculos paralelos son realizados en cada iteración y existen dependencias de datos entre iteraciones (limitando por tanto el paralelismo).

En este tipo de entornos, la heterogeneidad presente a nivel de computación y de comunicación obliga a implantar algún mecanismo de equilibrio de carga para llevar a cabo la ejecución eficiente del tipo de aplicaciones mencionadas. En cualquier otro caso, los procesos del programa paralelo-distribuido permanecerían en un estado ocioso durante unidades de tiempo muy dispares, degradando así el tiempo de ejecución de la aplicación paralela-distribuida. Por tanto, el principal objetivo de esta tesis doctoral es diseñar una estrategia de equilibrio de carga para entornos LAN-WLAN no dedicados donde el número de computadores portátiles puede variar en tiempo de ejecución.

Debido a las características del entorno, al rendimiento aleatorio del canal inalámbrico, a la variación dinámica y aleatoria de los recursos de computación portátiles y al recurso finito de energía de sus baterías, es necesario obtener información en tiempo real del estado de los computadores y de las redes de comunicación para estimar de forma eficiente la distribución de carga. En este sentido, otro objetivo de este trabajo es desarrollar una arquitectura *software* ligera y transparente al programador que sea capaz de monitorizar el entorno, y proveer los datos recopilados a la estrategia de equilibrio de carga.

Para cumplir estos objetivos se diseñó un protocolo de equilibrio de carga, y en base a dicho protocolo se ha implantado un *middleware* de cinco funciones que oculta su complejidad y facilita la programación de aplicaciones paralelas-distribuidas que aprovechan eficientemente todos los recursos presentes en el entorno LAN-WLAN.

Abstract

In the last years, wireless local area networks based on the standard IEEE 802.11 have had spectacular peak, providing new services as: mobility of users, indoors location, wireless management of resources, etc. This development has demonstrated that one of its possibilities is the parallel-distributed computing. In particular, in a previous work has been demonstrated that a LAN-WLAN computing environment can be used to efficiently execute applications that follow the Master/Slave parallel programming paradigm, where the parallel calculations are done in each iteration and there are data dependencies between iterations (therefore the parallelism is limited).

In this kind of heterogeneous computing environments is neccesary to implement some mechanism of load balancing to carry out the efficient execution for the type of mentioned applications. In any other case, the processes of the parallel-distributed program will remain in idle state during some time units, and therefore the execution time of the parallel-distributed application will be increased. In this sense, the main aim of this PhD is to design a strategy of load balancing for non-dedicated LAN-WLAN environments where the number of portable computers can vary at run time.

Due to the environment characteristics, the random performance of the wireless channel, the dynamic and random behaviour of the portable computers and the finite energy stored in their batteries, it is necessary to obtain information in real time of the performance of computers and networks to estimate in an efficient way the load distribution. In this sense, another aim of this research work is to develop a light weight software architecture to monitor the environment, and to provide the data gathered to the strategy of load balancing.

In order to carry out these objectives a protocol of load balancing was designed, and based on it we have implemented a middleware of five functions which hides the protocol complexity and makes easy the programming of parallel-distributed applications that efficiently take advantage of all resources of the LAN-WLAN environment.

Glosario

ATM	*Asynchronous Transfer Mode*
BER	*Bit Error Rate*
CF	*Computador Fijo*
CP	*Computador Portátil*
DCF	*Distribution Coordination Function*
EF	*Entidad Esclava Fija*
EM	*Entidad Maestra*
EP	*Entidad Esclava Portátil*
HNOW	*Heterogeneous Network of Workstations*
HTTP	*HyperText Transfer Protocol*
IAB	*Internet Architecture Board*
IANA	*Internet Assigned Numbers Authority*
IP	*Internet Protocol*
IPC	*Interprocess Communication*
LAN	*Local Area Network*
LAM	*Local Area Multicomputer*

MAC	*Medium Access Control*
MIB	*Management Information Base*
MIPS	*Million Instructions Per Second*
MPI	*Message Passing Interface*
MPMD	*Multiple Program Multiple Data*
NA	*Número de procesos en computadores fijos*
NI	*Número de procesos en computadores portátiles*
NWS	*Network Weather Service*
PAN	*Personal Area Network*
PDU	*Protocol Data Unit*
PVM	*Parallel Virtual Machine*
QoS	*Quality of Service*
RPC	*Remote Procedure Call*
RSS	*Received Signal Strength*
SAMR	*Structured Adaptive Mesh Refinement*
SNMP	*Simple Network Management Protocol*
SPMD	*Single Program Multiple Data*
SVM	*Support Vector Machine*
TCP	*Transport Control Protocol*
TRAP	*Notificación enviada desde un agente SNMP a un gestor*
UDP	*User Datagram Protocol*
VHDL	*Very high speed integrated circuit Hardware Description Language*
VM_LAM	*Máquina Virtual del entorno de computación LAM*
WLAN	*Wireless LAN*

Índice

1. Introducción

En este capítulo se presenta una introducción a los retos actuales para la ejecución eficiente de aplicaciones paralelas-distribuidas sobre redes LAN-WLAN de computadores heterogéneos donde el número de éstos pertenecientes a la WLAN puede variar de forma dinámica y en tiempo de ejecución. También, se analizan las motivaciones que han sugerido la realización de este trabajo de investigación, así como los objetivos y aportaciones de esta tesis doctoral. Por último, se presenta la estructura de esta memoria.

1.1 Antecedentes

La computación paralela y distribuida sobre redes de computadores heterogéneos es un área de investigación que ha sido desarrollado durante los últimos años [SuG99]. Además, el desarrollo de computadores de altas prestaciones y de redes de comunicación con rendimiento elevado permite la combinación de redes heterogéneas de estaciones de trabajo para ser utilizadas como una máquina paralela virtual [DaP97]. Por otro lado, la proliferación de computadores portátiles y el auge espectacular de las redes inalámbricas de área local posibilita la combinación de redes LAN-WLAN para realizar computación paralela-distribuida. De hecho, en [Mac01] se demuestra que es factible la computación paralela llevada a cabo en arquitecturas que combinan segmentos de red cableados e inalámbricos obteniéndose buenos resultados en la ejecución de aplicaciones paralelas que siguen el paradigma Maestro/Esclavo [BLR03] con dependencias de datos entre iteraciones.

En [Zom02] se afirma que es un reconocido hecho que hay un solape entre los fundamentos de los entornos móviles y las máquinas paralelas. Esta situación puede ser utilizada para abrir nuevas líneas de investigación en el diseño de algoritmos paralelos que pueden resolver problemas en computación móvil. Por tanto, la movilidad trae consigo una completa gama de nuevos desafíos que necesitan ser tenidos en cuenta para desarrollar nuevos y eficientes programas paralelos-distribuidos.

Para llevar a buen término la ejecución eficiente de aplicaciones paralelas en un entorno de computación LAN-WLAN hay que tener en cuenta por un lado, el rendimiento de los recursos de computación y su comportamiento aleatorio y dinámico en cuanto a movilidad se refiere, y por otro lado, la heterogeneidad del sistema de interconexión. Estas situaciones introducen desequilibrios en el entorno de computación, tanto en computación como en comunicación, afectando a los tiempos de ejecución de las tareas y al rendimiento global del sistema [StS04]. Para mejorar el rendimiento de las aplicaciones paralelas cuando éstas son ejecutadas en entornos heterogéneos es necesario utilizar técnicas de equilibrio de carga, que asignen trabajo a cada recurso de computación de manera que éste sea proporcional al rendimiento del mismo y al

segmento de red que lo conecta al resto de computadores [NiS93], evitando así ciclos ociosos de CPU en los computadores más rápidos y minimizando el tiempo de ejecución de las mismas. Precisamente, una de las líneas futuras de investigación planteadas en [Mac01] es dotar al *middleware* desarrollado en dicho trabajo, de nuevas funcionalidades para implementar un mecanismo de equilibrio de carga que reduzca los tiempos ociosos de los procesos esclavos que cooperan en la ejecución de una aplicación paralela en un entorno de computación LAN-WLAN. Sin embargo, debido al comportamiento aleatorio y no predeterminado de los recursos portátiles que conforman el segmento de red WLAN, dichas técnicas también deben controlar su disponibilidad para evitar la implementación de envíos y recepciones de datos innecesarios.

Las técnicas de equilibrio de carga han sido estudiadas ampliamente sobre entornos de computación homogéneos [ELZ86][WiR93]. Sin embargo, estos esquemas no pueden ser aplicados directamente sobre una red de computadores debido a la gran cantidad de nuevos desafíos que éstas presentan: comunicaciones lentas, latencias altas, arquitecturas de procesadores diferentes, segmentos de red con diferentes velocidades de transmisión, movilidad de los computadores, *jitter* variable, etc. De hecho, en [FoK99] se afirma que el equilibrio de carga en los sistemas heterogéneos es más complejo debido a que tiene que ser considerada la potencia computacional de cada recurso, así como el rendimiento de los enlaces de comunicación entre ellos. Todo ello implica que se tengan que desarrollar técnicas para realizar una predicción efectiva sobre el rendimiento de los recursos de computación. En este sentido, en [BSM01] se especifica que el equilibrio de carga es una línea de investigación abierta *(hot topic)* en computación heterogénea, y ello ha contribuido a que en los últimos años varios autores hayan propuesto técnicas o estrategias para mejorar el tiempo de ejecución de aplicaciones paralelas en entornos de computación heterogéneos. Algunos trabajos, como [TQD01][EgE02][BLR03], diseñan estrategias de equilibrio de carga considerando solamente la heterogeneidad a nivel de computación y no de la red de comunicación. Incluso, [EgE02] se desmarca de los otros autores porque su mecanismo está diseñado para entornos no dedicados. Sin embargo, las técnicas desarrolladas en estos trabajos presentarán sus deficiencias cuando sean

implementadas en entornos de comunicación heterogéneos con variación significante y aleatoria de rendimiento, como ocurre en una red inalámbrica. Por otro lado, la propagación de la señal inalámbrica se ve afectada por numerosas condiciones medioambientales [KaK04], y por tanto, la utilización de esquemas que parametrizan el canal inalámbrico antes de la ejecución de las aplicaciones, como se realiza en [DHE05], sólo pueden ser aplicados en situaciones estacionarias ambientales y de movilidad.

En favor de este trabajo de investigación, se desconoce la existencia de trabajos que estudien el equilibrio de carga en entornos de computación LAN-WLAN no dedicados donde los computadores portátiles pueden desplazarse de forma aleatoria y vincularse/ desvincularse al entorno de computación en tiempo de ejecución.

1.2 Motivaciones

En [Mac01] se desarrolló un *middleware*, denominado *LAMGAC*, que permite la gestión de la variación dinámica de los recursos de computación en un entorno LAN-WLAN en tiempo de ejecución. Una de las líneas futuras planteadas en este trabajo es el equilibrio de carga en este tipo de entornos de computación. En este sentido, nuestra primera motivación es desarrollar un mecanismo de equilibrio dinámico de carga que se adapte a la variación dinámica del número de recursos de computación, así como al rendimiento variable de los computadores y de la red de comunicación de un entorno no dedicado. Este mecanismo está orientado a la ejecución de aplicaciones que siguen el paradigma Maestro/Esclavo [BLR03], donde los cálculos paralelos son realizados en cada iteración y pueden existir dependencias de datos entre iteraciones. Este mecanismo debe ser integrado en el *middleware LAMGAC*.

En un entorno de computación tan variable como puede ser un entorno LAN-WLAN es necesario monitorizar diversos aspectos acerca del rendimiento de los computadores y la red, como pueden ser: la carga del sistema, la velocidad del procesador, el *throughput* y latencia de red, la disponibilidad de los recursos portátiles, la energía remanente en la batería de éstos, etc. Todos estos aspectos tienen que ser considerados por el mecanismo de equilibrio de carga para realizar una distribución óptima de datos. En este sentido, la

segunda motivación de este trabajo de investigación es desarrollar una arquitectura que sea capaz de monitorizar los aspectos mencionados y anticiparlos al mecanismo de equilibrio de carga para estimar la mejor distribución de datos posible. Debido a que esta arquitectura está en funcionamiento mientras se ejecuta la aplicación paralela, es decir, existe un solapamiento entre la ejecución de la aplicación paralela y la monitorización de los recursos, el diseño de la misma debe ser lo más eficiente posible en el sentido de que no debe sobrecargar a los recursos de computación ni a la red de comunicación. Por este motivo, la arquitectura de monitorización está basada en el protocolo de gestión de red SNMP el cual es un protocolo ligero a nivel de aplicación y está diseñado para monitorizar y ofrecer información de rendimiento de los recursos de la red de una forma sencilla y con la mínima sobrecarga.

Por último, debido al comportamiento dinámico de los computadores portátiles en cuanto a la movilidad se refiere, y a que éstos suelen estar alimentados por un recurso finito como es la batería, se está interesado en controlar aquellos dispositivos con un nivel bajo de energía o que estén situados en una zona de cobertura limitada. Este control nos permite evitar el envío y la recepción de datos sobre procesos que se ejecutan en computadores que no están disponibles en dicho instante, evitando así desequilibrios en los tiempos de ejecución del resto de procesos y transferencias de datos innecesarias.

1.3 Objetivos y aportaciones de la tesis

El objetivo principal de este trabajo es demostrar que es posible llevar a cabo el equilibrio de carga sobre aplicaciones paralelas en un entorno de computación heterogéneo y cambiante como puede ser un entorno basado en una red LAN-WLAN. Para ello, se ha desarrollado un mecanismo de equilibrio dinámico de carga que tiene en cuenta aspectos de rendimiento de los computadores y de la red que los conecta, así como la variación dinámica del número de computadores portátiles.

El segundo objetivo es demostrar que es posible solapar los cálculos paralelos con el control y monitorización dinámica de los recursos, sin que el tiempo de ejecución de la aplicación paralela-distribuida se vea perjudicada considerablemente.

Por último, se pretende demostrar que es necesario controlar los aspectos relacionados con la cobertura y la batería de los recursos portátiles para evitar desequilibrios en el resto de recursos del entorno de computación.

Para llevar a cabo los objetivos anteriores se han realizado las siguientes aportaciones:

- *Equilibrio de carga en entornos heterógeneos.* Se ha diseñado un protocolo de equilibrio de carga que tiene en cuenta la variación dinámica del número de recursos de computación. Para ello se ha utilizado una estrategia dinámica basada en el rendimiento de los computadores y de las comunicaciones [SMS03c][SMS02]. Tomando como base esta estrategia se ha ampliado la biblioteca desarrollada en [Mac01] con cinco nuevas funciones para realizar distribuciones de datos equilibradas [SMS04]. Las pruebas realizadas demuestran que la sobrecarga del tiempo de ejecución de nuestra biblioteca es mínimo, y a su vez, mejora el tiempo de ejecución de las aplicaciones paralelas.

- *Monitorización transparente de los recursos de computación.* La estrategia de equilibrio de carga implantada necesita información de rendimiento de los computadores y de la red de comunicación. Para obtener esta información se ha desarrollado una arquitectura basada en SNMP que se encarga de monitorizar ciertos parámetros del entorno y proveérselos al mecanismo de equilibrio de carga [SMS03b]. Esta arquitectura se ejecuta de forma solapada con la aplicación paralela-distribuida, siendo transparente al usuario, es decir, el programador no tiene acceso a ella y no tiene por qué conocer su funcionamiento.

- *Control de los recursos de computación portátiles.* Por regla general, los computadores portátiles tienen un comportamiento dinámico y aleatorio. En este sentido, en un instante dado un computador puede situarse fuera de cobertura del punto de acceso y, por tanto, no podrá enviar ni recibir datos. Además, debido a que éstos suelen estar alimentados por una batería, su nivel puede agotarse durante el transcurso de una iteración sin poder enviar los

resultados calculados al proceso maestro. Por ello, en este trabajo se desarrolló un mecanismo basado en SNMP para controlar estos aspectos de los recusos portátiles [SMS03a][SMS05a]. Los resultados de este mecanismo son accesibles por la estrategia de equilibrio de carga para ser considerados en la siguiente distribución de datos. Por otro lado, esta información también es accesible al usuario a través de las funciones de la biblioteca para que pueda implantar un esquema de recepción controlada de datos como el presentado en [SMS05b]. Este esquema es muy importante, dado que evita la espera prolongada por datos que se sabe a priori que no van a llegar.

1.4 Estructura de la memoria

El trabajo de investigación que se presenta en esta memoria se divide en seis capítulos.

En el capítulo dos se introduce la arquitectura *hardware* y *software* del entorno de computación LAN-WLAN utilizado en este trabajo, el estado del arte sobre los mecanismos de equilibrio de carga teniendo en cuenta la calidad del enlace inalámbrico y el nivel de batería en los computadores portátiles, y la obtención de información dinámica de red para estimar adecuadamente la distribución de carga. Por último, se presentan las ideas generales del trabajo realizado.

En el capítulo 3 se exponen las primitivas del protocolo de equilibrio de carga diseñadas en este trabajo de investigación. También, se explica la arquitectura SNMP utilizada para la obtención de información dinámica de rendimiento y del estado de los computadores y la red de comunicación.

En el capítulo 4 se presenta la biblioteca de equilibrio de carga que implementa las primitivas diseñadas en el capítulo anterior. Además, utilizando dicha biblioteca se propone un esquema de recepción controlada de datos. También, se presenta el modelo de programación a seguir con la biblioteca desarrollada, así como su aplicación a la solución de un problema del álgebra lineal. Por último, se comentan las medidas realizadas para evaluar la eficiencia del protocolo desarrollado.

En el capítulo 5 se presentan los resultados experimentales obtenidos al aplicar la biblioteca diseñada en la ejecución de problemas reales de ingeniería de telecomunicación y del álgebra lineal.

En el capítulo 6 se presentan las conclusiones de este trabajo, así como las líneas futuras de trabajo a seguir.

Finalmente, se expone la bibliografía consultada.

2. Equilibrio de carga en entornos de computación LAN-WLAN

En este capítulo, en primer lugar se revisa la arquitectura *hardware* y *software* del entorno de computación heterogéneo LAN-WLAN utilizado para ejecutar aplicaciones paralelas-distribuidas. A continuación, y debido a la heterogeneidad presente en dicho entorno, tanto a nivel de computación como de comunicación, se estudian las estrategias de equilibrio de carga para minimizar los tiempos de ejecución de las aplicaciones paralelas-distribuidas que se desarrollan en estos entornos. También, se analiza el protocolo simple de gestión de red que se utiliza para proporcionar la información de rendimiento del entorno de computación a la técnica de equilibrio de carga. Por último, se plantean las ideas generales del esquema de equilibrio de carga desarrollado en este trabajo.

2.1 Antecedentes

En este trabajo de investigación se parte de la arquitectura LAN-WLAN propuesta en [Mac01], donde los computadores con conexión de red cableada se consideran fijos y nunca abandonan el entorno de computación mientras se ejecute la aplicación paralela. Por otro lado, los computadores con conexión de red inalámbrica pueden abandonar o incorporarse al entorno de computación en tiempo de ejecución. En la figura 2.1 se muestra un ejemplo de la arquitectura *hardware* de un entorno de computación LAN-WLAN. La topología de la red cableada está formada por recursos que utilizan la familia de estándares de comunicación IEEE 802.3. La topología de la red inalámbrica está formada por recursos que utilizan la familia de estándares de comunicación IEEE 802.11. La conexión entre ambas redes se realiza a través de un punto de acceso que implementa los dos estándares de red mencionados.

La arquitectura propuesta para ejecutar aplicaciones paralelas en el entorno de computación LAN-WLAN utiliza una arquitectura *software* como la mostrada en la figura 2.2. En [Mac01] se desarrolló un *middleware*, denominado *LAMGAC*, que permite gestionar la expansión dinámica y en tiempo de ejecución de nuevos procesos esclavos en los computadores portátiles que entran en la WLAN. De la misma forma, este *middleware* permite la desvinculación de aquel proceso que se ejecuta en un computador portátil que desea abandonar el entorno de computación [MSO01].

El modelo de arquitectura presentado y el diseño del *middleware LAMGAC* están validados para la implementación de aplicaciones que siguen el paradigma Maestro/ Esclavo. Dentro de este paradigma, la ejecución de aplicaciones paralelas que desarrollan los cálculos paralelos en cada iteración, y poseen dependencias de datos entre iteraciones resultan muy interesantes cuando son implementadas en un entorno LAN-WLAN debido a: la sincronización de datos existente en cada iteración, la heterogeniedad presente tanto en computación como en comunicación y a la variación dinámica del número de computadores. Estos motivos implican que para lograr una ejecución eficiente se tengan que diseñar mecanismos de equilibrio de carga efectivos.

Figura 2.1 Arquitectura hardware del entorno de computación LAN-WLAN

En la figura 2.3 se muestra un esquema de dependencia de datos entre iteraciones de una aplicación donde los cálculos paralelos son realizados dentro de cada iteración. Además, se muestra la expansión dinámica de un nuevo proceso. La expansión de este proceso debe realizarse de forma controlada debido a las dependencias existentes en cada iteración, y por tanto, debe ser tenido en cuenta en la distribución de datos.

A continuación detallamos algunos de los aspectos más relevantes de cada uno de los niveles que forman la arquitectura *software* utilizada.

Nivel físico

Como se puede apreciar en la figura 2.1, en el entorno de computación LAN-WLAN hay comunicaciones cableadas e inálambricas, es decir, se utiliza el cable como medio de comunicación de los computadores fijos, y el aire como medio de transmisión en la comunicación de los computadores portátiles. Por tanto, debido a las diferentes características de propagación presentes en cada medio, el rendimiento de uno con respecto al otro difiere en cuanto a la transmisión de datos, es decir, la arquitectura LAN-WLAN es heterogénea en el nivel físico. La diferencia de rendimiento se debe principalmente a que la red inalámbrica posee una tasa mayor de errores, la congestión

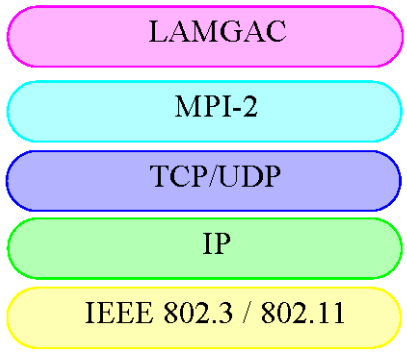


Figura 2.2 Arquitectura software del entorno de computación LAN-WLAN

en el canal no se diferencia de la pérdida de paquetes, el ancho de banda de transmisión se ve afectado por obstáculos físicos y la presencia de personas, la banda de frecuencias de operación no está regulada con lo que el despliegue de otras redes inalámbricas pueden interferir en la red de trabajo, etc.

Por otro lado, el estándar IEEE 802.11 especifica dos tipos de tecnologías utilizadas en la capa física de las redes inalámbricas: espectro ensanchado (RF) e infrarrojos. A nivel físico, en el entorno de computación LAN-WLAN utilizamos la tecnología de espectro ensanchado, debido a que la tecnología de infrarrojos presenta claras limitaciones, como son la baja velocidad de transmisión, rango de alcance reducido, dependencia con la orientación del emisor y del receptor, y además, sólo está definida en el estándar IEEE 802.11, con poca disponibilidad en el mercado, y no en el resto de especificaciones del estándar, como son el IEEE 802.11a/b/g.

Nivel de enlace

A nivel de enlace se utiliza la familia de estándares de comunicación IEEE 802.3 para la red cableada e IEEE 802.11 para la red inalámbrica. Por tanto, a nivel de enlace también existe heterogeneidad con respecto a los protocolos utilizados.

Existen otros protocolos de comunicación inalámbrica como Bluetooth [Web-1] e HiperLAN/2 [Web-2], pero se ha desestimado su utilización en el entorno LAN-WLAN por presentar limitaciones frente a la familia IEEE 802.11. Bluetooth posee una tasa de

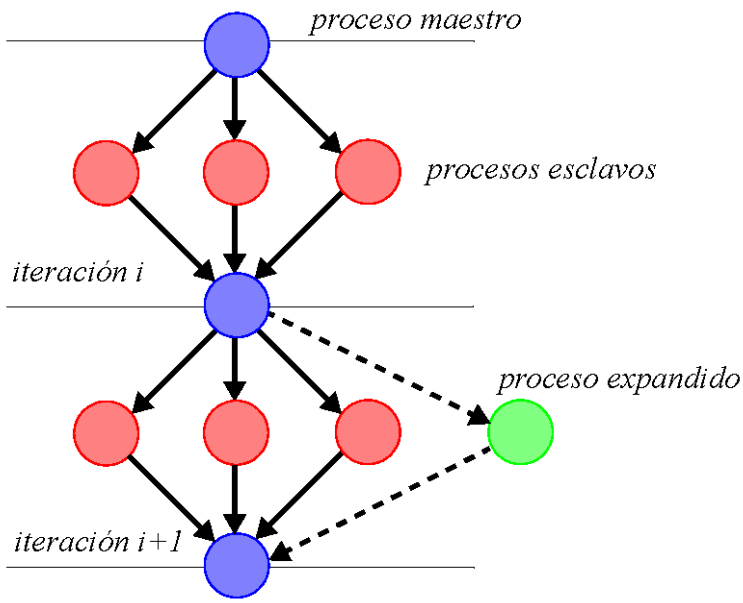


Figura 2.3 Dependencia de datos entre iteraciones

transferencia baja y un rango de alcance reducido, además de estar pensado para redes de área personal (PAN) [FaA00]. Por otro lado, aunque HiperLAN/2 puede proporcionar una tasa de transferencia de datos de hasta 54 Mbps, igual que IEEE 802.11a/g, no ha llegado a implantarse de forma definitiva debido a su coste elevado, y al auge e implantación rápida del estándar IEEE 802.11.

Nivel de red

El protocolo IP se ha convertido en el estándar de facto de la capa de red, siendo utilizado en Internet y en la mayoría de las redes de área local.

Nivel de transporte

Al igual que el protocolo IP, el protocolo TCP se ha convertido en el estándar de facto para comunicaciones fiables en la capa de transporte. El protocolo TCP fue diseñado originalmente para redes donde la BER es reducida, y por tanto, la pérdida de paquetes se debe principalmente a la congestión. Por tanto, su rendimiento en las redes inalámbricas es pobre, debido a la alta BER, la elevada congestión y los *handoffs* producidos por la movilidad de los usuarios. Aunque recientemente han aparecido

nuevas versiones de TCP, como TCP Veno [FLL03] que mejora el rendimiento del protocolo TCP en redes inalámbricas, Freeze-TCP [GMP00] diseñado para considerar la movilidad de los usuarios, o TCP-Vegas [BrP95] que ha sido incorporada recientemente a la versión oficial del kernel de Linux 2.6.6, nosotros no vamos a tener en cuenta el medio de transmisión ni el comportamiento de los usuarios para utilizar el protocolo TCP más adecuado. Por tanto, independientemente del tipo de interconexión presente en el entorno de computación, nosotros vamos a utilizar siempre la misma implementación del protocolo TCP.

Por otro lado, también se utiliza el protocolo de transporte no fiable UDP cuando sea requerido por aplicaciones que deseen ofrecer el mínimo impacto sobre la red. UDP proporciona una mejora sustancial sobre las WLAN en cuanto al rendimiento se refiere, respecto a los protocolos de transporte fiables como TCP [AAM01]. Esto se debe a que UDP no establece control de flujo, secuenciamiento de paquetes, corrección de errores, retransmisiones y reconocimientos.

Nivel de middleware

En el entorno de computación propuesto en [Mac01] se utiliza la biblioteca de paso de mensajes MPI-2 [GLT99] para desarrollar aplicaciones paralelas, y el *middleware LAMGAC* para controlar y gestionar la variación del número de procesos en los computadores portátiles.

Aunque la biblioteca PVM [Sun90] fue la primera que se diseñó para realizar computación paralela-distribuida en entornos heterogéneos, el estándar MPI-2 ha sido adoptado por la mayoría de los centros de investigación ya que proporciona un conjunto amplio de funciones para realizar comunicaciones punto a punto y colectivas, y además permite, al igual que PVM, la creación y expansión dinámica de procesos. Se ha elegido MPI por las siguentes razones: MPI tiene un mayor número de implementaciones de libre distribución, tiene comunicación asíncrona que permiten solapar las comunicaciones con los cálculos, tiene grupos de comunicación sólidos, eficientes y deterministas, maneja eficientemente los *buffers* de mensajes, otros entornos software de comunicación se pueden combinar con él sin problemas de sincronización, maneja

eficientemente los computadores paralelos y los *clusters*, es totalmente portable, ya que está definido formalmente y es un estándar de facto.

Por otro lado, se utiliza el *middleware LAMGAC*, ya que ningún otro middleware diseñado para la computación heterogénea, como pueder ser Globus Toolkit [Fok97], se adapta mejor a los entornos de computación LAN-WLAN, puesto que *LAMGAC* ha sido diseñado específicamente para ser utilizado en entornos donde el número de computadores puede variar en tiempo de ejecución.

2.2 Equilibrio de carga

En los últimos años, los avances tecnológicos han dado lugar al desarrollo de computadores de altas prestaciones y de redes de comunicación con rendimiento elevado. La combinación de redes heterogéneas de estaciones de trabajo que son utilizadas como una máquina paralela virtual ha llegado a ser una alternativa viable a las tradicionales máquinas paralelas dedicadas, y actualmente, muchos grupos de investigación utilizan estos entornos para ejecutar aplicaciones científicas paralelas. Sin embargo, hay ciertas diferencias entre los dos tipos de máquinas que necesitan ser consideradas [DaP97]:

- *Tipo de computadores*. Normalmente, un sistema multiprocesador está formado por un conjunto de procesadores de iguales características, mientras que los computadores que forman un entorno de computación distribuido suelen tener diferencias en la arquitectura *hardware* y *software*, como pueden ser: el sistema operativo, la velocidad del procesador, la memoria y el espacio de almacenamiento físico.
- *Carga del sistema*. En una máquina paralela dedicada, la carga del sistema se puede predecir con cierta certeza, y puede ser controlada por un planificador dedicado. Sin embargo, la carga en cada computador de un entorno de computación distribuido y heterogéneo puede variar en cada instante debido a la intervención de otros usuarios y la ejecución de otras aplicaciones. Por tanto, la carga del sistema no puede ser estimada con antelación.

- *Red de interconexión.* La red de comunicación en las máquinas paralelas suele estar implementada como una topología fija, y diseñada específicamente para obtener un rendimiento elevado. Un ejemplo de este tipo de red de interconexión es una red hipercubo o toro, en las cuales se utilizan técnicas de conmutación que proporcionan una interconexión ultra rápida [AbM88]. Por otro lado, las redes de computadores que utilizan estándares de red de área local, como IEEE 802.3 e IEEE 802.11, tienen latencias elevadas y anchos de banda reducidos comparado con los valores obtenidos en la red de interconexión de una máquina paralela. A esto hay que añadir que en una LAN pueden existir segmentos de red con diferentes características de transmisión y diferente rendimiento, como puede ser el entorno de computación LAN-WLAN utilizado en este trabajo de investigación.

- *Fallos en el entorno.* Una máquina paralela dedicada, donde la arquitectura está diseñada específicamente para obtener un rendimiento elevado, tiene una fiabilidad alta. Sin embargo, en las redes de computadores hay que tener en consideración la tolerancia y detección de fallos debido a: las diferentes versiones *hardware* y *software* instaladas en cada máquina, las cuales pueden provocar una disminución del rendimiento del recurso si no están bien sintonizadas, al rendimiento del medio transmisión el cual puede variar de forma aleatoria debido a la tasa de error elevada y a la congestión, y a otros factores sujetos a las características de los dispositivos inalámbricos, como son la movilidad y la dependencia con su batería.

Estas consideraciones ponen en entredicho la viabilidad de los entornos de computación distribuidos como una alternativa a las máquinas paralelas dedicadas. Sin embargo, un estudio realizado en el proyecto Condor [LLM88] demuestra que los computadores en red de una empresa típica están en un estado ocioso o *idle* el 80% del tiempo. Además, en dicho estudio también se afirma que cuando los usuarios están ejecutando aplicaciones interactivas, un número importante de ciclos de CPU no son utilizados, los cuales podrían haber sido empleados para un propósito específico. La

disponibilidad de ciclos libres de CPU como también la mejor relación rendimiento-precio han sido los factores decisivos para el crecimiento rápido de la computación paralela en redes de computadores.

De esta forma, cuando se utiliza una red de computadores para ejecutar aplicaciones paralelas, como puede ser una combinación LAN-WLAN, hay que tener en cuenta por un lado, el comportamiento dinámico y el rendimiento de los recursos de computación y, por otro lado, la heterogeneidad del sistema de interconexión. Estas consideraciones introducen desequilibrios en el entorno, tanto en computación como en comunicación, afectando a los tiempos de ejecución de las tareas y al rendimiento global del sistema [StS04]. Por tanto, es necesario utilizar técnicas de equilibrio de carga que mejoren el rendimiento de las aplicaciones paralelas-distribuidas cuando éstas son ejecutadas en entornos heterogéneos, evitando ciclos ociosos de CPU en los computadores más rápidos y minimizando su tiempo de ejecución.

El equilibrio de carga puede definirse como la asignación de trabajo a cada computador de tal forma que el tiempo de ejecución de la aplicación paralela-distribuida sea minimizado. De esta forma, el equilibrio de carga implica la asignación de trabajo a cada recurso de computación de manera que éste sea proporcional al rendimiento del mismo y al segmento de red que lo conecta al resto de computadores [NiS93].

Las estrategias de equilibrio de carga pueden ser clasificadas en dos categorías importantes: estática y dinámica. Los algoritmos estáticos de equilibrio de carga asignan el trabajo a realizar por los recursos de computación basándose en las características de rendimiento de éstos y de la red, como puede ser la velocidad del procesador, la carga del sistema y el *throuhgput* de la interfaz de red. Esto es, se realiza a priori una estimación de la carga a distribuir a cada computador. La ventaja de este tipo de algoritmos es que no introducen sobrecarga durante la ejecución de la aplicación paralela, puesto que son utilizados en tiempo de compilación, y debido a esto último, pueden ser diseñados empleando mecanismos complejos que contemplen todas las propiedades de la aplicación y del sistema. Por ejemplo, en [AtH04] se utiliza una estrategia estática de equilibrio de carga resuelta en dos fases antes de la ejecución de la aplicación. La

primera fase encuentra la asignación óptima de tareas a procesadores aplicando el algoritmo heurístico *Simulated Annealing* [HaH00]. La segunda fase utiliza la técnica de *Branch and Bound* [FMT02] tomando como solución inicial el resultado obtenido en la primera fase. Sin embargo, para un óptimo rendimiento, estas estrategias necesitan una estimación muy precisa de la utilización de los recursos por cada tarea antes de su ejecución, lo cual es generalmente muy difícil de conseguir en la mayoría de las aplicaciones [TaT85]. Por estos motivos, las estrategias estáticas de equilibrio de carga solamente son válidas cuando el entorno de computación está dedicado y su arquitectura es conocida e invariante, es decir, el número de recursos de computación no va a variar durante la ejecución de la aplicación. Por tanto, los algoritmos de equilibrio de carga estáticos no son adecuados para la ejecución de aplicaciones paralelas en un entorno de computación LAN-WLAN, donde la carga introducida por otras aplicaciones y usuarios influye de forma significativa en la ejecución de las tareas, el rendimiento del canal inalámbrico es sensible a pequeñas variaciones del entorno físico y tiene una tasa elevada de errores de transmisión, y los computadores de la WLAN se pueden incorporar o desvincular del entorno de computación en tiempo de ejecución.

Por otro lado, las estrategias dinámicas de equilibrio de carga estiman en tiempo de ejecución la carga a distribuir. Para ello, se utiliza información reciente acerca del rendimiento y estado de cada uno de los recursos de computación. Como resultado, estos esquemas proporcionan una mejora significante frente a las estrategias estáticas, sobre todo cuando las características del entorno varían de forma impredecible. Sin embargo, precisan de la tarea adicional de recopilar periódicamente información sobre los recursos de computación y sobre la evolución de los tiempos de ejecución de cada tarea, lo que implica una sobrecarga que podría afectar al rendimiento del sistema y, por tanto, al tiempo de ejecución global de la aplicación paralela.

Aunque existen muchas estrategias dinámicas de equilibrio de carga, hay cuatro pasos comunes que siguen la mayoría de los algoritmos [ZLP97]:

- *Obtención de información de rendimiento.* Con independencia de la estrategia utilizada, las estrategias dinámicas de equilibrio de carga necesitan conocer

cierto tipo de información acerca de las características y el estado actual de cada recurso de computación, como pueden ser el número de procesadores, la carga del sistema, la latencia de red, el ancho de banda de comunicación, etc. Esta información se utiliza para estimar de forma adecuada la distribución de datos a realizar. En este sentido, el Protocolo Simple de Gestión de Red (SNMP) [Sta97] permite monitorizar aspectos relacionados con el rendimiento de cada recurso de una red, e informar sobre cualquier evento o anomalía que se produzca. Por ejemplo, en [SMS03b] se utilizan el protocolo SNMP para obtener información de rendimiento de los recursos, y en base a esos datos y otras estrategias que se irán comentando, se realiza una estimación de la carga a comunicar a cada proceso para que el sistema alcance y permanezca en equilibrio.

- *Sincronización*. Consiste en el intercambio de información de rendimiento entre los recursos de computación en un instante determinado. Este paso se implementa dependiendo de la estrategia de equilibrio de carga utilizada. Por ejemplo, si se utiliza una estrategia distribuida donde cada recurso tiene que conocer el rendimiento del resto de computadores, éstos tienen que ponerse de acuerdo para realizar el intercambio de la información de rendimiento. Por el contrario, en una estrategia centralizada cada computador se sincroniza solamente con el computador donde se implanta la estrategia.

- *Estimación de datos*. Se estima la distribución de datos adecuada para reducir el tiempo en estado ocioso en el que pueden incurrir los procesos, y por tanto, se minimiza el tiempo de ejecución global de la aplicación. Esta estimación se puede realizar en base a un histórico del rendimiento de cada tarea en cada recurso, así como también se tiene en consideración el estado actual de cada recurso y del entorno de computación, permitiéndonos de esta forma predecir el tiempo de ejecución futuro de cada tarea.

- *Movimiento de datos y tareas*. Dependiendo del tipo de aplicación, cuando se detecta un desequilibrio de carga es necesario realizar una redistribución de

datos entre los procesos, o una migración de datos y/o tareas de un proceso a otro.

Debido a que el entorno de computación en el que centramos este trabajo de investigación tiene un comportamiento dinámico, las estrategias estáticas no contribuirán de forma efectiva a la minimización del tiempo de ejecución global. Por tanto, en este trabajo solamente las estrategias dinámicas serán estudiadas en profundidad.

2.3 Clasificación de las estrategias dinámicas de equilibrio de carga

En términos generales, el equilibrio de carga es una línea de investigación abierta *(hot topic)* en computación heterogénea [BSM01], y ello ha contribuido a que en los últimos años varios autores hayan propuesto técnicas o estrategias para mejorar el tiempo de ejecución de aplicaciones paralelas-distribuidas en entornos de computación heterogéneos.

La elección adecuada de un algoritmo de balanceo de carga no es una tarea fácil, y normalmente depende del dominio específico de la aplicación. Algunas estrategias están enfocadas hacias aplicaciones que utilizan un volumen alto de comunicación de datos, mientras que otras están diseñadas para aplicaciones intensivas en cálculo. Incluso, para una misma aplicación, diferentes estrategias pueden ser óptimas, dependiendo de parámetros como el número de procesadores, número de iteraciones, volumen de datos transmitidos, etc.

Las estrategias dinámicas de equilibrio de carga se pueden clasificar en base a dos parámetros, los cuáles responden a dos cuestiones importantes [Mal00]:

i) Dónde se toma la decisión de equilibrar la carga, y

ii) Qué información se utiliza para realizar la estimación.

Para responder a la primera pregunta podemos decir que el balanceador de carga puede estar ubicado en un único computador o distribuido en todos los recursos. Por otro lado, la información de rendimiento puede ser obtenida de todos los computadores o por

grupos de computadores, respondiendo así a la segunda pregunta. Por tanto, en base a las respuestas a las preguntas anteriores, se puede distinguir entre estrategias centralizadas o distribuidas, y estrategias locales o globales. Estas estrategias pueden combinarse entre sí de forma que podemos tener: esquemas centralizados globales, esquemas distribuidos globales, esquemas centralizados locales y esquemas distribuidos locales. En la figura 2.4 se muestra una clasificación de las estrategias utilizando dos ejes de cartesianas.

2.3.1 Estrategias centralizadas y distribuidas

El esquema centralizado consiste en ubicar el balanceador de carga en un único computador, denominado computador maestro, y debido a esto, todas las estimaciones y decisiones sobre las distribución de carga son realizadas desde el mismo computador. En este esquema, el balanceador de carga detecta las situaciones de desequilibrio realizando un análisis sobre la información que recolecta de uno, varios o todos los computadores. En el caso de una estrategia centralizada global, ésta implica un intercambio de información desde todos los computadores que realizan cálculos hacia el balanceador de carga *(all-to-one exchange)* e instrucciones para la distribución de carga desde el computador maestro hacia el resto de los computadores *(one-to-all exchange)*. Por tanto, este tipo de esquemas puede limitar la escabilidad del sistema debido al cuello de botella que puede ser encontrado en el computador donde se ubica el balanceador de carga.

El esquema distribuido consiste en replicar el balanceador de carga en varios o todos los computadores, y debido a esto, las estimaciones y decisiones sobre distribución de carga son realizadas en cada computador que, por lo general también colabora en los cálculos. Este tipo de esquemas solucionan el problema de la escabilidad de los esquemas centralizados, sin embargo, precisan de un intercambio de información de rendimiento entre todos los computadores *(all-to-all broadcast exchange)* con la consecuente sincronización global en ambos sentidos de comunicación [ZLP97]. Por tanto, existe un compromiso entre ambas estrategias para seleccionar la más adecuada según las características de la aplicación paralela y del entorno de computación.

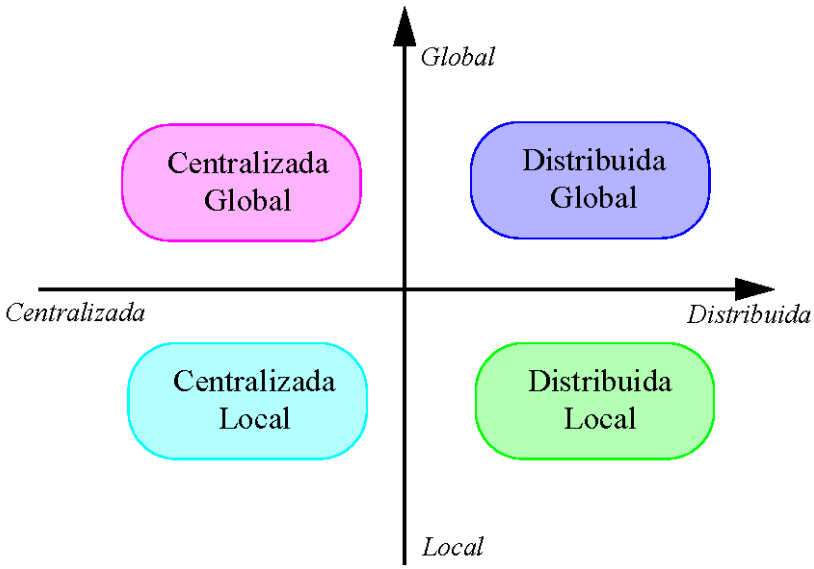


Figura 2.4 Clasificación de las estrategias dinámicas de equilibrio de carga

Dentro de los esquemas distribuidos, en [Mar97] se realizan dos clasificaciones: *bidding-algorithms* y *drafting-algorithms*. En los primeros, la idea es que los computadores sobrecargados migren unidades de carga hacia los computadores cuya carga está por debajo de un umbral, en función de la carga actual, algunos parámetros de rendimiento y un factor que refleja la distancia que separa el computador sobrecargado del posible computador destinatario de la carga. El valor de este factor puede aumentar o disminuir dinámicamente, dependiendo del número de computadores descargados que realicen peticiones de carga a los computadores sobrecargados. En los segundos, *drafting-algorithms*, cada computador mantiene una tabla que contiene la información más reciente de los procesadores vecinos, y en base a esta información, los computadores descargados solicitan más unidades de carga a los computadores sobrecargados.

Debido a que en este trabajo de investigación estamos interesados en la ejecución eficiente de aplicaciones que utilizan el paradigma Maestro/Esclavo, donde el proceso maestro se encarga de distribuir la carga, el esquema desarrollado implementa una estrategia centralizada que reside en el computador maestro.

2.3.2 Estrategias globales y locales

En los esquemas globales, las decisiones adoptadas en el equilibrio de carga se realizan en base al conocimiento global del sistema, es decir, todos los computadores envían su perfil de rendimiento al balanceador de carga. En estos esquemas, el estado de equilibrio de carga converge más rápido que en los esquemas locales debido a que todos los computadores son considerados al mismo tiempo. Sin embargo, requieren de sincronización y comunicación adicional entre todos los computadores. Esta sobrecarga se minimiza en los esquemas locales.

En los esquemas locales, los computadores son agrupados en diferentes grupos de tamaño *k*. La partición en grupos se puede hacer considerando la proximidad física de las máquinas, de manera aleatoria o agrupándolos por identificadores consecutivos [Raj99]. En este último caso, si hay N máquinas y P grupos, tal que $N = P \times k$, entonces en el primer grupo están los computadores con identificadores *{0, 1, ..., k-1}*, en el segundo grupo tendríamos *{k, k+1, ..., 2k-1}*, y así sucesivamente. Las decisiones de equilibrio de carga se realizan dentro de cada grupo, y por tanto, sólo los computadores de un mismo grupo envían su perfil de rendimiento al balanceador de carga de dicho grupo, reduciéndose así el tráfico de información por toda la red, y el número de computadores que tienen que sincronizarse. Sin embargo, para que los esquemas locales funcionen correctamente todos los grupos deben tener una potencia computacional similar, ya que si existen grupos con grandes diferencias de rendimiento, los grupos con procesadores más rápidos finalizarán antes que los grupos con procesadores más lentos [ZLP97]. Esta última situación puede ser solucionada proporcionando un mecanismo para intercambiar información entre grupos o modificando el número de miembros de cada grupo en tiempo de ejecución, lo cual puede implicar una sobrecarga adicional porque se requiere sincronización entre los intervinientes.

Dado que el entorno de computación LAN-WLAN tiene un comportamiento dinámico y los computadores pueden tener diferente rendimiento entre sí, no se pueden crear grupos de computadores con rendimiento similar, y por tanto, no se pueden implementar estrategias locales. Por este motivo, el esquema desarrollado en este trabajo

implementa un esquema global, donde existe un computador que conoce el estado y rendimiento del resto de computadores.

2.4 Equilibrio dinámico de carga en entornos heterogéneos

Durante muchos años, las estrategias dinámicas de equilibrio de carga en entornos de computación homogéneos, tales como máquinas paralelas, han sido estudiadas ampliamente [ELZ86][WiR93]. Sin embargo, estos esquemas no pueden ser aplicados directamente sobre una red de computadores heterogéneos debido a la gran cantidad de nuevos desafíos que éstas presentan: comunicaciones lentas, latencias elevadas, arquitecturas de procesadores heterogéneas, segmentos de red con diferentes velocidades de transmisión, movilidad de usuarios, etc. [ZLC95].

En términos generales, el diseño de un mecanismo de equilibrio de carga para entornos heterogéneos ha de tener en cuenta información sobre los siguientes aspectos:

- *Recursos de computación.* El rendimiento de cada computador puede ser igual o diferente (velocidad de procesamiento y tamaño de memoria heterogénea). El entorno de computación puede estar dedicado a la ejecución de aplicaciones paralelas o ser un entorno multiusuario o multitarea. Los computadores pueden estar disponibles para la ejecución de aplicaciones durante todo el tiempo o incorporarse y abandonar el entorno de computación en tiempo de ejecución.
- *Topología de la red.* Los enlaces de red pueden ser homogéneos o heterogéneos, e incluso para éste último caso pueden coexistir segmentos de red con diferentes anchos de banda y con diferentes protocolos de red. Por ejemplo, la combinación de los estándares IEEE 802.3 e IEEE 802.11.
- *Tipo de aplicaciones.* El tipo de aplicación que se implementa influye en la distribución de datos a realizar. En una primera clasificación, las aplicaciones paralelas pueden ser divididas en dos grupos: centralizadas y distribuidas. En las aplicaciones centralizadas hay un proceso que controla la distribución de carga (paradigma Maestro-Esclavo), y en las aplicaciones distribuidas todos los procesos toman decisiones. Por otro lado, si la aplicación puede ser dividida en

tareas independientes, entonces el equilibrio de carga puede ser llevado a cabo migrando tareas entre computadores, pero si la aplicación posee dependencias de datos entre iteraciones, entonces el equilibrio de carga tiene que ser realizado con distribuciones de datos periódicas en cada iteración.

Atendiendo a estos aspectos y debido al auge y popularidad de los entornos heterogéneos, numerosos autores han planteado nuevos algoritmos de equilibrio de carga sobre estos entornos.

En [Mah96] se presenta un algoritmo de equilibrio de carga para entornos de computación heterogéneos basado en la prioridad de las tareas. El algoritmo intenta reducir el tiempo de ejecución del programa paralelo determinando dinámicamente el grafo de dependencias de tareas y cambiando la prioridad de ejecución de aquellas tareas que más lo necesitan. En cada máquina existe un agente encargado de implementar un algoritmo de prioridad, el cual consiste en modificar la prioridad de las tareas en ejecución en función de las dependencias existentes entre todas las tareas del programa paralelo. De esta forma, si una tarea *i* está bloqueada por la espera de datos que tienen que ser comunicados por otra tarea *j* que se ejecuta en otra máquina, el agente de la primera máquina le envía un mensaje al de la segunda para que priorice la ejecución de la tarea *j*. Sin embargo, este esquema sólo es válido para aplicaciones donde existen dependencias entre tareas; en las aplicaciones en las que estamos interesados, del tipo Maestro/Esclavo con dependencias de datos entre iteraciones y donde las operaciones de cálculo que se realizan en cada iteración son similares, no se puede aplicar dicho esquema debido a que los procesos esclavos se sincronizan, en cada iteración, con el proceso maestro después de realizar los cálculos para enviar los resultados.

En [BLR03] se determina la mejor asignación de tareas para aplicaciones que siguen el paradigma Maestro/Esclavo sobre un conjunto de procesadores heterogéneos y que tienen que ser ejecutadas en un tiempo dado. Los autores estudian cuatro variantes de este paradigma, y para cada uno presentan un algoritmo que tiene en cuenta el patrón de comunicación: 1) comunicación sólo antes de lanzar todas las tareas, 2) comunicación antes y después de procesar todas las tareas, 3) comunicación antes de procesar cada

tarea, y 4) comunicación antes y después de procesar cada tarea. Los autores consideran la ejecución de tareas independientes en computadores dedicados, y todas las tareas representan la misma cantidad de procesamiento. También, ellos asumen que la red de comunicación es homogénea y sólo puede ser accedida en modo exclusivo. Los algoritmos presentados pueden ser aplicados a un amplio rango de problemas. Sin embargo, su solución no puede ser aplicada en aplicaciones con dependencias de datos ni en entornos de comunicación heterogéneos con variación significante del rendimiento, como ocurre en un entorno LAN-WLAN.

En [TQD01] se indican las consideraciones generales a tomar cuando una red formada por computadores heterogéneos entre sí se utiliza para realizar computación paralela. En este trabajo se utiliza la potencia de cálculo normalizada de cada computador para estimar la distribución de carga. Además, el entorno de computación es dedicado y todos los enlaces de red son homogéneos. Por tanto, el coste de comunicación es siempre igual para un volumen de datos dado. La utilización de este mecanismo trae consigo el equilibrio en los tiempos de cómputo de cada proceso sin tener en cuenta el tiempo de comunicación. Por otro lado, en [EgE02] se presenta un método dinámico de equilibrio de carga para entornos de computación no dedicados que se basa en parámetros relacionados con la carga actual de cada sistema y las características de los computadores. En concreto, los autores tienen en cuenta el número medio de trabajos en la cola *run* de cada recurso y la velocidad de cada CPU para estimar una distribución inicial de carga. En las iteraciones sucesivas, se redistribuye la carga en función del trabajo completado en un intervalo de tiempo, en oposición a [TQD01] donde el trabajo se reparte sólo en función de la potencia de cada computador. Es decir, se tiene en cuenta la progresión de la carga del sistema en cada recurso para realizar la redistribución de datos. Sin embargo, en este trabajo también se considera una red homogénea para interconectar los recursos. Por tanto, ambos trabajos pueden presentar desequilibrios de carga si sus mecanismos son aplicados en un entorno de computación donde la topología de red es heterogénea y la variación del rendimiento de la misma es significante, como ocurre en las redes inalámbricas.

En [LaT01] se presenta un esquema de equilibrio de carga dinámico para aplicaciones SAMR sobre sistemas distribuidos. El esquema presentado tiene en cuenta tanto la heterogeneidad de los procesadores como la heterogeneidad de la red. Para afrontar la heterogeneidad de los computadores, a cada procesador se le asigna un peso en función de su rendimiento. Cuando se realiza la distribución de carga, ésta se equilibra de forma proporcional a cada peso. Para afrontar la heterogeneidad de la red, el esquema divide el mecanismo de equilibrio de carga en dos fases: equilibrio global y equilibrio local. Para ello, define el concepto de *grupo* como aquel conjunto de computadores que tienen el mismo rendimiento y comparten el mismo segmento de red, es decir, un grupo representa un sistema homogéneo de computadores. La fase de equilibrio global se encarga de equilibrar la carga a nivel de grupos, y la fase de equilibrio local equilibra la carga dentro de cada grupo. Si se detecta algún desequilibrio dentro de un grupo, se migra carga desde los recursos más cargados hacia los computadores más descargados. Si se detecta algún desequilibrio entre grupos, se invoca un algoritmo heurístico que evalúa el coste de la redistribución de carga entre los grupos para alcanzar el equilibrio, y también estima la ganancia computacional obtenida al realizar la migración de datos. Si la ganancia es mayor que el coste que implica la redistribución de carga, ésta se realiza entre los grupos afectados. Sin embargo, aunque este esquema considera la heterogeneidad de la red, esta estrategia no es válida en nuestro trabajo de investigación porque el esquema presentado en [LaT01] no soporta que un computador se vincule o desvincule en tiempo de ejecución del entorno de computación. Además, si el tipo de aplicaciones a ejecutar posee dependencias de datos entre iteraciones, no existe la posibilidad de migrar la carga, pues una vez que la carga se distribuye en cada iteración, ésta se utiliza en su totalidad.

En [DeH02], se muestra un esquema de equilibrio de carga basado en un agente móvil. Basándose en unas tablas de encaminamiento, el agente móvil es capaz de desplazarse de forma autónoma y con cierta inteligencia por todos los computadores que forman el entorno de computación para realizar algunas acciones definidas. El agente es el responsable de la distribución y migración de tareas. Cuando el agente móvil llega a

un computador, éste recoge los datos acerca del redimiento actual del sistema (número de tareas activas y en cola). En función del estado del sistema, el agente puede decidir llevarse con él algunas tareas. Más tarde, las tareas recogidas pueden ser puestas en ejecución en computadores descargados. El agente móvil es ligero y sólo lleva consigo información acerca de los computadores y una lista con las tareas a ser transferidas. Además, una aportación interesante que se introduce en este trabajo es la posibilidad que tiene el agente de incluir nuevos recursos descargados de un conjunto de computadores disponibles si el entorno de computación está sobrecargado. Para poder realizar esta operación, en los nuevos recursos tiene que ejecutarse un *software* específico que permite que el agente móvil se pueda mover a ellos. Este trabajo presenta un mecanismo de equilibrio de carga muy novedoso ya que aparte de poder incluir nuevos computadores en tiempo de ejecución, la utilización de agentes móviles permite la reducción de carga en la red, ya que al ir desplazándose por todos los computadores, se evita el intercambio de información de rendimiento de todos los recursos con el computador que decide la migración de tareas. Sin embargo, la migración de tareas sólo puede ser aplicada sobre tareas independientes entre sí, y no se considera el coste de comunicación para mover una tarea de un nodo a otro. Además, si se usa un entorno LAN-WLAN donde la movilidad de los computadores puede provocar que éstos se desplacen a zonas de no cobertura de forma inesperada, el agente móvil no podría regresar al entorno de computación si éste se ejecuta en un computador que se ha quedado sin cobertura de forma repentina. Sin embargo, esta situación podría ser resuelta si el agente móvil migra a otro computador cuando se detecta que la calidad del enlace inalámbrico o el nivel de energía en la batería está por debajo de un determinado umbral. Para ello, se tendría que monitorizar periódicamente estos parámetros, acción que por lo que sabemos, no realizan los autores de este trabajo.

Por último, en [DHE05] se presenta un modelo de equilibrio de carga para computación distribuida sobre WLAN donde se tiene en cuenta el efecto de los retardos en la red. El modelo se divide en dos fases: primero se realizan una serie de experimentos para determinar la función de densidad de probabilidad del retardo de

comunicación de un nodo a otro en una red inalámbrica. A continuación, en la segunda fase, esta función de probabilidad se utiliza en un modelo matemático para diseñar una política de equilibrio de carga óptima basada en la cola de trabajos de cada sistema y la migración de tareas. El entorno utilizado para realizar las medidas consiste en dos computadores comunicándose a través de un punto de acceso no dedicado que sigue el estándar IEEE 802.11b. En una red IEEE 802.11 los retardos son aleatorios por naturaleza, el *jitter* es variable y la calidad de la señal, la latencia de red y el *throughput* de red tienden a variar significativamente con la posición geográfica de los computadores y los obstáculos presentes en el área de cobertura [KaK04]. Por tanto, como este modelo depende fuertemente de las medidas empíricas, no se puede aplicar en un entorno de computación donde los computadores portátiles pueden moverse de forma aleatoria y con alta probabilidad, y donde el rendimiento de la red puede variar significativamente cuando cambia el número de computadores.

Como ya se ha mencionado, en un entorno de computación LAN-WLAN, la heterogeneidad está presente tanto en la computación como en la comunicación. Estos aspectos pueden ser modelados para conseguir una distribución de carga de acuerdo al rendimiento del sistema. Sin embargo, en estos entornos es muy probable que los recursos portátiles cambien su localización física, entrando y saliendo de cobertura [Mac01]. Este hecho introduce nuevos cambios en el diseño de aplicaciones paralelas-distribuidas, y en particular, en el diseño de técnicas de equilibrio de carga, debido a que el número de procesos cambia en tiempo de ejecución de forma natural y con alta probabilidad. Claramente, si esta variación de procesos no se gestiona correctamente, se generan desbalanceos en los cálculos y en las comunicaciones [StS04], y por tanto, en los procesadores existirán ciclos de computación no utilizados y los tiempos de ejecución de las aplicaciones no serán los óptimos. Debido a esto, para estimar la distribución adecuada de carga a realizar sobre los nuevos procesos es importante diseñar una estrategia de equilibrio de carga que pueda conocer con antelación las características de rendimiento de los nuevos computadores que entran a formar parte del entorno de computación. Las características de los nuevos computadores pueden ser

conocidas con antelación si se dispone de una base de datos de los computadores que pueden participar en los cálculos. Sin embargo, esta situación no permitiría conocer el rendimiento actual de cada recurso, como pueder ser la carga de cada sistema y la tasa actual de transferencia de datos. Este problema puede ser solucionado ubicando en cada recurso un agente que se encargue de monitorizar periódicamente el sistema e informe al balanceador de carga sobre el estado y los cambios que se produzcan en ellos. En este sentido, el Protocolo Simple de Gestión de Red define una entidad, denominada agente, que puede ser programado para controlar el estado del recurso donde se ubica. Por ejemplo, en [SMS03b] se presenta una ampliación del *middleware LAMGAC* que utiliza una arquitectura basada en SNMP para obtener información de rendimiento de los computadores y suministrarla al esquema de equilibrio de carga implementado en dicho trabajo. Además, la estrategia de equilibrio de carga considera la incorporación en tiempo de ejecución de nuevos computadores. Todos los recursos de computación que quieran realizar cálculos paralelos formando parte de un sistema de carga equilibrado, incluidos los que se incorporan en tiempo de ejecución, tienen que ejecutar un agente SNMP extendido diseñado para tal fin.

2.4.1 Equilibrio de las comunicaciones y los cálculos

El auge de las nuevas tecnologías de comunicación, las cuales proporcionan un acceso rápido, económico y eficaz, han hecho posible que una arquitectura *hardware* LAN-WLAN, como la mostrada en la figura 2.1, pueda ser construida de forma sencilla y ser utilizada para ejecutar aplicaciones paralelas-distribuidas. En este tipo de entornos, a la heteregoneidad presente en la capacidad de procesamiento, hay que añadir la heterogeneidad presente en la comunicación de datos. Esto último se debe a la utilización de los conocidos estándares de red de la familia IEEE 802.3 (Ethernet, Fast Ethernet y Gigabit Ethernet), y a la aparición de nuevos estándares de red inalámbricos como IEEE 802.11 cuyos costes son bastantes asequibles a cualquier economía. La combinación de segmentos de red que utilizan diferentes estándares a nivel de la capa física y de enlace introduce nuevos desafíos en la ejecución eficiente de aplicaciones

paralelas. Por tanto, no sólo hay que contemplar el hecho de la diferencia de procesamiento presente entre los recursos, sino también hay que considerar las diferentes propiedades de los medios de transmisión y las características de los estándares utilizados; la influencia de la comunicación de datos es más notable cuando se ejecutan aplicaciones paralelas-distribuidas que realizan grandes transferencias de datos.

Para analizar el efecto del canal de comunicación, se ha medido el *throughput* y la latencia de comunicación en segmentos de red que utilizan diferentes estándares de comunicación. Las medidas se realizaron con la herramienta NetPIPE [TOC03] bajo unas condiciones de dedicación exclusiva por parte de los computadores y de los segmentos de red utilizados. Esta herramienta ha sido diseñada para evaluar el rendimiento de protocolos de red, como pueden ser: Ethernet, FDDI o ATM. La implementación utilizada en los experimentos está basada en MPI. A groso modo y en su configuración básica, su funcionamiento es el siguiente: la herramienta genera un tren de paquetes entre dos computadores partiendo desde un paquete con tamaño un byte hasta un paquete cuyo tamaño genera un tiempo de transferencia superior a un segundo. A partir del número de bytes transferidos y el tiempo empleado en cada comunicación se estima el *throughput* de red. En la tabla 2.1 se muestra la tasa de transferencia nominal de cada interfaz de red correspondiente a cada computador, origen y destino, según el experimento realizado sobre cada estándar de red.

Tabla 2.1 Tasa de transferencia

Experimento	***Origen***	***Destino***
Fast-Ethernet	100	100
Ethernet	100	10
802.11b	100	11
802.11	2	2

En los dos primeros experimentos se utilizó un *switch Fast-Ethernet* para conectar los dos equipos. En el tercer experimento se utilizó un punto de acceso con dos interfaces de red, la primera cumple con el estándar Fast-Ethernet y la segunda cumple con el

estándar IEEE 802.11b. El cuarto experimento se hizo mediante la visión directa y cercana de dos computadores con interfaz de red que cumple el estándar IEEE 802.11, y en modo *ad hoc*.

En la figura 2.5, se muestra la latencia de comunicación para diferentes estándares de red de área local, la cual viene indicada por el punto de comienzo de cada curva. Como se puede apreciar, la latencia obtenida en una red Fast-Ethernet (~46 microseg.) es alrededor de 20 veces inferior a la alcanzada en una red IEEE 802.11b (~890 microseg.). Por otro lado, otro aspecto importante que se aprecia en la figura anterior es la diferencia de *throughput* entre los estándares de la familia IEEE 802.3 e IEEE 802.11. Esta diferencia no se mantiene constante a lo largo de todo el experimento, obteniéndose antes el *throughput* máximo en los medios físicos cableados, debido principalmente a sus mejores características de propagación.

En la figura 2.6, se muestra el *throughput* real de cada estándar. Como se puede observar, todos los experimentos realizados están por debajo de su máximo teórico, existiendo diferencias más acuciadas en el estándar IEEE 802.11b, donde ligeramente se superan los 5 Mbps de los 11Mbps teóricos. Como se puede apreciar, la tasa de transferencia aumenta linealmente con el volumen de datos transmitidos, hasta un cierto punto, conocido como punto de saturación, a partir del cual la tasa de transferencia permanece relativamente constante. Este punto de saturación está situado en diferente lugar para cada estándar. También, se observa que aunque la tasa nominal del estándar IEEE 802.11b es superior a la tasa nominal del estándar Ethernet, su tasa real es inferior a esta última. Aquí se demuestra los acuciantes problemas que tienen las redes inalámbricas para realizar una transmisión correcta, ya que para una tasa de transferencia nominal bastante similar (10 Mbps para Ethernet y 11 Mbps para IEEE 802.11b) y un tamaño de bloque de 1 KB, se alcanza un *throughput* de 5.01 Mbps en Ethernet y de 2.52 Mbps en IEEE 802.11b, aproximadamente la mitad. La diferencia existente entre los estándares de comunicación, respecto a la tasa de transferencia, implica que los procesos de un programa paralelo ubicados en los computadores que implementan las interfaces más rápidas comiencen el proceso de cálculo antes que el resto, debido a que disponen

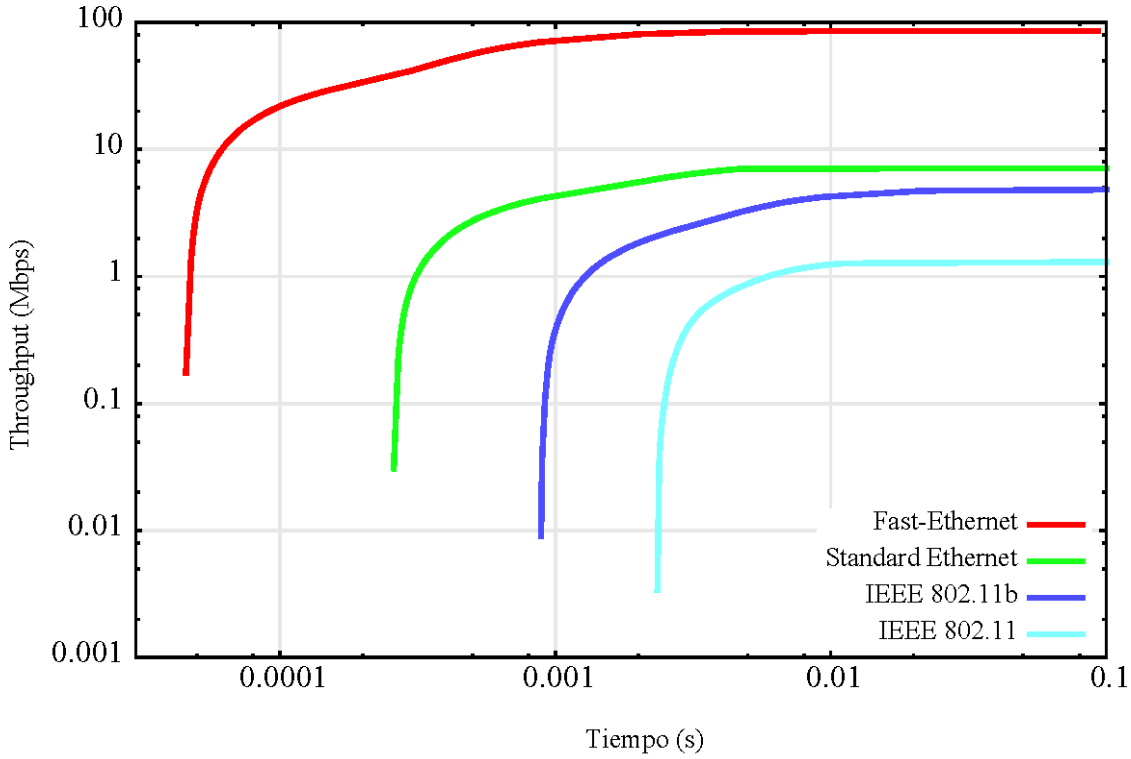


Figura 2.5 Latencia de comunicación

de los datos con antelación. De la misma forma, los resultados procedentes de estos computadores llegarán antes al computador de destino. A modo de ejemplo, para una transferencia de 10 KB, un computador que implemente Fast-Ethernet recibirá los datos aproximadamente un milisegundo después de iniciada la transmisión, sin embargo, un computador con interfaz de red basada en el estándar IEEE 802.11b tardará en recibir el mismo volumen de datos 17 mseg.

Todas estas diferencias nos demuestran que en un entorno de computación donde existen segmentos de red con tasas de transferencias diferentes, cualquier esquema de equilibrio de carga que se emplee tiene que considerar la heterogeneidad presente a nivel de las comunicaciones. De otro modo, siempre existirán desequilibrios en los tiempos de ejecución de las tareas. Si a todo esto añadimos que nosotros estamos interesados en la ejecución eficiente de aplicaciones paralelas-distribuidas que poseen dependencias de datos entre iteraciones, el problema se agrava aún más todavía debido a la sincronización

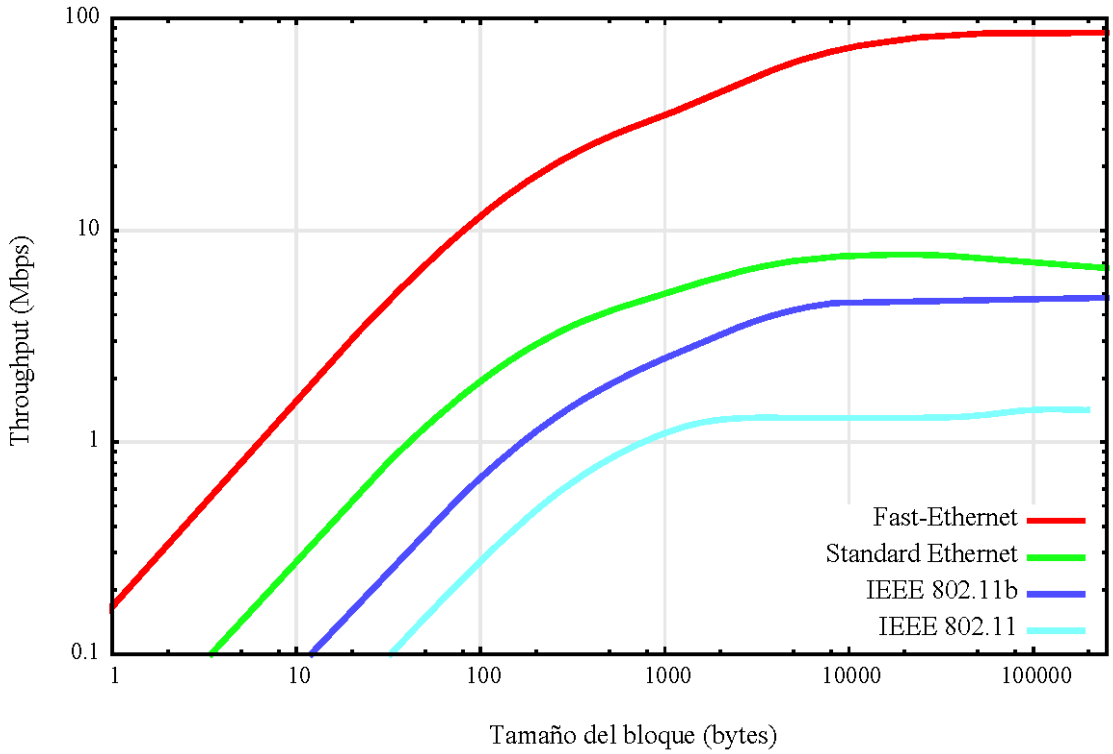


Figura 2.6 Throughput de red

de datos existente en cada iteración. Gráficamente, esta situación puede ser representada como se indica en la figura 2.7. Supongamos un entorno de computación formado por un computador, al que denominaremos computador maestro, donde se desarrolla un proceso encargado de distribuir la carga (proceso maestro), y dos computadores, a los que denominaremos computadores esclavos, donde se desarrollan los procesos esclavos encargados de realizar los cálculos paralelos. Todos los computadores tienen las mismas características en cuanto a procesamiento, sin embargo la interfaz de red del computador maestro y del computador esclavo A implementa el estándar IEEE 802.3 Fast-Ethernet y la del computador esclavo B implementa el estándar IEEE 802.11b. Si el proceso maestro distribuye el mismo volumen de datos a los dos procesos esclavos, el tiempo de cálculo empleado por cada computador será el mismo, pero los resultados obtenidos en el computador A serán enviados al computador maestro antes que los resultados obtenidos en el computador B. La consecuencia de esta situación es que el computador A estará en un estado ocioso durante un cierto tiempo hasta que finalice la iteración. En

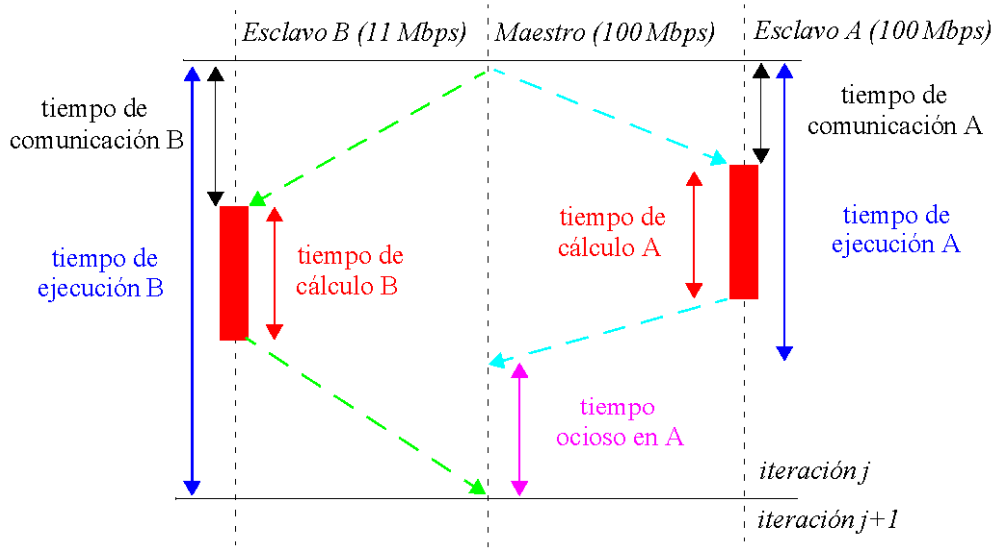


Figura 2.7 Desequilibrio en los tiempos de ejecución debido a segmentos de red diferentes

este caso, la única forma de evitar el tiempo ocioso del proceso que se ejecuta en el recurso A es enviando un volumen de datos diferente a cada proceso, tal que aunque los tiempos de cálculo de cada uno sean diferentes, los tiempos de ejecución de cada proceso sean similares, como se muestra en la figura 2.8, logrando que el sistema permanezca equilibrado.

Por tanto, es necesario tener en cuenta la heterogeneidad tanto en computación como en comunicación para lograr tiempos de ejecución equilibrados y evitar tiempos ociosos en los procesos. De esta forma, se minimiza el tiempo de ejecución global de la aplicación paralela.

2.4.2 Efecto del protocolo de transporte en IEEE 802.11

En la última década ha existido un incremento espectacular en el rendimiento de los sistemas de computación y de comunicación. La velocidad de los procesadores se dobla aproximadamente a un ritmo de 18 meses [HeP96]. Esta tasa de incremento, conocida como la Ley de Moore, se ha mantenido desde los años setenta hasta la actualidad, y se espera que continúe en el futuro inmediato. Por otro lado, el ancho de banda de red ha

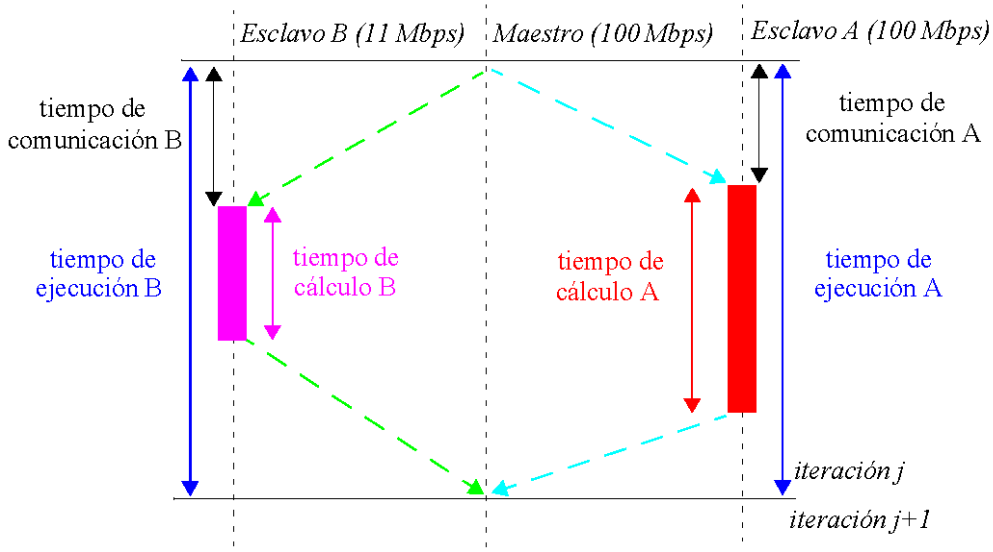


Figura 2.8 Equilibrio en los tiempos de ejecución utilizando segmentos de red diferentes

seguido una tasa de mejora similar, triplicando su rendimiento cada año, como indica la Ley de Gilder [Web-3].

Basándonos en la Ley de Moore y en la Ley de Gilder podríamos concluir que el rendimiento de aplicaciones distribuidas podría mejorar en tasas similares. Desafortunadamente esta mejora no se consigue, debido, entre otras cosas, a que la comunicación entre procesos no sigue las tasas de mejora indicadas por las leyes mencionadas. Esto se debe en parte a los protocolos de transportes fiables como TCP. En [Mar02] se estudia el rendimiento del protocolo TCP/IP en una red formada por computadores heteregéneos, tanto en potencia de cálculo como en *software* instalado, concluyendo que el rendimiento de TCP/IP no escala en la misma proporción que lo hacen los recursos, y que su pobre rendimiento se propaga hacia protocolos de alto nivel, como puede ser HTTP.

La utilización de sistemas donde el tráfico de información o la comunicación entre procesos fluye a través de redes heterogéneas complican aún más el rendimiento del protocolo TCP. En [NaS98] se presentan diversos esquemas que mejoran la congestión del tráfico TCP producido en un router que encamina los paquetes desde una red

Ethernet hacia una red ATM. Esta mejora se realiza regulando el flujo de los reconocimientos (ACK) retornados del destino al origen.

En una red cableada, la BER es despreciable, y la congestión es una de las principales causas de la pérdida de paquetes. Los enlaces de comunicación inalámbricos están sujetos a muchos factores incontrolables, tales como las condiciones del tiempo, obstáculos físicos, interferencias, reflexiones de las señales transmitidas, canal compartido, etc. Como resultado, las WLAN poseen una BER alta contribuyendo a la pérdida de paquetes. En [LeV00] se analiza el comportamiento del protocolo TCP original concluyendo que tiene un pobre rendimiento en las redes inalámbricas debido a su incapacidad para distinguir entre la pérdida de paquetes causada por la congestión y los paquetes perdidos causados por los errores de transmisión. Por tanto, el protocolo TCP necesita ser adaptado para ser eficiente en las redes inalámbricas.

En [TXA05] se analizan los problemas que tiene el protocolo TCP en diferentes entornos inalámbricos, tales como las redes de satélites, las redes *ad hoc* y las redes celulares. Para cada uno de estos entornos, se aportan posibles soluciones utilizando diferentes esquemas del protocolo TCP. Además, se presenta una comparativa entre diferentes implementaciones de TCP, como son: TCP-Vegas [BrP95], TCP-Veno [FuL04], TCP-Westwood [MCG01], TCP-Jersey [XTA04], TCP-Peach [AMP01], Freeze-TCP [GMP00], ATCP [LiS01], TCP-New Reno [FlH99] e I-TCP [BaB95], concluyendo que aquellos esquemas que utilizan un control de congestión proactivo, gestionan y utilizan el ancho de banda red de una forma más eficiente. También, las limitaciones de cobertura y la movilidad de los usuarios suelen generar *handoffs* con bastante frecuencia, apareciendo de esta forma deconexiones temporales. En [TXA05] también se demuestra que un breve evento de desconexión puede detener la transmisión TCP durante un periodo largo, generando de esta forma un degradamiento en el rendimiento de la red.

Por otro lado, la contienda de los computadores para acceder al medio inalámbrico, las colisiones, las pérdidas de paquetes y la presencia de terminales ocultos pueden contribuir negativamente al rendimiento del protocolo MAC para IEEE 802.11,

influyendo negativamente en los protocolos de niveles superiores, sobre todo en el rendimiento de protocolos de transporte fiables como TCP. En [WPL02] se propone un nuevo protocolo, denominado DCF+, compatible con la especificación original de DCF (Distributed Coordination Function) del protocolo MAC de IEEE 802.11, que mejora el rendimiento del protocolo de transporte fiable TCP sobre WLAN. También, en [KaA00] se estudia la sobrecarga introducida por la capa física (PHY), el MAC de la capa de enlace y la capa de transporte para modelar el *throughput* de red del estándar IEEE 802.11b. Los resultados obtenidos con el modelo concuerdan con las medidas experimentales, obteniéndose un *throughput* real de 5.2 Mbps para una velocidad de transmisión de 11 Mbps. En dicho trabajo se concluye que parte de la pérdida de rendimiento se debe a la sobrecarga introducida por el protocolo de transporte TCP y el MAC, pero principalmente, alrededor de un 80% de las pérdidas, se debe a la sobrecarga introducida por la capa física, las colisiones en el canal y el tiempo ocioso entre paquetes.

En [PPP01] se realiza un estudio sobre el rendimiento del protocolo TCP/IP para las WLAN. Además, en dicho trabajo se presenta un algoritmo de retransmisión de paquetes perdidos utilizando un control de congestión sencillo que varía el tamaño del mensaje para adaptarse a las variaciones del ancho de banda del canal. Este último trabajo demuestra que el diseño de aplicaciones distribuidas sobre TCP/IP debe ser bien sintonizado cuando se mezclan comunicaciones cableadas e inalámbricas.

Por último, hacer notar que existen otros protocolos de transporte, como UDP, que al no soportar control de flujo, secuenciamiento de paquetes, corrección de errores, retransmisiones y reconocimientos, proporcionan una mejora sustancial en cuanto al rendimiento, respecto a los protocolos de transporte fiables como TCP. En [AAM01] se caracteriza el comportamiento del protocolo UDP, en términos de *throughput* y latencia en redes inalámbricas IEEE 802.11b. En el estudio realizado se comprueba que el rendimiento de aplicaciones que utilizan UDP se aproxima bastante al máximo teórico cuando se transmite en un canal ideal [Bin99], logrando 6 Mbps cuando se utiliza una tasa nominal de 11 Mbps. Por tanto, aquellas aplicaciones que emplean el protocolo

UDP para la comunicación de datos, como pueden ser las que están basadas en el protocolo SNMP, no disminuyen su rendimiento debido a las caracteríticas de las WLAN, afectando de forma mínima al rendimiento de otras comunicaciones basadas en TCP.

En este trabajo de investigación no modificamos el protocolo TCP ni utilizamos una implementación específica para los computadores portátiles. Sin embargo, debido al bajo rendimiento del protocolo de transporte en las WLAN, en este trabajo se desarrolla un esquema de equilibrio de carga que tiene en cuenta la tasa de transferencia de datos de cada recurso para enviarle la carga justa y necesaria. De este modo, el protocolo de transporte no se ve afectado por un tráfico incontrolado.

2.4.3 Control de energía en recursos que utilizan IEEE 802.11

Hoy en día, el consumo de energía ha llegado a ser un criterio de optimización tan importante en el diseño de sistemas como es el rendimiento de los mismos. Los dispositivos de computación desarrollados para entornos empotrados y móviles necesitan mecanismos de ahorro de la energía de la batería para estar operativos durante el mayor tiempo posible. Sin embargo, debido a su tamaño reducido y movilidad activa, estos recursos requieren un tamaño y peso de la batería tan pequeño como sea posible. Aunque se han realizado numerosos avances en el desarrollo de semiconductores, la tecnología disponible actualmente para el desarrollo de baterías no permite una independencia total de la red eléctrica. Debido a esta razón, muchos autores han dedicado sus esfuerzos en el desarrollo de diferentes técnicas, ya sean a nivel *hardware* o *software*, que permitan prolongar el tiempo de vida de la batería entre dos procesos de carga consecutivos.

La energía consumida por una tarjeta de red inalámbrica suele ser a menudo mucho mayor que el consumo de energía de una CPU. Por tanto, un objetivo a perseguir en entornos de computación con recursos portátiles es encontrar las restricciones temporales requeridas de diversas aplicaciones mientras se minimiza la cantidad de energía utilizada por las tarjetas de red inalámbricas. Típicamente, una tarjeta de red

puede estar en uno de los cinco estados siguientes: apagado, dormido (*sleep*), ocioso (*idle*), recepción y transmisión. En la tabla 2.2 se muestra la energía media consumida en cada estado por una tarjeta de red Cisco Aironet 350 con una tasa de transferencia teórica de 11 Mbps [MCD03]. Resulta obvio que maximizando el tiempo de *sleep* se incrementará la eficiencia en el consumo de energía. Sin embargo, hay que tener en cuenta el tiempo empleado en la transición de un estado a otro para evitar continuos cambios hacia y desde el estado *sleep*, con la consecuente pérdida de rendimiento.

Tabla 2.2 Disipación de potencia en diferentes estados

Estado	**Transmisión**	**Recepción**	**Idle**	**Sleep**
Potencia disipada (mW)	1680	1435	1340	185

En este sentido, diversos trabajos se han publicado para reducir el consumo de energía en los recursos portátiles. Debido a que el consumo de potencia de las tarjetas de WLAN es elevado, en [ZGH03] se propone una modificación del protocolo MAC del estándar IEEE 802.11 para minimizar el consumo de potencia permitiendo a las tarjetas de red pasar a un estado dormido cuando no hay transmisión de datos planificada. De esta forma, se reduce el tiempo en el que la interfaz de red está activa. También, en [PoL03], se propone un algoritmo, denominado *SmartNode*, utilizado para extender y mejorar el protocolo MAC del estándar IEEE 802.11 cuando se utilizan redes *ad hoc multihop*. La idea es que los nodos que ejecutan el algoritmo *SmartNode* estimen el nivel apropiado de potencia para iniciar la transmisión de datos a partir de la intensidad de potencia de la señal de los paquetes recibidos. Con los datos obtenidos en la estimación, se realizan los ajustes pertinentes sobre la potencia de emisión, mejorando de esta forma el uso del espectro y reduciendo los niveles de contención en la WLAN.

En [MoV03] se propone un *middleware* distribuido, denominado PARM, para optimizar el consumo de energía con independencia de las aplicaciones ejecutadas. A medida que la energía de los recursos disminuye, PARM reconfigura dinámicamente la distribución de componentes o servicios que se ejecutan en el recurso. Las decisiones

tomadas por PARM son enviadas a una entidad externa con conocimiento del sistema y capacidad para detener dichos servicios. Las decisiones se toman en base a un patrón realizado sobre el consumo de energía de varias aplicaciones y servicios.

A nivel de aplicación existen otros trabajos, como el que se presenta en [SKV02], donde se estudian diferentes técnicas de compilación que generan código optimizado para minimizar el consumo de energía. Estas técnicas tienen en consideración la gestión de la memoria, la planificación de la CPU y los eventos del recurso. En [XLW03] se estudia la utilización de la compresión de datos para reducir el consumo de potencia en los dispositivos portátiles cuando se realizan descargas o transferencias de archivos a través de una WLAN. Para ello, se presenta un modelo que reduce el consumo de energía, el cual consiste en solapar la comunicación de los datos con los cálculos implicados en la descompresión de los archivos descargados. De esta forma, a medida que se van descargando los bloques del archivo, éstos se van descomprimiendo aprovechando el tiempo ocioso de la CPU entre la descarga de un bloque y el siguiente.

Por último, un aspecto importante a tener en cuenta es que la optimización del consumo de energía no sólo debe ser considerada en situaciones estacionarias, sino también en situaciones de movilidad, ya que según la velocidad de desplazamiento que pueda tener un recurso que utiliza comunicación inalámbrica, éste puede necesitar mayor o menor potencia de transmisión para alcanzar al receptor, si se está alejando o acercando, respectivamente. En este sentido, en [YuS03] se analiza el efecto de la distancia de transmisión de los paquetes transmitidos en el consumo de energía. El estudio se realiza teniendo en consideración la situación estacionaria y dinámica de los recursos en modo *ad hoc*. En la situación estacionaria, se demuestra que existe un rango de transmisión óptimo que mejora el *throughput* de red a la vez que se minimiza el consumo de energía. Sin embargo, en una situación de movilidad, los resultados obtenidos demuestran que no hay un rango de transmisión que optimice el *throughput* de red, pero si hay un rango de transmisión tal que el consumo de energía se minimiza. En ambas situaciones, se determina que el rango óptimo de transmisión es mucho mayor que el rango de transmisión cuando los recursos están en una zona de cobertura reducida.

Como se puede apreciar, debido a la proliferación y auge de los dispositivos portátiles, la mejora en la eficiencia del consumo de energía es una área de investigación de interés elevado.

En un entorno de computación formado por recursos fijos y móviles es muy importante controlar durante el tiempo de ejecución de cada aplicación la disponibilidad de los recursos implicados. Uno de los problemas encontrados al hacer uso de computadores portátiles se produce cuando el proceso maestro espera por datos que nunca van a llegar, es decir, éste se mantiene a la espera por los resultados calculados en un computador esclavo que no puede comunicarse con el computador maestro. En una situación normal, un recurso portátil puede desaparecer del entorno debido a un desgaste de la energía remanente en la batería o al estar localizado fuera del área de cobertura. En este caso, el proceso maestro puede quedarse bloqueado indefinidamente o, si está programado eficientemente con un mecanismo de espera controlada, como el presentado en [SMS05b], puede continuar la ejecución volviendo a redistribuir los datos implicados en el cálculo de los resultados no obtenidos. Debido a esta situación, es muy importante detectar los recursos que no están disponibles o que pueden desaparecer del entorno de computación en breve.

En este trabajo de investigación no se ha desarrollado ninguna técnica para minimizar el consumo de energía en los computadores portátiles, sin embargo, sí hemos implementado un esquema que controla el nivel de energía en cada recurso de computación que está alimentado por una batería, y además, detecta cuando un recurso está fuera de cobertura. Respecto al nivel de energía de la batería de un recurso, si éste es inferior a un umbral establecido, el dispositivo inalámbrico envía una notificación al computador maestro. En base a esta información y al tiempo de ejecución de cada iteración, el proceso maestro decidirá si realiza o no la distribución de datos a dicho recurso, o cancela la recepción de resultados si la notificación se recibe durante el tiempo en el que se realizan los cálculos en paralelo. De esta forma, se evita la espera por datos que se sabe de antemano que no van a llegar, con la consecuente prevención del desequilibrio del sistema o finalización abrupta de la aplicación. De la misma forma, si

el nivel de potencia de señal recibido en un computador es inferior a un umbral, éste envía una notificación al computador maestro para que tenga en cuenta esta situación y monitorice aquel dispositivo ubicado en una zona de cobertura reducida. En [SMS03a] presentamos un esquema de equilibrio de carga que tiene en cuenta tanto el nivel de batería de los recursos portátiles como el nivel de potencia del enlace inalámbrico que se recibe. En [SMS05a] presentamos un esquema ligero para monitorizar los recursos portátiles situados en zonas de cobertura limitada. Este esquema se ejecuta de forma solapada con la aplicación paralela-distribuida.

2.4.4 Influencia de la posición de los computadores portátiles

Uno de los desafíos de la computación móvil es desarrollar técnicas que permitan el seguimiento y localización de usuarios [Zom02]. En este sentido, la computación basada en localización permite diseñar aplicaciones con capacidad para modificar su configuración y comportamiento en función de la posición del recurso donde se desarrollan [WaS01]. Por tanto, una estrategia de equilibrio de carga para entornos de computación LAN-WLAN puede beneficiarse de un mecanismo de localización de dispositivos para mejorar sus resultados. Con un mecanismo de localización se puede caracterizar el desplazamiento habitual de los computadores portátiles, y por tanto, se puede predecir con cierta probabilidad de éxito la disponibilidad, a nivel de cobertura, de estos computadores en un futuro inmediato. Esta información puede ser utilizada hábilmente por la estrategia de equilibrio de carga para conocer con anticipación la posición de los computadores, y por tanto, considerarlos o no en la siguiente distribución de datos. Por otro lado, este mecanismo resulta bastante útil e interesante cuando se aplica sobre redes *ad hoc*, debido a la variación del área de cobertura cuando los computadores que establecen la red se desplazan, provocando que algunos computadores no puedan comunicarse con el resto.

En los últimos años, muchos autores han dedicado sus esfuerzos al desarrollo de técnicas de localización en interiores utilizando WLAN basadas en el estándar IEEE 802.11. Por regla general, las técnicas desarrolladas utilizan un modelo basado en la

señal recibida (RSS), y son implementadas en dos fases. Durante la primera fase, se construye una base de datos que relaciona cada posible posición con un conjunto de muestras de RSS. A continuación, en la segunda fase, dado una muestra de RSS y un sistema de clasificación que utiliza la base de datos generada en la fase anterior, se determina la posición. Por tanto, la calidad de las muestras recogidas en cada posición determinará la calidad de los resultados. La mayoría de los autores diferencian sus trabajos en la segunda fase. En [WFL04][SCS03][YAU03] se utilizan métodos de clasificación diferentes basados en la técnica de vectores soporte (SVM), redes neuronales y distribuciones de probabilidad, respectivamente. Sin embargo, dependiendo del tamaño de la base de datos, estos métodos pueden ser computacionalmente costosos, y por tanto, introducen una carga adicional en el computador. Este hecho es un factor muy importante a tener en cuenta cuando se utilizan los computadores para ejecutar aplicaciones intensivas en cálculo.

Recientemente, en [SAM06] se han dado los primeros pasos para desarrollar un sistema ligero de localización. Este sistema sigue un modelo Maestro/Esclavo donde los computadores a localizar (esclavos) monitorizan la señal inalámbrica de los puntos de acceso, y envían los valores de RSS al computador maestro cuando detectan variación entre muestras consecutivas, indicador de que su posición ha variado. En base a estos datos, en el computador maestro se convierten las muestras de RSS a distancias a los puntos de acceso, y a continuación, se estima la posición de los recursos implementando un sistema de ecuaciones que resuelve el método de triangulación de las distancias. Para minimizar la sobrecarga en los computadores y en la red, debido a la monitorización y envío de información, se utiliza el protocolo SNMP. Debido al modelo seguido, este sistema se puede adaptar fácilmente a una estrategia centralizada de equilibrio de carga como la implementada en este trabajo de investigación.

2.5 El Protocolo Simple de Gestión de Red

SNMP es el protocolo más ampliamente utilizado para la administración de redes. La mayoría de los recursos utilizados en las redes empresariales poseen agentes que pueden

responder a consultas realizadas por uno o varios gestores. La facilidad para añadir componentes y configurarlos para su gestión, así como la sencilla implementación de las operaciones del protocolo, ha contribuido en gran medida a la aceptación y popularidad de los sistemas de administración de redes basados en SNMP [Sub00].

El núcleo de SNMP consta de un simple conjunto de operaciones que proporcionan a los administradores la habilidad para monitorizar y cambiar el estado de dispositivos que utilizan el protocolo SNMP. Por ejemplo, SNMP puede ser utilizado para apagar una interfaz en un *router*, comprobar la velocidad al cual está operando una interfaz Ethernet, controlar el número de paquetes TCP de entrada y salida, etc.

Un sistema de gestión basado en SNMP está compuesto por dos tipos de entidades: gestor y agente [MaS01]. Un gestor es una aplicación que puede realizar tareas de gestión o administración sobre una red, y es el responsable de realizar consultas (*polling*) y recibir notificaciones desde los agentes. Un *poll,* en el contexto de gestión de red, es la tarea de consultar a un agente (situado en un computador, *router*, *switch*, ...) solicitando o actualizando algún tipo de información. Esta información puede ser utilizada para determinar si algún tipo de evento significativo ha tenido lugar. La segunda entidad, el agente, es una aplicación que se ejecuta en cada dispositivo que el administrador desea gestionar. El agente puede ser un programa separado, como un duende en UNIX, o puede ser incorporado en el sistema operativo, como Cisco IOS en un *router.* Actualmente, la mayoría de los dispositivos IP poseen algún tipo de agente SNMP empotrado. El agente proporciona la información de gestión solicitada por el gestor monitorizando diversos aspectos del dispositivo gestionado. Un *trap* o notificación es la forma que posee un agente para informar al gestor, de forma asíncrona, que algún evento ha ocurrido en el dispositivo donde está ubicado el agente. El agente gestiona la base de datos de información de gestión, MIB, la cual puede ser vista como una base de datos que almacena información sobre los aspectos gestionados de cada recurso.

El protocolo SNMP utiliza el protocolo de transporte UDP para la comunicación entre las diferentes entidades, por tanto, es un protocolo no orientado a conexión, es decir, cada vez que se realiza una consulta o se envía una notificación, se crea una nueva

conexión. La utilización de este protocolo de transporte representa un beneficio con respecto al ancho de banda ocupado, ya que como se comentó en la sección 2.4.2, las aplicaciones que utilizan el protocolo de transporte UDP se ven mínimamente afectadas por las características de la red, logrando casi el *throughput* máximo teórico. Este hecho es más acuciado en las redes inalámbricas, donde la latencia alta de comunicación, la elevada congestión y el acceso a un medio compartido hacen que los protocolos de transporte fiables como TCP se vean perjudicados aún más.

En la figura 2.9 se muestra una configuración típica de una red donde los agentes SNMP están ubicados en los dispositivos a gestionar y, por tanto, cualquier sistema que permita un control vía SNMP puede ser controlado desde una estación de gestión.

2.5.1 SNMP como plataforma para obtener información dinámica

En [CBP02] se presenta una arquitectura adaptable para la gestión de la calidad de servicio en aplicaciones multimedia colaborativas sobre entornos heterógeneos que incluyen recursos cableados e inalámbricos. Esta arquitectura posee un componente, denominado *Network State Interface*, que utiliza el protocolo SNMP para determinar el estado de los computadores y de los elementos de red, como *routers* y *switchs*. Este componente obtiene datos sobre la carga de la CPU, memoria disponible, ancho de banda y latencia de red. Para ello, utiliza un gestor que se lanza en la estación de gestión y agentes que están ubicados en los diferentes recursos.

El NWS [WSP97] es un sistema para medir y predecir la utilización de recursos distribuidos. Esta información se utiliza para planificar trabajos en un entorno de metacomputación. Originalmente, este sistema tenía un componente denominado *Network/CPU Sensor* utilizado para obtener información de la red y de los procesadores. El protocolo SNMP ha sido utilizado en [BNA00] para mejorar el rendimiento del sistema NWS. Para ello, se implementaron dos módulos, un Sensor SNMP que obtiene los datos que son monitorizados por los agentes ubicados en los recursos, y un agente extendido para monitorizar y proporcionar predicciones tanto al Sensor como a otras aplicaciones externas basadas en SNMP. El beneficio de utilizar estos módulos basados

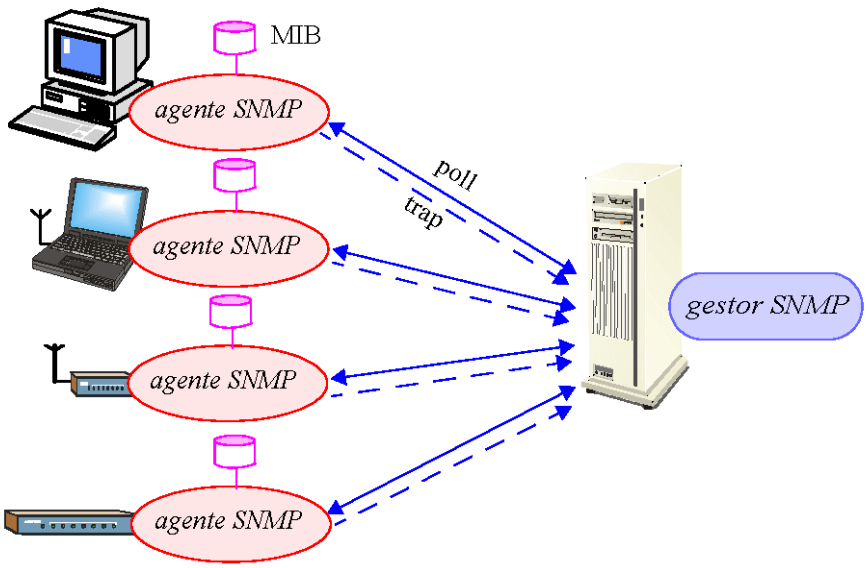


Figura 2.9 Configuración típica de un sistema de gestión basado en SNMP

en SNMP es que se puede obtener información de recursos que no tienen la capacidad de ejecutar el componente *Network Sensor* diseñado en [WSP97], como pueden ser en *routers*, *switchs*, *hubs*, etc, para obtener de una forma más precisa el rendimiento del sistema.

NICAN [SoC01] es una herramienta compuesta por varios módulos capaces de obtener información paramétrica acerca del entorno de computación, como puede ser el *throughput* de red, latencia, contención existente en la red y carga de la CPU. NICAN ofrece un mínimo impacto sobre la aplicación que la utiliza, y es capaz de proporcionarle los parámetros obtenidos de una forma clara y no intrusiva. En esta herramienta, el protocolo SNMP se utiliza para calcular el *throughput* de la red. En [KuS02] se utiliza la información de rendimiento de la red proporcionada por NICAN para minimizar el tiempo de espera debido a las comunicaciones en la solución de sistemas de ecuaciones lineales dispersas.

Los trabajos mencionados en esta sección tienen el denominador común en la forma de obtener los parámetros, es decir, cada gestor necesita hacer periódicamente un *polling* a los agentes para conocer el estado del sistema. Aunque el protocolo SNMP está

diseñado para introducir una mínima sobrecarga en el sistema donde se ejecuta, la acción de realizar consultas periódicas sobre cada recurso puede perjudicar el rendimiento del computador donde se ubica el gestor y de la red. Esta situación es más notable en entornos de red que tienden a crecer en el número de recursos gestionados, ya que se puede generar un cuello de botella a medida que se escala el sistema. Para minimizar esta sobrecarga, los agentes SNMP pueden ser programados para enviar notificaciones o *traps*, de forma asíncrona, cuando ocurre algún evento significativo o algún parámetro varía bruscamente, en lugar de monitorizar remota y periódicamente desde el gestor. En este sentido, en [DuN96] se presenta una versión modificada del algoritmo de *Diagnosis Distribuido Adaptativo* presentado en [BiB92] utilizando las capacidades de SNMP. En concreto, se emplean agentes SNMP ubicados en cada recurso que testean a sus vecinos inmediatos e informan al gestor mediante *traps*. De esta forma, el resultado es un esquema de diagnosis que tiene una alta tolerancia a fallos, y al mismo tiempo evita retrasos en la diagnosis al no realizar consultas periódicas a los agentes desde el gestor.

Como se puede apreciar, el protocolo SNMP ha sido utilizado ampliamente para obtener información sobre las características y estado de los recursos que forman parte de una red, así como para detectar e informar sobre posibles fallos ocurridos en los mismos. En [SMS03b] se presenta una arquitectura basada en SNMP, donde en cada recurso de computación se ejecuta un agente encargado de recopilar información sobre el rendimiento del recurso. En esta arquitectura, el gestor sólo se encarga de recibir y procesar las notificaciones enviadas por los agentes. Estas notificaciones solamente son enviadas cuando comienza un proceso del programa paralelo-distribuido o cuando el número medio de trabajos en la cola *run* del sistema varía de forma significativa, afectando por tanto al rendimiento del mismo. De esta forma, se evita que el gestor realice consultas periódicas sobre parámetros que no han variado, y por tanto, no se introduce tráfico adicional en la red ni se consumen ciclos de CPU. Sin embargo, en la actualidad, y hasta donde nuestro conocimiento alcanza, no existen trabajos donde aparte de obtener información de rendimiento de los recursos de un entorno de computación heterogéneo, se controle el nivel de batería remanente en cada recurso portátil, así como

el nivel del enlace inalámbrico. En un entorno formado por recursos de computación portátil, estos datos son muy importantes, ya que su conocimiento previo puede evitar el envío o recepción de datos que nunca van a ser recibidos en el computador maestro, debido a que la tarea no va a finalizar por la falta de energía en el recurso o la información no alcanzará su destino debido a que el computador está situado fuera del área de cobertura del punto de acceso. En [SMS03a] y en [SMS05a], basándonos en la arquitectura presentada en [SMS03b] ampliamos las funciones de los agentes SNMP para controlar el nivel de batería y la calidad del enlace inalámbrico que reciben los recursos portátiles.

2.6 Ideas generales sobre el middleware desarrollado

El *middleware* desarrollado en este trabajo de investigación proporciona al programador de aplicaciones paralelas-distribuidas, basadas en el paradigma Maestro-Esclavo, una serie de funciones para estimar y controlar, de forma transparente, la distribución adecuada de carga entre los procesos paralelos. De esta forma, los procesadores son utilizados durante el mayor tiempo posible, evitando estados ociosos, y por tanto, el tiempo de ejecución global de la aplicación paralela-distribuida se minimiza.

Como se mencionó al comienzo de este capítulo, este trabajo se implementó a partir de la biblioteca *LAMGAC*, desarrollada en [Mac01], cuyo objetivo principal es el manejo transparente de la variación de la VM_LAM en tiempo de ejecución desde la aplicación paralela. Por tanto, en todo momento se tiene conocimiento del número de computadores presentes en la WLAN. Este hecho es importante, dado que el mecanismo de equilibrio de carga implementado tiene que tener en cuenta, a parte de la heterogeneidad presente en un entorno de computación LAN-WLAN, el número exacto de computadores disponibles.

Para llevar a cabo un equilibrio de carga efectivo es necesario conocer con antelación el rendimiento de los computadores que participan en los cálculos, así como el rendimiento de las redes de comunicación utilizadas. Además, debido a la naturaleza cambiante y aleatoria de las características de las redes WLAN, hay que dotar al entorno

de computación de ciertos mecanismos que comprueben periódicamente los parámetros que afectan al estado de la red y de los computadores. En este sentido, para obtener información sobre el rendimiento del sistema que ayude a mejorar las estimaciones del mecanismo de equilibrio de carga, a la arquitectura *software* utilizada en [Mac01] se le ha incorporado una plataforma de gestión de red basada en el protocolo SNMP, existiendo un solapamiento entre los cálculos y comunicaciones de datos de la aplicación paralela-distribuida y la obtención de la información de rendimiento. En la figura 2.10 se muestra la nueva arquitectura *software*.

En esta plataforma, los agentes SNMP son ubicados en los recursos de computación esclavos, y periódicamente, monitorizan e informan a un gestor SNMP, de los eventos de importancia ocurridos. Tanto el proceso paralelo esclavo como el agente SNMP son procesos independientes que se desarrollan en cada computador esclavo. El proceso gestor se ubica en el computador maestro, y pone a disposición del mecanismo de equilibrio de carga los valores de los parámetros capturados por los agentes SNMP. Este proceso gestor es creado por el proceso maestro cuando este último comienza su ejecución. En la figura 2.11, se muestra la ubicación y solape de los distintos procesos dentro del entorno de computación.

Debido a su naturaleza, los computadores de la WLAN pueden desaparecer de forma inesperada del entorno debido a una ubicación fuera del alcance de la cobertura del punto de acceso inalámbrico o debido a que su batería se agote. Incluso, en las regiones donde el nivel de potencia recibido desde el punto de acceso es débil (cobertura limitada), donde las variaciones de la intensidad de señal son muy cambiantes, el recurso pudiendo permanecer fuera de cobertura en un instante determinado o temporalmente. Por tanto, en esta situación, el gestor SNMP debe aplicar un sistema de control periódico sobre aquellos recursos situados en un área de cobertura limitada. Si este control no es realizado, el computador puede salir de cobertura de forma inesperada, sin la posibilidad de que las notificaciones lleguen al computador maestro. El sistema de control consiste en que una vez detectado algún computador situado dentro de una zona de cobertura limitada (nivel de potencia por debajo de un cierto umbral), el gestor SNMP realiza

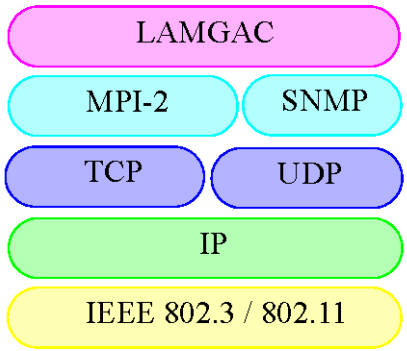


Figura 2.10 Nueva arquitectura software para el entorno de computación LAN-WLAN

consultas periódicas al agente sobre el valor del nivel de potencia hasta que dicho valor sea superior a un umbral (zona de cobertura aceptable) o hasta que no se reciban las respuestas de las consultas realizadas (fuera de cobertura). Para que todo esto sea posible, los agentes SNMP han sido debidamente programados para monitorizar la calidad del enlace inalámbrico y el nivel de energía remanente en la batería.

Por otro lado, para mantener la compatibilidad con las aplicaciones desarrolladas con LAMGAC, el modelo de computación que se ha diseñado sigue los patrones indicados en [Mac01]. Este modelo está basado en el diseño de tres esqueletos de programas: uno para cada tipo de computador del entorno (computador maestro, computadores de la LAN, y computadores de la WLAN). El computador maestro puede ser cualquier computador del entorno de computación que sea capaz de comunicarse con el resto de recursos; en este computador se ejecuta el proceso maestro. Los computadores de la LAN se consideran fijos y están accesibles durante toda la ejecución de la aplicación paralela. Los computadores de la WLAN pueden variar su posición geográfica, e incluso pueden entrar y salir de cobertura en tiempo de ejecución. Esta diferencia impone que se tengan que diseñar funciones específicas del *middleware* para los procesos que se ejecutan en los computadores fijos y para los que se desarrollan en los computadores portátiles.

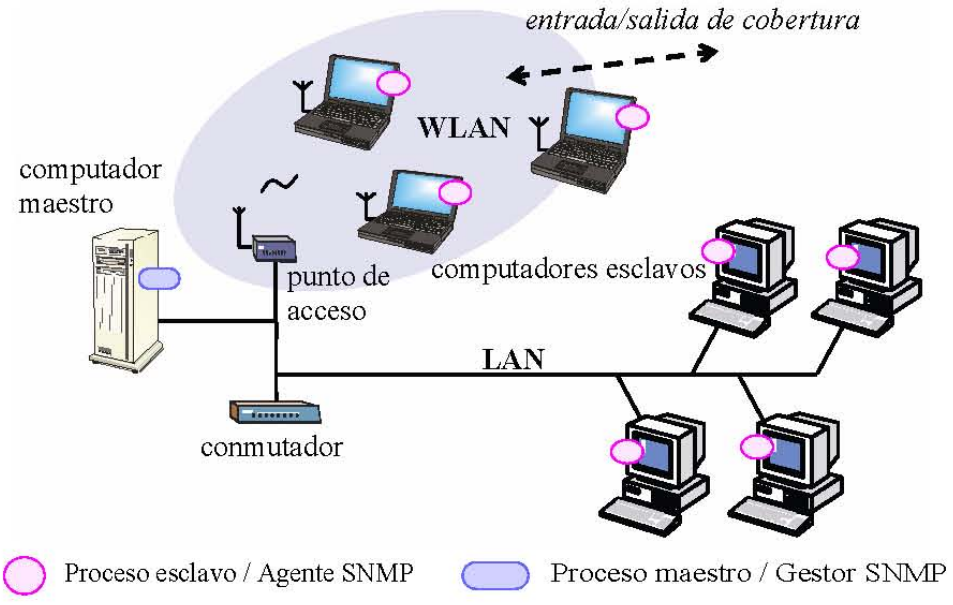


Figura 2.11 Solapamiento entre la ejecución paralela y la monitorización de recursos

3. El protocolo de equilibrio de carga

En este capítulo se presenta el desarrollo de una serie de primitivas que constituyen el protocolo de equilibrio de carga para entornos de computación y de comunicación heterogéneos. Para ello, previamente se analiza el diseño de una arquitectura software que facilita la obtención de información acerca de los recursos de computación, la cual es necesaria para llevar a cabo la estimación del volumen de carga a distribuir y controlar la batería y cobertura de los computadores portátiles.

3.1 Introducción

En este trabajo de investigación se ha desarrollado e implantado un protocolo de equilibrio de carga dinámico para aplicaciones que siguen el paradigma Maestro/ Esclavo, de tal forma que los procesos esclavos permanezcan en un estado ocioso el menor tiempo posible. De esta manera, se maximiza la utilización de los procesadores de los computadores de la VM_LAM reduciéndose el tiempo de ejecución global de la aplicación. Esta consideración es necesaria debido a que utilizamos un entorno de computación heterógeneo, tanto en computación como en comunicación, donde las aplicaciones paralelas-distribuidas a ejecutar pueden tener estrictas dependencias de datos entre iteraciones, y los procesos esclavos pueden ser expandidos, en tiempo de ejecución, sobre nuevos computadores portátiles que se incorporen al entorno de computación.

Generalmente, en este entorno de computación, la VM_LAM está formada por computadores con procesadores con diferente potencia de cálculo e interconectados entre sí por diferentes protocolos de red. En concreto, se utilizan los estándares de la familia de protocolos IEEE 802.3 e IEEE 802.11. El protocolo de equilibrio de carga diseñado en este trabajo de investigación debe conocer el rendimiento de cada computador que forma parte de la VM_LAM y de cada proceso que se ejecuta en éstos, para estimar la mejor distribución de datos, de tal forma que el tiempo ocioso de los procesos se minimice. Cualquier variación producida en un computador que afecte de forma inmediata al rendimiento del mismo y al entorno de computación debe ser notificada al proceso encargado de realizar la distribución de carga, que es el proceso maestro de la aplicación paralela, para que tome las consideraciones oportunas en la siguiente distribución de datos. Para este cometido, el protocolo SNMP, debidamente programado, es capaz de monitorizar diversos aspectos de los recursos de computación y generar una notificación cuando algún evento significativo ocurre en ellos. De esta forma, con SNMP se puede obtener información útil, tanto de los computadores conectados a la red cableada como de los computadores portátiles, para realizar una distribución de carga óptima y equilibrada.

Por esta razón, en primer lugar se ha de considerar una arquitectura *software* basada en el protocolo SNMP para obtener la información necesaria sobre el rendimiento de los diferentes recursos de computación. Además, esta arquitectura debe interaccionar con el programa paralelo para suministrarle la información sobre el entorno de computación. Con el fin de utilizar esta información para estimar adecuadamente la mejor distribución de carga, es necesario establecer un mecanismo mediante el cual ambas partes, arquitectura y programa paralelo, se sincronicen y la información generada tenga consistencia. Este mecanismo se implanta bajo las primitivas diseñadas en el protocolo de equilibrio de carga.

3.2 Utilizando SNMP para obtener información dinámica del entorno de computación

En este trabajo de investigación se ha desarrollado una arquitectura *software* basada en el protocolo SNMP para obtener información de los recursos de computación, tales como, potencia del procesador, *throughput* de la interfaz de red, calidad del enlace inalámbrico, energía remanente en la batería de los computadores portátiles, y otros parámetros que se detallan más adelante, en la tabla 3.1. El mecanismo de equilibrio de carga utiliza esta información para estimar el volumen de datos a distribuir a cada proceso esclavo. Por tanto, en este trabajo de investigación se aprovechan las características y funcionalidades del SNMP para ejecutar eficientemente aplicaciones Maestro/Esclavo en entornos heterogéneos de computación y comunicación.

La arquitectura SNMP desarrollada en este trabajo de investigación tiene dos tipos de entidades: a) un agente extendido, denominado *Colector*, encargado de monitorizar aspectos de rendimiento de los computadores y de enviar las notificaciones oportunas, y b) un administrador, denominado *Gestor*, encargado de recibir y procesar las notificaciones y de controlar la disponibilidad de los computadores portátiles cuando éstos están situados en zonas de cobertura limitada. Por tanto, el proceso *Colector* es un demonio UNIX que se ejecuta en cada computador del entorno de computación LAN-WLAN, y el proceso *Gestor* se ejecuta en el computador maestro junto con el proceso

maestro de la aplicación paralela-distribuida.

Para implementar dicha arquitectura, se han extendido las funcionalidades del agente estándar de la distribución Net-SNMP [Web-5], y para ello, se ha ampliado la base de datos de información de gestión con nuevos objetos y nuevas notificaciones. La información que va adjunta a las notificaciones enviadas por cada proceso *Colector* se almacena en una zona de memoria compartida entre el proceso *Gestor* y el proceso maestro de la aplicación paralela-distribuida, tal que ésta pueda ser accedida por este último mediante la llamada a una serie de primitivas del protocolo de equilibrio de carga. En la figura 3.1 se puede apreciar la interacción entre la arquitectura SNMP desarrollada y un programa paralelo que sigue el paradigma Maestro/Esclavo, así como la torre de protocolos utilizada en cada computador. Como se observa, dicha arquitectura es transparente al usuario, es decir, la información generada por ésta solamente es accesible a través de las primitivas diseñadas del protocolo de equilibrio de carga. Estas primitivas utilizan el SNMP y MPI-2 para llevar a cabo la interacción entre el programa paralelo y la obtención de información de rendimiento. A continuación se explica con más detalle el diseño de las entidades que forman parte de la aquitectura propuesta.

3.2.1 Proceso Colector

El proceso Colector es una extensión del agente estándar SNMP. Para llevar a cabo su implementación se han realizado dos tareas. Por un lado, se ha diseñado una nueva base de datos de información de gestión, denominada LBGAC-MIB [SMS03a][SMS03b], donde se almacenan los parámetros necesarios sobre el rendimiento y estado del computador donde se ejecuta el proceso *Colector*, y también, se definen los tipos de notificaciones que pueden ser enviadas por dicho agente. Estos parámetros se utilizan en el mecanismo de equilibrio de carga para estimar el volumen de carga a distribuir a cada proceso. Por otro lado, se han implementado diversos procedimientos para monitorizar dichos parámetros y enviar la notificación oportuna cuando alguno de ellos supera su umbral predeterminado.

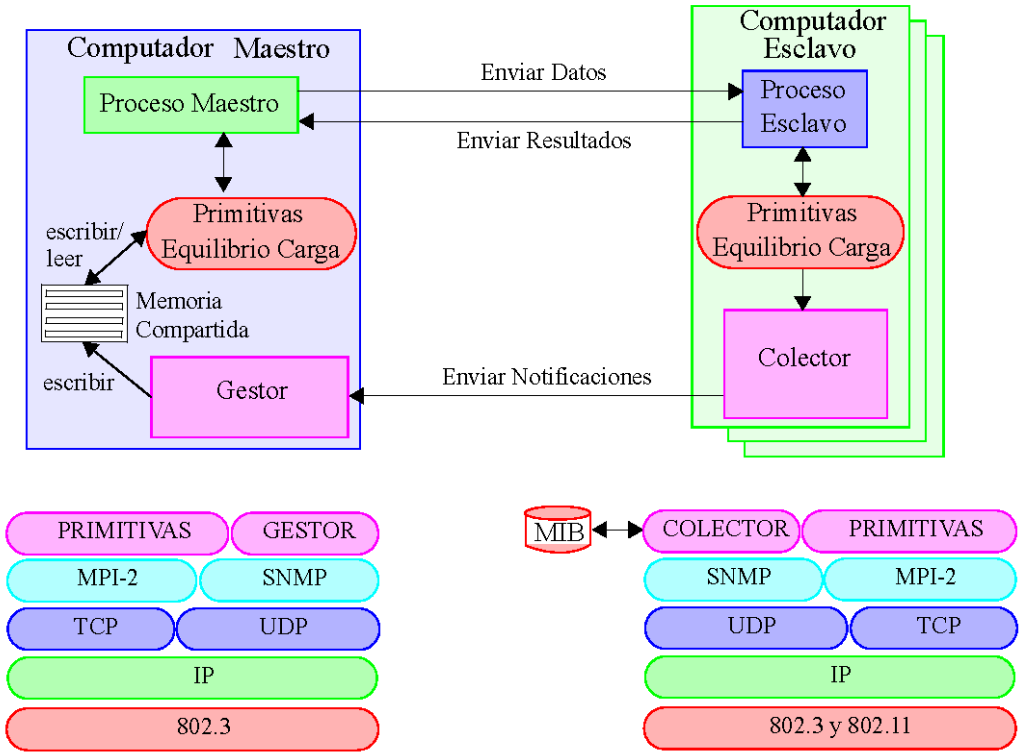


Figura 3.1 Interacción entre la arquitectura SNMP y la aplicación paralela-ditribuida

La información de gestión definida en LBGAC-MIB está ubicada en el grupo *lbgac* con identificador *1*, el cual pertenece al grupo *gac* con identificador *15100* (asignado por la IANA [Web-6]) dentro del grupo *enterprises* administrado por la IAB [Web-7]. En la figura 3.2 se puede observar la estructura jerárquica de LBGAC-MIB. Los objetos que almacenan las propiedades de los recursos gestionados están ubicados dentro del grupo *lbObjects*, las notificaciones están definidas en el grupo *lbNotif*, y los objetos que representan parámetros de configuración del agente están definidas en el grupo *lbConfig*. A continuación, se describe los objetos de cada uno de estos grupos.

Grupo *lbObjects*

Como se puede observar en la tabla 3.1, debido a que el entorno de computación está formado por computadores fijos y por computadores portátiles, en la base de datos LBGAC-MIB se han definido ciertos parámetros para controlar el estado de éstos

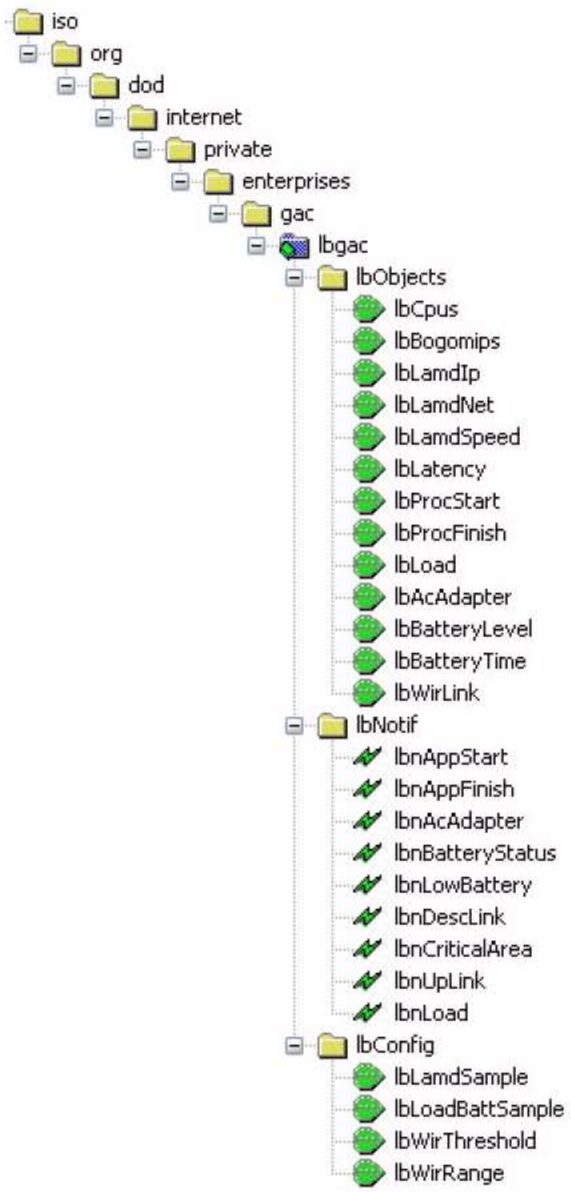


Figura 3.2 Estructura de la base de datos de información de gestión LBGAC-MIB

últimos, tales como la calidad del enlace inalámbrico, el nivel de energía remanente en la batería y el tiempo estimado de duración de la batería. El conocimiento de estos parámetros es muy importante, ya que los computadores portátiles pueden salir de la VM_LAM sin previo aviso debido a una localización geográfica fuera del área de cobertura del punto de acceso, o simplemente el nivel de energía en su batería se ha agotado. Por tanto, para evitar distribuciones de datos que a priori se sabe que no van a

Tabla 3.1 Objetos definidos en el grupo lbOjects

Objeto	Descripción	Tipo de dato
lbCpus	Número de procesadores que hay en el computador gestionado	Entero
lbBogomips	Valor que indica los bogomips del/los procesador/es	Entero
lbLamdIp	Dirección IP de la máquina desde donde se inició la VM_LAM. Debe coincidir con la dirección IP del computador maestro	Cadena
lbLamdNet	Indica el tipo de red, cableada o inalámbrica, que utiliza el demonio *lamd*	Entero
lbLamdSpeed	*Throughput* (bps) de la interfaz utilizada por el demonio *lamd* (distribución LAM-MPI)	Entero
lbLatency	Latencia de comunicación medida en segundos entre el recurso y el computador maestro	Cadena
lbProcStart	Indica el rango del último proceso MPI que comenzó su ejecución asociado al mecanismo de equilibrio de carga	Entero
lbProcFinish	Indica el rango del último proceso MPI que se desvinculó del mecanismo de equilibrio de carga	Entero
lbLoad	Número medio de trabajos en la cola *run* del sistema en el último minuto	Cadena
lbAcAdapter	Indica si el computador está conectado a la red eléctrica o es alimentado por una batería	Entero
lbBatteryLevel	Nivel de energía remanente en la batería (mAh)	Entero
lbBatteryTime	Tiempo estimado en segundos para agotarse la batería	Entero
lbWirLink	Indica la calidad del enlace inalámbrico si el demonio *lamd* utiliza el estándar de comunicación IEEE 802.11 (normalmente en computadores portátiles)	Entero

llegar a su destino, y también, para evitar la espera de resultados que se sabe con antelación que tampoco van a ser recibidos, es necesario tener en cuenta estos parámetros para controlar el estado y disponibilidad de los computadores portátiles. Esto

es, el proceso maestro y el mecanismo de equilibrio de carga deben tener conciencia del número de computadores que están operativos y disponibles en el instante de realizar la distribución de datos y en la recepción de resultados. A continuación, se explica cómo se obtiene el valor de cada uno de los objetos, también denominados parámetros, definidos en tabla 3.1.

Los parámetros *lbCpus* y *lbBogomips* dependen de las características físicas de cada computador. Por tanto, sus valores permanecen estáticos durante la ejecución de la aplicación paralela-distribuida, y por ello, son calculados cuando el agente SNMP comienza su ejecución. Para conocer ambos parámetros, el agente consulta y extrae del fichero */proc/cpuinfo* del sistema de archivos UNIX la información correspondiente.

Por otro lado, los parámetros *lbLamdIp, lbLamdNet, lbLamdSpeed* y *lbLatency* dependen de la interfaz de red del computador esclavo utilizada por el demonio *lamd* para comunicarse con el resto de computadores de la máquina paralela virtual. El proceso *lamd* es un demonio de la distribución LAM/MPI [Web-4] que se inicia en cada computador cuando se establece la VM_LAM. Este demonio permite la comunicación entre los procesos del programa paralelo. Los valores de los dos primeros parámetros se calculan cuando dicho demonio es lanzado en el computador (el proceso *Colector* detecta la ejecución de un proceso *lamd*), es decir, cuando el computador pasa a formar parte de la VM_LAM. Además, debido a la variación dinámica del rendimiento de la WLAN, los valores de los dos últimos parámetros son calculados periódicamente. Por defecto, el periodo de muestra se establece cada cinco minutos, aunque este valor puede ser alterado realizando una operación *Set* del protocolo SNMP sobre el parámetro *lbLamdSample* del grupo *lbConfig*. Para calcular el parámetro *lbLamdIp,* el agente realiza una serie de consultas para determinar la dirección IP de la máquina desde donde se inició el demonio *lamd*, concretamente analiza la información suministrada por el comando de UNIX *ps* invocado sobre dicho demonio. Una vez obtenida la dirección IP, averigua la interfaz de red utilizada por el demonio *lamd* (comando *route*), y a continuación se determina si dicha interfaz corresponde a un enlace IEEE 802.3 o IEEE 802.11 utilizando el comando *iwconfig*. Esta información se almacena en el parámetro

lbLamdNet. El agente SNMP estándar nos permite conocer la tasa de datos nominal de las interfaces de red que cumplen el estándar IEEE 802.3, y éste puede ser fácilmente extendido para las interfaces de red que utilizan el estándar IEEE 802.11. Sin embargo, debido a la heterogeneidad de las comunicaciones y a la no exclusividad del entorno de computación, se ha diseñado una rutina que determina la tasa real de transferencia de datos, el *throughput* de red. Para ello, el proceso *Colector* lanza una aplicación MPI que crea dos procesos paralelos. Uno de ellos se ejecuta de forma local en la entidad esclava, y el otro, en la entidad maestra. Dicha aplicación está basada en la herramienta NetPipe [TOC03]. Básicamente, la aplicación mide el tiempo de ida y vuelta de un tren de paquetes, y en base a ello calcula la transferencia de bits por segundo correspondiente a cada paquete enviado. Para minimizar la sobrecarga en la red y en las entidades que genera la ejecución de esta aplicación, el tamaño de los paquetes a enviar es de 1 byte, para calcular la latencia de red, y de 4 KB, para calcular el *throughput* de red. Se ha elegido este último valor, por ser un tamaño medio cercano al punto de saturación de los estándares de comunicación utilizados en este trabajo de investigación, los cuales se pueden observar en la figura 2.6.

El valor de los parámetros *lbProcStart* y *lbProcFinish* se actualiza automáticamente cuando el proceso MPI que se ejecuta en el computador esclavo invoca las primitivas *Iniciar_EF* o *Iniciar_EP* y *Finalizar_Cooperante*, respectivamente. Estas primitivas se invocan al comienzo de la ejecución del proceso esclavo y al finalizar el mismo, respectivamente. Estas primitivas se explicarán en detalle en el apartado 3.3.1.

El resto de parámetros, debido a que pueden variar su valor durante la ejecución de la aplicación paralela, necesitan ser monitorizados periódicamente. Por defecto, el parámetro *lbLoad* se extrae del archivo */proc/loadavg* cada sesenta segundos. Los parámetros *lbAcAdapter* y *lbBatteryLevel* se obtienen de los archivos ubicados en el directorio */proc/acpi*, los cuales también son monitorizados cada sesenta segundos. El periodo de monitorización de estos tres últimos parámetros puede ser también configurado realizando una operación *Set* sobre el parámetro *lbLoadBattSample* perteneciente al grupo *lbConfig*. El parámetro *lbBatteryTime* se estima con la ecuación

3.1 cuando el computador portátil trabaja con la batería, y cada vez que se actualiza el valor del parámetro *lbBatteryLevel*. En caso contrario, cuando el computador está conectado a la red eléctrica, se asume un valor infinito. En la ecuación 3.1, la variable t_s es el tiempo de actualización del parámetro *lbBatteryLevel*. Las variables *lbBatteryLevel(t)* y *lbBatteryLevel(t-t_s)* se corresponden con el valor de *lbBatteryLevel* en el instante actual y en la última consulta realizada, respectivamente.

$$lbBatteryTime = \frac{lbBatteryLevel(t) \times t_s}{lbBatteryLevel(t - t_s) - lbBatteryLevel(t)} \qquad (3.1)$$

La calidad del enlace inalámbrico decrece cuando aumenta la distancia entre transmisor y receptor, y en interiores, los efectos de propagación tales como la reflexión, refracción y difracción influyen en la señal recibida, y hacen que ésta no sea estable, varíe aleatoriamente y posea un *jitter* variable. Además, normalmente los usuarios de computadores portátiles permanecen cerca de éstos, y debido a que la frecuencia de resonancia del agua es 2.4 Ghz y el cuerpo humano está formado por un 70% de agua, la señal inalámbrica es absorbida y atenuada en presencia de usuarios [KaK04]. Por tanto, es necesario establecer una función de monitorización dinámica para actualizar el parámetro *lbWirLink*. Este control dinámico consiste en variar el tiempo de muestreo de calidad de la señal recibida en función de la variación que ésta tenga en las últimas muestras adquiridas. Su funcionamiento es el siguiente: cuando la diferencia entre la calidad del enlace inalámbrico entre muestras tomadas de forma consecutiva se mantiene dentro de un margen establecido en el parámetro *lbWirRange*, el tiempo de monitorización se fija en tres segundos. Sin embargo, cuando se detectan variaciones considerables entre tres medidas consecutivas, fuera del margen establecido, el tiempo de monitorización se fija en un segundo. Además, cuando el nivel de señal es inferior a un determinado umbral, el tiempo de monitorización también se fija en un segundo. La calidad del enlace inalámbrico se calcula como la diferencia entre el nivel de la señal recibida y el nivel de ruido, cuyos valores se extraen del archivo */proc/net/wireless*, el cual es actualizado por el driver de la interfaz de red inalámbrico. En la figura 3.3 se

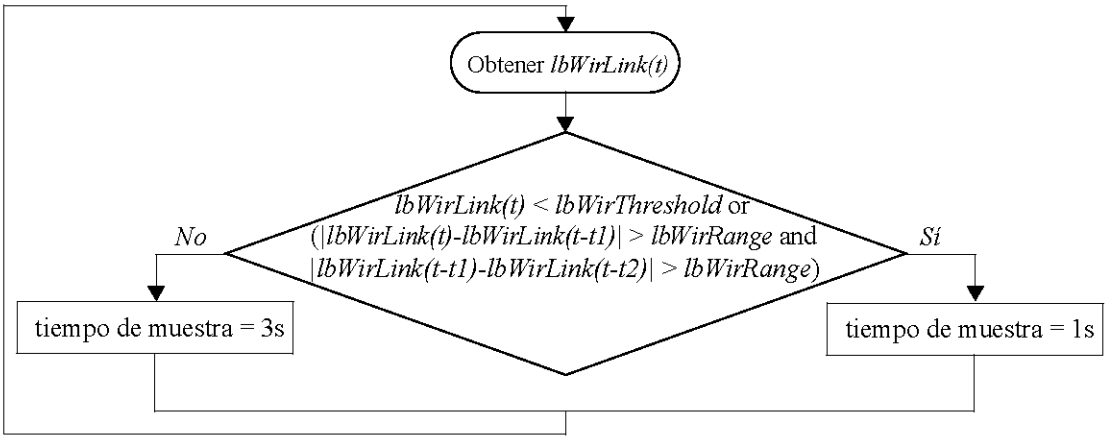


Figura 3.3 Diagrama de flujo para la monitorización dinámica del parámetro lbWirLink

muestra el diagrama de flujo de este sistema de monitorización dinámica. Por defecto, el umbral (parámetro *lbWirThreshold*) y el margen de confianza (parámetro *lbWirRange*) se establece en -80 dBm y 5 dB, respectivamente, aunque estos valores pueden ser actualizados realizando una operación *Set* sobre los parámetros correspondientes del grupo *lbConfig*.

Grupo *lbNotif*

Cuando un proceso *Colector* detecta un evento significativo, éste envía al proceso *Gestor* una notificación acerca del evento producido. La descripción de estas notificaciones se especifica en la tabla 3.2. Como se puede observar, se han incluido notificaciones específicas de los computadores portátiles para notificar el estado del nivel de energía de su batería y la calidad de la señal recibida desde el punto de acceso.

Las notificaciones *lbnAppStart* y *lbnAppFinish* son enviadas hacia el computador maestro cuando un proceso esclavo escribe su rango, dentro de la aplicación paralela-distribuida MPI, en el objeto *lbProcStart* y *lbProcFinish*, respectivamente, es decir, cuando un proceso paralelo comienza o finaliza su vinculación al mecanismo de equilibrio de carga.

La notificación *lbnAcAdapter* se envía cuando el objeto *lbAcAdapter* indica que el computador portátil está conectado a la red eléctrica. Por tanto, esta notificación se envía

Tabla 3.2 Notificaciones definidas en LBGAC-MIB

Notificación	Descripción	Parámetros notificados
lbnAppStart	Indica que un proceso MPI ha comezado su ejecución y quiere ser considerado por el mecanismo de equilibrio de carga	*lbCpus, lbBogomips, lbLamSpeed, lbLamdNet, lbLatency, lbBatteryTime, lbLoad, lbAcAdapter, lbWirLink, lbProcStart*
lbnAppFinish	Indica que un proceso MPI no quiere ser considerado por el mecanimo de equilibrio de carga	*lbProcFinish*
lbnAcAdapter	Indica que el computador está conectado a la red eléctrica	
lbnBatteryStatus	Indica que el computador está conectado a su batería	*lbBatteryLevel, lbBatteryTime*
lbnLowBattery	Indica un nivel bajo de batería (se estima un tiempo inferior a 10 minutos para que se agote)	*lbBatteryLevel, lbBatteryTime*
lbnDescLink	Indica que la calidad del enlace inalámbrico está disminuyendo	*lbWirLink*
lbnCriticalArea	Indica que el computador está situado en zona de poca cobertura, por debajo de un determinado umbral	*lbWirLink*
lbnUpLink	Indica que el computador está situado en una zona con cobertura. Esta notificación se envía solamente después de haberse enviado una notificación del tipo *lbnCriticalArea*	*lbWirLink, lbAcAdapter, lbBatteryTime, lbLoad*
lbnLoad	Indica que el número medio de trabajos en la cola *run* del sistema ha variado fuertemente en el último minuto	*lbLoad*

cuando el computador pasa de ser alimentado por su batería a estar conectado a la red eléctrica.

La notificación *lbnBatteryStatus* se envía cada cinco minutos cuando el computador portátil está alimentado por su batería. Se ha elegido este intervalo de notificación

relativamente corto porque la energía de la batería es un parámetro crítico cuya descarga no sigue una relación lineal, ya que depende de factores como la carga del sistema, utilización de periféricos, etc. Por tanto, el proceso *Gestor* debe estar informado con frecuencia sobre el estado de la misma. La notificación *lbnLowBattery* se envía cada minuto cuando el tiempo estimado para que la batería se agote es menor a diez minutos, y en caso contrario no se envía esta notificación.

La notificación *lbnDescLink* se transmite si la calidad del enlace inalámbrico, parámetro *lbWirLink*, disminuye durante tres medidas consecutivas. La notificación *lbnCriticalArea* se envía cuando la calidad del enlace inalámbrico está por debajo de un umbral previamente establecido. La notificación *lbnUpLink* se envía una vez que el nivel de señal tiene un valor aceptable, por encima del umbral, después de haber enviado una notificación del tipo *lbnCriticalArea*. Diversas puebas experimentales realizadas dieron como resultado que cuando la calidad del enlace es inferior a 10 dB es muy probable que se pierda la conexión inalámbrica. Por este motivo, el umbral de calidad del enlace inalámbrico se establece en dicho valor.

Por último, la notificación *lbnLoad* se envía cuando el número medio de procesos en la cola *run* del sistema (*lbLoad*) ha aumentado al menos en una unidad en un minuto. Este valor indica que al menos hay dos procesos utilizando de forma intensiva el procesador del sistema durante el último minuto.

Grupo *lbConfig*

Los objetos definidos en el grupo *lbConfig* especifican valores de configuración del proceso *Colector* referentes al tiempo de actualización de los valores de los objetos del grupo *lbObjects,* umbral de área de cobertura limitada y margen entre medidas consecutivas de la calidad del enlace inalámbrico utilizado en la monitorización dinámica del parámtero *lbWirLink*. Estos valores puede ser configurados por el usuario de forma remota realizando una operación SNMP *SetRequest*. En la tabla 3.3 se muestra una descripción de los parámetros.

Tabla 3.3 Objetos definidos en el grupo lbConfig

Objeto	Descripción	Tipo de dato
lbLamdSample	Especifica el tiempo de actualización en segundos de los parámetros *lbLatency* y *lbLamdSpeed*	Entero
lbLoadBattSample	Especifica el tiempo de actualización en segundos de los parámetros *lbLoad, lbAcAdapter* y *lbBatteryLevel*	Entero
lbWirThreshold	Especifica el umbral de la calidad del enlace en dBm que define un área de cobertura limitada	Entero
lbWirRange	Especifica un rango en dB que define la variación máxima entre dos medidas consecutivas de la calidad del enlace	Cadena

3.2.2 Proceso Gestor

Como se ha descrito anteriormente, el proceso *Gestor* implantado dentro de la arquitectura basada en SNMP, se encarga, como tarea principal, de recibir y decodificar las notificaciones enviadas desde cualquier proceso *Colector*. La información que se recibe en cada notificación se almacena en una zona de memoria compartida (figura 3.1) accesible también por el proceso maestro de la aplicación paralela. Además, se encarga de controlar los computadores portátiles cuando éstos están ubicados en zonas de cobertura reducida. En particular, el proceso *Gestor* es un proceso creado por el proceso maestro de la aplicación paralela-distribuida.

En la figura 3.4 se indican las acciones del proceso *Gestor*, las cuales se describen a continuación.

Para recibir notificaciones, el proceso *Gestor* debe iniciar una sesión SNMP, en la cual se especifica el puerto *udp:15162* para la recepción de notificaciones, la función *callback* para la decodificación de las mismas y otros parámetros referentes al protocolo SNMP. A continuación, una vez iniciada la sesión SNMP, el proceso *Gestor* envía una señal del tipo *LAM_SIGA* (señal de usuario definida en la distribución LAM/MPI [Web-4]) al proceso maestro para indicarle que ya está preparado para recibir notificaciones, y

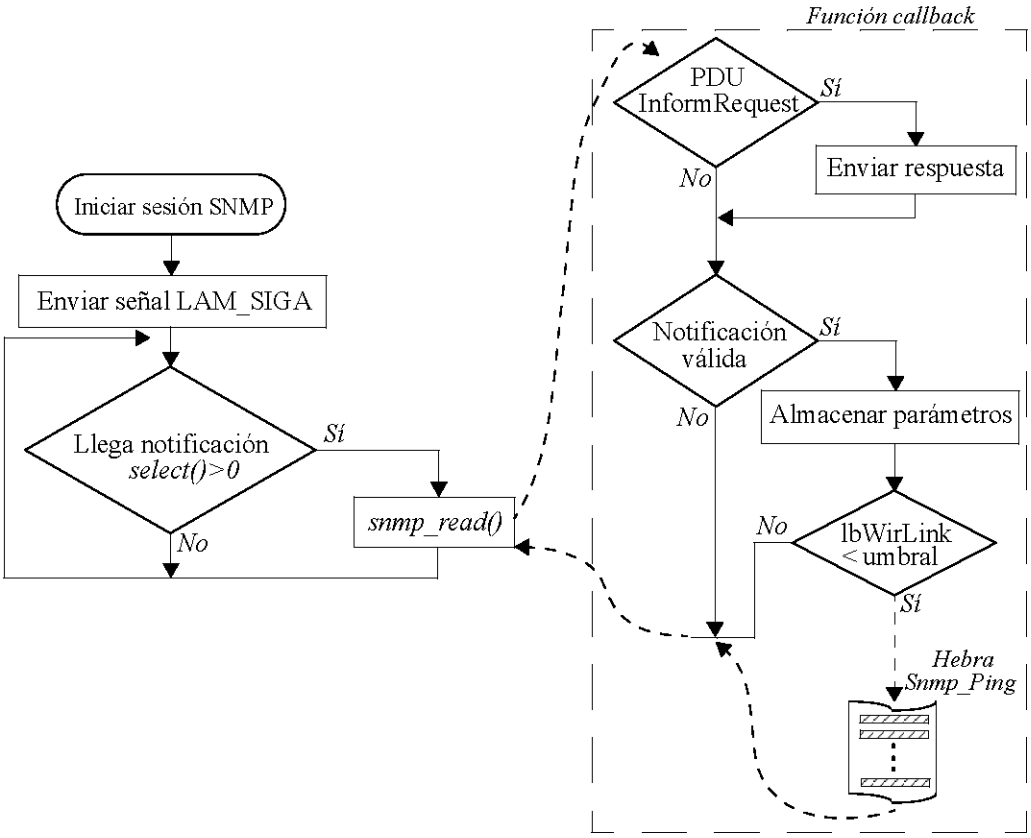


Figura 3.4 Acciones del proceso Gestor

por tanto, el proceso maestro puede utilizar la información recogida por la arquitectura SNMP para realizar distribuciones de carga equilibradas. Para que el proceso *Gestor* pueda enviar la señal, previamente debe vincularse al demonio LAM, proceso *lamd*, que se encuentra en ejecución.

Dado que el envío de notificaciones se realiza de forma totalmente asíncrona, el proceso *Gestor* queda en estado suspendido, y por tanto, no consume recursos del procesador hasta que se recibe un mensaje por el puerto especificado. Esta acción se realiza cuando se invoca la rutina *select()* del lenguaje C. Si el valor retornado por esta función es mayor que cero, entonces se invoca a la función *snmp_read()* para leer el mensaje recibido, y automáticamente, se invoca la función *callback* con los párametros adecuados para procesar el mensaje. Si el valor devuelto por la función *select()* es cero es indicativo de que ha expirado el *timeout* de la función, y si es negativo es indicativo de

que ha ocurrido un error. En ambos casos, se vuelve a invocar dicha función para iniciar otra vez la espera de mensajes.

En el caso de recibir un mensaje, al comienzo de la función *callback* se comprueba si la notificación recibida se corresponde con una *PDU InformRequest*. En caso de recibir un mensaje de este tipo, automáticamente se genera una PDU de respuesta hacia a la entidad que envió la notificación para indicarle su recepción. Esta acción se debe realizar para cumplir con las especificaciones del protocolo SNMP en cuanto a la *PDU InformRequest* se refiere. Una vez realizado este paso, se comprueba si la trama de datos recibida pertenece a alguna de las notificaciones definidas en la base de datos de información de gestión LBGAC-MIB, detalladas en la tabla 3.2. Si el mensaje recibido no concuerda con ninguna de las notificaciones definidas, la función *callback* finaliza, y el proceso *Gestor* vuelve al estado suspendido a la espera de la llegada de otro mensaje. En caso contrario, se extrae la información adjunta a cada notificación y se almacena en la memoria común existente entre el proceso *Gestor* y el proceso maestro de la aplicación paralela-distribuida. Para evitar la inconsistencia de datos durante el acceso a la zona de memoria compartida, este acceso se realiza en exclusión mutua utilizando un semáforo, denominado *Sem_Gn*. La estructura de esta memoria compartida se explica en el apartado 3.4.1.

Otra de las tareas que realiza el proceso *Gestor* es controlar los computadores portátiles que están situados dentro de una zona de cobertura reducida [SMS05a]. Para ello, cada vez que se reciba alguna notificación de los tipos *lbnAppStart, lbnDescLink, lbnCriticalArea* y *lbnUpLink*, donde el valor de *lbWirLink* adjunto está por debajo de un cierto umbral, el proceso *Gestor* crea y lanza una hebra de ejecución, denominada *Snmp_Ping*, por cada computador portátil que esté en esta situación. Cada hebra verifica el estado del canal de la WLAN existente entre el computador maestro y el computador portátil. La verificación consiste en implementar una operación SNMP *GetRequest,* la cual se invoca periódicamente, cada segundo, para consultar el valor del objeto *lbWirLink* que monitoriza el proceso *Colector* ubicado en el computador desde donde se envió la notificación. Cada hebra permanece en ejecución mientras el computador

portátil está ubicado dentro del área de cobertura reducida. Esto es, la hebra finaliza cuando el nivel de la calidad del enlace inalámbrico esté por encima del umbral (área de cobertura aceptable), o cuando no hay enlace de red entre ambos computadores (expira el tiempo de espera de cinco consultas *GetRequest* consecutivas). El tiempo de espera puede expirar debido a que el dispositivo no está accesible (computador apagado, fuera de cobertura, proceso *Colector* no iniciado, ...) o debido a condiciones de la red (congestión, pérdida de paquetes,...) ajenas al dispositivo. Por esta razón, el valor del tiempo de espera se incrementa de forma dinámica si en la consulta anterior no se obtuvo respuesta. El factor utilizado para incrementar el tiempo de espera es dos veces la latencia de red (2 * $t_{lat}(c_i)$). De esta forma, el tiempo de espera para el computador c_i se determina con la siguiente métrica:

$$timeout(c_i)_{inicial} = 2 \times t_{lat}(c_i) + \frac{(sizeof(sent) + sizeof(recv))}{B} + t_{get}(c_i) \qquad (3.2)$$

$$timeout(c_i)_j = 2 \times t_{lat}(c_i) + timeout(c_i)_{j-1} \quad \text{si la petición j-1 falló}$$

Donde:

- $t_{lat}(c_i)$ es la latencia de red entre el computador donde está ubicado el proceso *Gestor* y el computador portátil donde se ejecuta el proceso *Colector* que envió la notificación. Por simplicidad, se asume que este valor es igual para ambos sentidos de comunicación, y se corresponde con el parámetro *lbLatency* del computador esclavo correspondiente.
- *sizeof (sent)* es el tamaño de una operación SNMP *GetRequest*. En nuestro entorno de computación tiene un tamaño de 92 bytes.
- *sizeof (recv)* es el tamaño de una operación SNMP *Response*. En nuestro entorno de computación tiene un tamaño de 93 bytes.
- B es la tasa transferencia de datos existente entre los computadores mencionados. Por simplicidad, se asume que este valor es igual para ambos

sentidos de comunicación, y se corresponde con el parámetro *lbLamdSpeed* del computador esclavo correspondiente.

- $t_{get}(c_i)$ es el tiempo consumido por el proceso *Colector* en decodificar la consulta, obtener el parámetro y construir la operación de respuesta al proceso *Gestor*. Este valor depende de varios factores como son: la velocidad del procesador, tamaño de la memoria y carga del sistema. Sin embargo, al ser una operación sencilla, su valor no difiere de forma apreciable entre diferentes máquinas, y por tanto, este valor se calcula de forma empírica. Se han realizado muchos experimentos sobre una consulta *GetRequest*, y se ha concluido que $t_{get}(c_i)$ tiene un valor aproximado de 1 ms en nuestro entorno de computación.

Una vez la hebra finaliza, ésta escribe en la memoria compartida el estado del computador portátil: *online* u *offline*. El estado de los computadores portátiles se consulta en algunas primitivas del protocolo de equilibrio de carga para informar al proceso maestro de aquellos procesos que se desarrollan en computadores que no están accesibles. De esta forma, mientras se obtenga una respuesta a cada consulta SNMP, se tiene la certeza de que el computador donde se desarrollan los cálculos paralelos está accesible a nivel de red, y no tiene problemas en el nivel de aplicación debido a que el proceso *Colector* responde a las consultas realizadas. En la figura 3.5 se puede apreciar el esquema de funcionamiento de la hebra de ejecución *Snmp_Ping*.

3.3 Protocolo de equilibrio de carga dinámico

Las aplicaciones paralelas que siguen el paradigma Maestro/Esclavo con dependencias de datos entre iteraciones y donde los cálculos paralelos son realizados dentro de cada iteración, deben controlar de forma estricta la distribución de carga para maximizar la eficiencia del entorno de computación, y así, minimizar el tiempo de ejecución de la aplicación. Es por ello, por lo que a continuación se muestra un protocolo que determina la distribución adecuada de carga a comunicar a cada proceso que colabora en la solución de una aplicación paralela-distribuida con las características mencionadas.

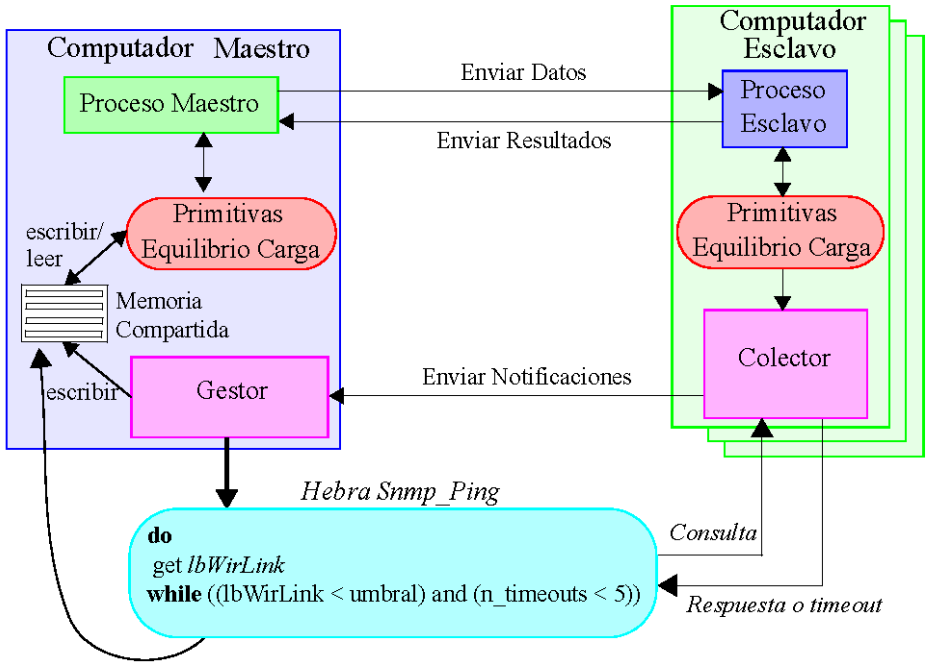


Figura 3.5 Esquema de la hebra Snmp_Ping

Dicha distribución minimiza el estado ocioso de los procesos, obteniéndose de esta forma una solución global más rápida. Además, se presentan las primitivas que deben llevar a cabo los diferentes tipos de computadores del entorno híbrido de computación para cumplir con este protocolo.

3.3.1 Especificación del protocolo

En este trabajo de investigación, se ha diseñado un esquema novedoso de equilibrio de carga dinámico que tiene en cuenta la variación del número de procesos que intervienen en el cálculo paralelo en tiempo de ejecución, y la heterogeneidad presente a nivel de computación y comunicación. Además, debido a que existe un trabajo previo [Mac01], realizado en nuestro grupo de investigación, que controla y gestiona la variación dinámica de recursos en entornos LAN-WLAN, el nuevo protocolo de equilibrio de carga dinámico se ha integrado en el *middleware LAMGAC*, basado en MPI, fruto del mencionado trabajo, para añadirle a éste nuevas funcionalidades.

A continuación, se presentan las ideas del nuevo protocolo de equilibrio de carga dinámico. En todo protocolo intervienen entidades, que pueden ser elementos lógicos o físicos, que comunican información de datos y control [Hol91]. Estas entidades realizan unas operaciones básicas de sincronización de acciones, también llamadas primitivas del protocolo. El programador de aplicaciones paralelas puede aprovechar la funcionalidad del protocolo haciendo uso de estas primitivas.

Teniendo en cuenta que en este trabajo se utiliza la misma arquitectura del entorno de computación para el cual fue desarrollado el *middleware LAMGAC*, en nuestro protocolo también existen tres entidades que se corresponden con los diferentes tipos de computadores que pueden formar el entorno de computación. En estas entidades se desarrolla la ejecución de los procesos paralelos. Por tanto, las entidades del protocolo son tres:

- *Entidad Maestra (EM)*, que controla el estado y rendimiento de las otras entidades, y es donde se desarrolla el proceso maestro de la aplicación (proceso MPI con rango 0, P0). De este tipo de entidad sólo existe una.
- *Entidad Esclava Fija (EF)*, que se comunica con otras entidades de su mismo tipo o con entidades de tipo diferente, y es donde se desarrollan los procesos esclavos que inicialmente forman parte de la aplicación paralela. Los procesos esclavos que se ejecutan en este tipo de entidades son los mismos al iniciar y al finalizar el programa paralelo. Este tipo de entidad hace referencia a un recurso de computación con una interfaz de red basada en el estándar IEEE 802.3 para la comunicación con otras entidades. De este tipo de entidad existen NA ejemplarizaciones iguales desde el punto de vista del protocolo.
- *Entidad Esclava Portátil (EP)*, que se comunica con otras entidades a través del punto de acceso, y es donde se desarrollan los procesos esclavos que son expandidos en tiempo de ejecución en dichas entidades mediante la utilización de las primitivas adecuadas del *middleware LAMGAC*. Los procesos que se ejecutan en estas entidades son controlados desde la entidad EM. Este tipo de entidad hace referencia a un recurso de computación que utiliza una interfaz de

red basada en el estándar IEEE 802.11 para la comunicación con otras entidades. De este tipo de entidad existen NI ejemplarizaciones iguales en un momento dado.

Las primitivas de nuestro nuevo protocolo permiten controlar en tiempo de ejecución: el rendimiento y el estado de las entidades, tanto de las entidades EF como de las entidades EP, estimar la cantidad adecuada de carga a distribuir a cada proceso esclavo para equilibrar los tiempos de ejecución, minimizando así el tiempo ocioso de los procesadores, monitorizar la batería y la calidad del enlace inalámbrico de las entidades EP y controlar aquellas entidades situadas en zonas de cobertura reducida. Las primitivas diseñadas en el protocolo de equilibrio de carga dinámico son las siguientes:

- *Iniciar_EM*. Esta primitiva se invoca en la entidad EM y desde el proceso maestro de la aplicación paralela-distribuida. La llamada a esta primitiva implica la creación del proceso *Gestor*. Por tanto, la entidad EM queda preparada para recibir las notificaciones y almacenar los parámetros acerca del estado y rendimiento del resto de entidades en una zona de memoria compartida entre el proceso maestro y el proceso *Gestor*.

- *Iniciar_EF*. Esta primitiva se invoca desde los procesos que son lanzados en las entidades EF, y su ejecución permite enviar una notificación a la entidad EM. Esta notificación le indica al proceso maestro de la aplicación paralela-distribuida que se ha iniciado un proceso MPI para el cual se puede estimar la carga a distribuir en cada iteración con la primitiva *Equilibrar*. La notificación que se envía lleva adjunto parámetros acerca del rendimiento y estado actual de la entidad EF.

- *Iniciar_EP*. Esta primitiva se invoca desde los procesos que son lanzados en las entidades EP, y su ejecución permite enviar una notificación a la entidad EM. Esta notificación le indica al proceso maestro que se ha iniciado un proceso MPI para el cual se puede estimar la carga a distribuir en cada iteración de la aplicación paralela con la primitiva *Equilibrar*. La notificación que se

envía lleva adjunto parámetros acerca del rendimiento y estado actual de la entidad EP.

- *Enviar_Notificación.* Las entidades EF y EP envían notificaciones a la entidad EM cuando algún evento significativo ocurre en ellas, como puede ser, un incremento considerable del número de procesos en la cola *run* del sistema, una variación de la calidad del enlace inalámbrico, disminución del nivel de batería en los computadores portátiles, inicio de un proceso MPI vinculado al protocolo de equilibrio de carga, etc. El proceso *Colector* de la arquitectura SNMP se encarga de invocar esta primitiva. De esta forma se consigue un solapamiento entre las acciones del proceso paralelo y la monitorización y envío de notificaciones acerca del rendimiento y estado de la entidad.

- *Equilibrar.* Esta primitiva se invoca en la entidad EM. Su ejecución permite estimar la cantidad de carga adecuada a distribuir a los procesos ubicados en el resto de entidades de la VM_LAM que están asociados al mecanismo de equilibrio de carga, con el objeto de minimizar el tiempo ocioso de los mismos. Para ello, básicamente se tiene en cuenta la información sobre el rendimiento que ha tenido cada proceso en la iteración previa, y el rendimiento y estado actual de cada computador.

- *Comprobar_Batería_y_Enlace.* Esta primitiva se invoca en la entidad EM para conocer si existen entidades EP fuera del área de cobertura y/o la existencia de computadores portátiles con un nivel bajo de energía en su batería, de forma que de seguir en este estado dichas entidades dejarán de formar parte de la VM_LAM en breve. Con esta primitiva nos adelantamos a las salidas no controladas de los computadores, es decir, preveemos la desconexión inesperada de las entidades EP, evitando fallos en las comunicaciones a nivel del programa paralelo MPI.

- *Almacenar_Info.* Cada vez que se reciben los resultados de cada proceso paralelo es necesario invocar a esta primitiva para calcular y almacenar el tiempo de ejecución empleado por el proceso en la iteración, así como la carga

efectiva computada por cada proceso. El tiempo de ejecución de cada proceso contempla: el tiempo empleado en recibir los datos, en realizar los cálculos y en enviar los resultados obtenidos. Esta información se utiliza en la primitiva *Equilibrar* para estimar el volumen de datos a distribuir en la siguiente iteración. Por tanto, esta primitiva se invoca en la entidad EM.

- *Finalizar_Cooperante*. Esta primitiva se invoca en los procesos esclavos que se ejecutan en las entidades EF y en las entidades EP, y su ejecución permite enviar una notificación a la entidad maestra indicándole que a partir de ese instante el proceso no desea participar en la distribución de carga obtenida con la primitiva *Equilibrar*.
- *Finalizar*. El proceso maestro finaliza con el atendimiento de notificaciones y la liberalización de la memoria compartida. Por tanto, esta primitiva se invoca en la entidad EM.

En la figura 3.6 se muestra un ejemplo de invocación de las primitivas del protocolo de intercambio de mensajes entre las tres entidades. Para explicar con mayor claridad el intercambio de mensajes, supongamos que la VM_LAM permanece igual desde el comienzo de la aplicación hasta el final, esto es, no se produce ninguna vinculación o desvinculación de algún computador portátil durante el transcurso de la ejecución mostrada. En esta figura, el proceso en la entidad EM, después de invocar a la primitiva *Iniciar_EM*, espera la llegada de notificaciones del tipo *lbnAppStart*, tanto desde las entidades EF como de las entidades EP, que indican el comienzo de un proceso MPI que quiere equilibrar su tiempo de ejecución con el resto de procesos. Esta notificación se envía al invocar la primitiva *Enviar_Notificación*, la cual se llama desde las primitivas *Iniciar_EP* e *Iniciar_EF*. Adjunta a cada notificación recibida hay información relevante acerca del rendimiento del computador desde donde se envió. Esta información se almacena en la memoria compartida existente en la entidad EM y se utiliza para estimar la distribución de carga inicial. En algún punto de cada iteración del programa paralelo y antes de realizar la distribución de carga, se ha de invocar a la primitiva *Equilibrar* para

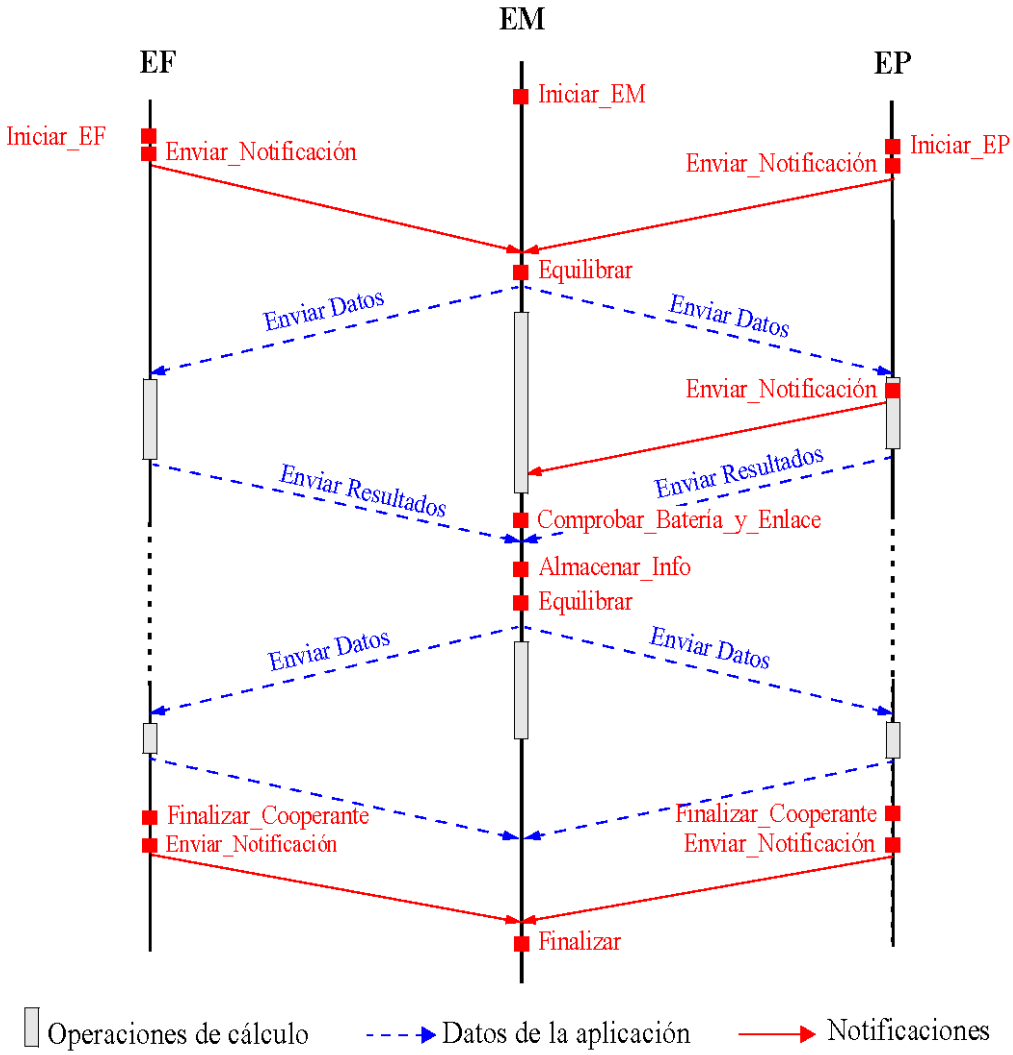


Figura 3.6 Invocación de primitivas e intercambio de mensajes entre entidades

estimar la cantidad de carga a enviar a cada proceso. Una vez realizada la distribución de datos, los procesos esclavos en las entidades EF y en las entidades EP llevan a cabo sus tareas de cálculo. Antes de comenzar la recepción de los resultados calculados por los procesos paralelos, la entidad EM puede invocar la primitiva *Comprobar_Batería_y_Enlace* para conocer la existencia de aquellas entidades que pueden desaparecer temporal o definitivamente de la VM_LAM. En el caso de existir uno o varios procesos en esta última situación, el programador no debería implantar la

recepción de resultados para éstos. La primitiva *Almacenar_Info* tiene que ser llamada cada vez que se reciben los resultados obtenidos por cada proceso, con el fin de registrar el tiempo de ejecución empleado y la carga efectiva computada por cada proceso en la iteración actual. Esta secuencia se repite en todas las iteraciones.

Durante el transcurso de cada iteración podría ocurrir que en algún computador se produzca un evento significativo, y por tanto, una notificación acorde al evento producido se envía a la entidad EM, primitiva *Enviar_Notificación*. Esta notificación, y sus parámetros adjuntos, se tendrá en cuenta en la próxima invocación de la primitiva *Equilibrar* para estimar la carga a distribuir al proceso que se está desarrollando en el computador desde donde se envió la notificación. Nótese que existe un doble solapamiento en las entidades que participan en los cálculos paralelos. Por un lado, hay un solapamiento entre las operaciones de decodificación de las notificaciones recibidas en la entidad EM, proceso *Gestor*, y las tareas de cálculo y distribución de datos realizadas en ésta, proceso maestro. De esta forma, este último proceso puede realizar acciones propias del programa paralelo sin tener que permanecer a la espera de posibles notificaciones. Por otro lado, en las entidades EF y EP, hay un solapamiento entre la ejecución de los procesos esclavos, y la monitorización de parámetros y envío de notificaciones, proceso *Colector*. De esta forma, también los procesos esclavos realizan las tareas de cálculo y comunicación con el proceso maestro sin dedicar tiempo a controlar el estado de la entidad.

Por último, cuando los procesos esclavos se van a desvincular del mecanismo de equilibrio de carga dinámica invocan a la primitiva *Finalizar_Cooperante* provocando que se envíe una notificación desde esas entidades. Además, el proceso maestro llama la primitiva *Finalizar* para no atender más notificaciones y no realizar distribuciones de datos equilibradas.

3.4 Implementación de las operaciones del protocolo

En el apartado anterior se ha puesto al descubierto una serie de operaciones del protocolo que deben invocar los procesos de la aplicación paralela para llevar a término un

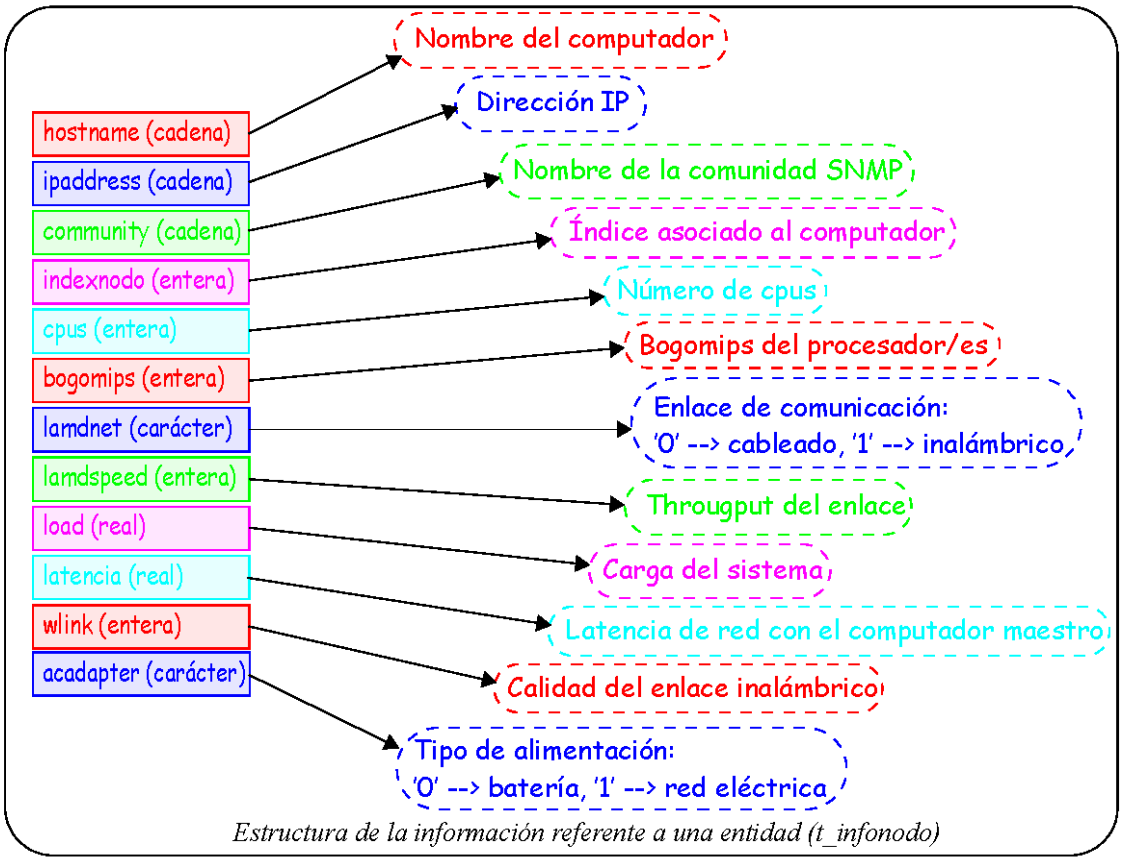
equilibrio de carga dinámico efectivo. En este apartado se presenta el diseño de estas operaciones.

3.4.1 Primitiva Iniciar_EM

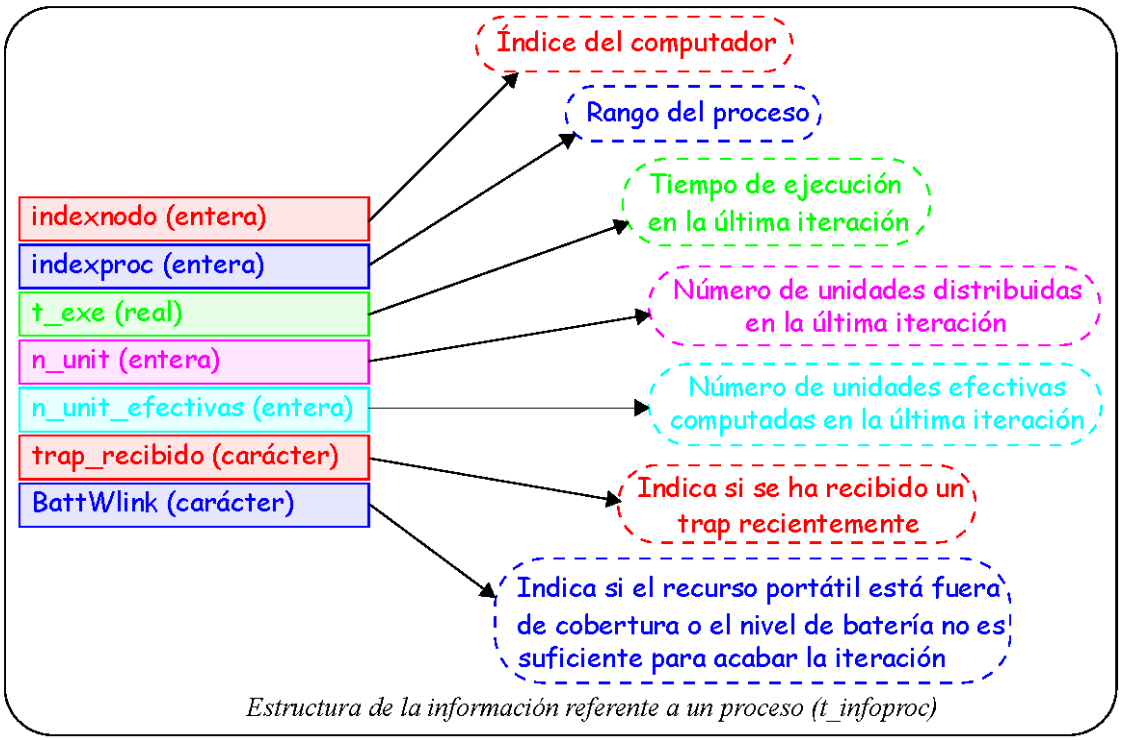
Para realizar esta operación, el proceso maestro que se desarrolla en la entidad EM crea un proceso (también podría crear una hebra de ejecución) encargado de recibir las notificaciones SNMP, almacenar la información acerca del rendimiento de las entidades que forman parte de la máquina paralela virtual, y controlar las entidades EP situadas en zonas de cobertura reducida. Con la creación de este proceso, al que hemos denominado proceso *Gestor*, se evita que el proceso maestro de la aplicación paralela sea interrumpido cada vez que se recibe una notificación desde alguna entidad. Nótese la importancia de delegar en este proceso, puesto que el proceso maestro puede estar realizando acciones propias del programa paralelo a la vez que el proceso *Gestor* espera, decodifica, almacena la información contenida en las notificaciones recibidas y controla la disponibilidad de las entidades EP. De esta manera, se solapan las operaciones de obtención de información relativa al rendimiento de los computadores y los cálculos realizados en el proceso maestro.

La información adjunta a las notificaciones tiene que ser almacenada en una zona de memoria que pueda ser accesible tanto por el proceso maestro como por el proceso *Gestor*. Es por ello, que al invocar esta primitiva, se establece una zona de memoria compartida para almacenar y compartir la información referente a los computadores y procesos esclavos. El acceso a esta zona de memoria común se realiza en exclusión mutua, garantizándose de esta forma la consistencia de los datos almacenados en ella. Para garantizar la exlusión mutua y para sincronizar la ejecución de instrucciones de la sección crítica se emplea un semáforo binario (*Sem_Gn*), inicialmente con valor igual a 1, el cual también se crea en la llamada a esta primitiva.

En la figura 3.7 se muestra la estructura de la memoria creada. La estructura de datos *t_infonodo* almacena la información de cada entidad, y la estructura de datos *t_infoproc* almacena la información referente a cada proceso esclavo.



Estructura de la información referente a una entidad (t_infonodo)



Estructura de la información referente a un proceso (t_infoproc)

Figura 3.7 Información almacenada en la memoria compartida

Antes de finalizar la ejecución de la primitiva Iniciar_EM, se inicia un manejador para atender la señal LAM_SIGA. El único proceso que puede enviar una señal del tipo LAM_SIGA al proceso maestro es el proceso *Gestor*, y éste lo realiza sólo una vez y cuando está preparado para recibir notificaciones. Cuando el proceso maestro recibe esta señal, éste desbloquea a los procesos esclavos que se desarrollan en la entidades EF que inicialmente comenzaron la ejecución de la aplicación paralela. De esta forma, se evita que alguna notificación enviada desde cualquier entidad esclava no pueda ser capturada, por no estar el proceso *Gestor* preparado para la recepción.

En la figura 3.8 se muestra un gráfico con el que se pretende aclarar las tareas realizadas al invocar la primitiva *Iniciar_EM* (en la figura, esta llamada se representa con 1). Como se puede apreciar, el proceso *Gestor* es un proceso que necesita estar vinculado al demonio *lamd* de la distribución LAM/MPI (2), y su objetivo es permanecer a la espera de recibir alguna notificación. Una vez que el proceso está preparado para recibir notificaciones, éste envía una señal del tipo LAM_SIGA al proceso maestro (3) para indicarle dicho estado. Cuando se recibe una notificación, la función de *callback* se invoca automáticamente (4), y en ella se realiza el proceso de decodificación y almacenamiento de la información recibida en la memoria compartida. Para acceder en exclusión mutua a la memoria compartida, la función hace uso del semáforo *Sem_Gn* (5). Una vez procesada la notificación recibida, el proceso *Gestor* pasa al estado de escucha (6), esperando por la llegada de una nueva notificación. Si alguna de las notificaciones recibidas se corresponde con una de las notificaciones del tipo *lbnAppStart, lbnDescLink, lbnCriticalArea* o *lbnUpLink,* y la calidad del enlace inalámbrico que posee la entidad desde donde se envió la notificación está por debajo de un cierto umbral, se lanza una hebra de ejecución (7), hebra *Snmp_Ping*, que periódicamente consulta el objeto *lbWirLink* en el proceso *Colector* de la entidad remota. Al finalizar la hebra, ésta escribe en la memoria compartida la disponibilidad de la entidad EP monitorizada. Por otro lado, cuando las primitivas desarrolladas son invocadas desde el proceso maestro, éstas también hacen uso del semáforo *Sem_Gn* para acceder en exclusión mutua a la memoria compartida.

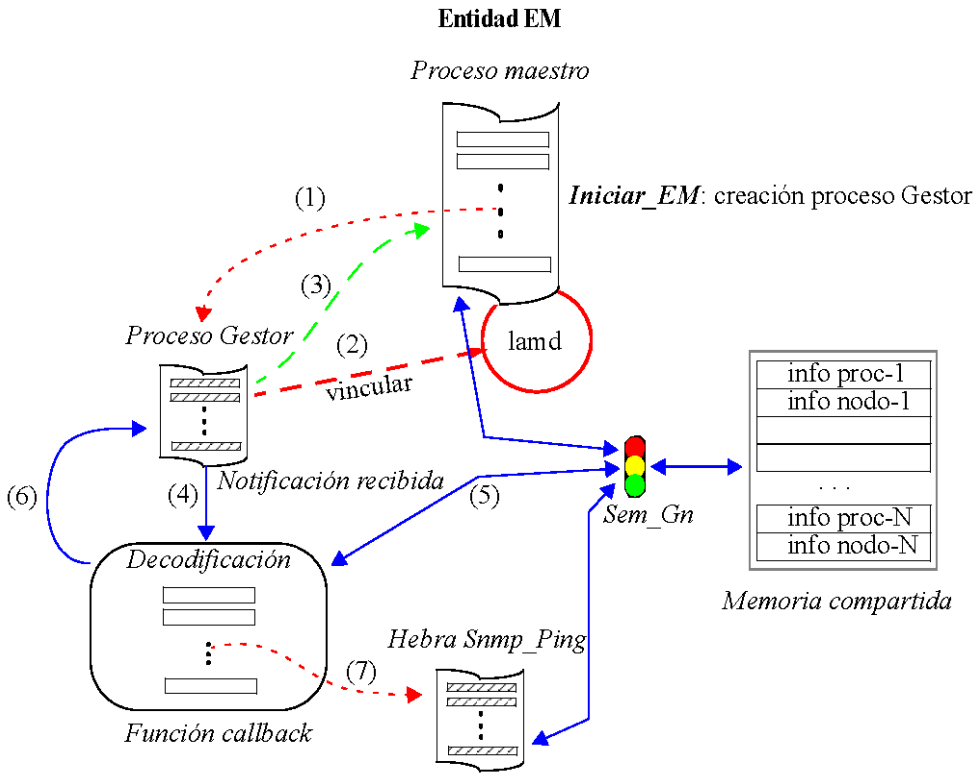


Figura 3.8 Acciones de la primitiva Iniciar_EM

3.4.2 Primitiva Iniciar_EF

Esta primitiva obtiene el rango del proceso que la invoca llamando a la función *MPI_Comm_rank()* y luego, el proceso se bloquea llamando a la función *MPI_Barrier()* de MPI. Una vez que el proceso *Gestor* está preparado para recibir notificaciones, se envía la señal LAM_SIGA al proceso maestro, y los procesos esclavos que se desarrollan en las entidades EF son desbloqueados. A continuación, se inicia una sesión SNMP para actualizar el valor del objeto *lbProcStart*, definido en la LBGAC-MIB del agente local, con el rango del proceso. La actualización de este objeto provoca que el proceso *Colector* invoque a la primitiva *Enviar_Notificación* para enviar una notificación del tipo *lbnAppStart* al proceso *Gestor*. De esta forma, se le indica que un nuevo proceso ha comenzado su ejecución y quiere formar parte de las distribuciones de carga estimadas por la primitiva *Equilibrar*.

3.4.3 Primitiva Iniciar_EP

Esta primitiva recibe del proceso maestro el número de procesos esclavos que se ejecutan en las entidades EF y en las entidades EP, además del rango del proceso dentro de la ejecución de la aplicación paralela-distribuida. El valor del rango es calculado por el proceso maestro, y viene dado por la siguiente expresión: *NA + indice + 1*, donde *NA* es el número de procesos en las entidades EF e *indice* es el número de procesos esclavos que se ejecutan en entidades EP antes de realizar la expansión de dicho proceso. A continuación, se inicia una sesión SNMP para actualizar el valor del objeto *lbProcStart* definido en la LBGAC-MIB del agente local con el rango del proceso. La actualización de este objeto provoca que el proceso *Colector* invoque a la primitiva *Enviar_Notificación* para enviar una notificación del tipo *lbnAppStart* al proceso *Gestor*. De esta forma, se le indica que un nuevo proceso ha comenzado su ejecución y quiere formar parte de las distribuciones de carga estimadas por la primitiva *Equilibrar*.

3.4.4 Primitiva Enviar_Notificación

Tanto en las entidades EF como en las entidades EP se utiliza el proceso *Colector* para enviar notificaciones al proceso *Gestor* cuando se produce algún evento significativo. La descripción de las notificaciones que pueden ser enviadas fueron descritas en la sección 3.2.1, tabla 3.2. Salvo las notificaciones *lbnAppStart* y *lbnAppFinish*, las cuales son enviadas por invocación directa de las primitivas *Iniciar_EF* o *Iniciar_EP* y *Finalizar_Cooperante*, respectivamente, el resto de notificaciones son enviadas cuando su parámetro relacionado supera un umbral prefijado o simplemente informan de un cambio producido. Por tanto, el proceso *Colector* envía dichas notificaciones automáticamente y de forma asíncrona; de esta forma se consigue un solapamiento en las entidades esclavas entre la ejecución de los cálculos paralelos y, la monitorización de la entidad y la notificación de eventos. El programador no puede realizar manualmente el envío de alguna notificación, ya que la arquitectura SNMP es transparante y no accesible al usuario.

3.4.5 Primitiva Equilibrar

Con la invocación de esta primitiva, se obtiene la distribución de carga óptima a realizar para que los resultados obtenidos por todos los procesos esclavos sean recibidos simultáneamente en la entidad EM. De esta forma, se minimiza el tiempo en el que dichos procesos permanecen en un estado ocioso. Además, esta primitiva devuelve la secuencia óptima de procesos para realizar el envío de datos. Esta primitiva tiene que ser invocada en cada iteración y antes de realizar la distribución de datos. El proceso maestro accede en exclusión mutua a la memoria que comparte con el proceso *Gestor* a través del semáforo *Sem_Gn* para obtener la información necesaria sobre los procesos y los computadores donde éstos se están desarrollando. Para estimar la distribución de datos, en la memoria compartida debe residir la información de todos los computadores y procesos que realizan los cálculos paralelos, en caso contrario, esta primitiva permanece bloqueada hasta que se reciba la información necesaria. Esta información se recibe con la notificación *lbnAppStart*.

Las acciones que realiza esta primitiva difieren en la primera iteración, y en el resto de iteraciones. Estas acciones se describen a continuación.

Primera iteración

En la primera iteración de la aplicación paralela-distribuida, el proceso maestro no posee información acerca del rendimiento que tiene cada proceso del programa paralelo en la entidad donde se ha lanzado. Por tanto, la estimación de la cantidad de datos a distribuir en la primera iteración tiene que ser calculada en base a parámetros de rendimiento de los computadores donde se ejecutan dichos procesos. Los valores de estos parámetros se leen de la zona de memoria compartida, los cuales fueron recibidos a través de la notificación *lbnAppStart* enviada al invocar las primitivas *Iniciar_EF* o *Iniciar_EP*.

A continuación detallamos los pasos seguidos para obtener la métrica que devuelve la distibución de datos óptima, en base a los parámetros de rendimiento de las entidades.

Para minimizar el estado ocioso en el que algunos procesos esclavos puedan estar, el tiempo de ejecución en cada iteración de todos los procesos tiene que ser igual. El

tiempo de ejecución de un proceso i (p_i) durante el transcurso de la iteración j puede ser expresado de la siguiente forma, considerando que no hay solapamiento de los cálculos con las comunicaciones:

$$t\_exe_j(p_i) = t\_comm_j(p_i) + t\_calc_j(p_i) + t\_idle_j(p_i) \qquad (3.3)$$

Donde el tiempo de comunicación ($t\_comm_j(p_i)$) es el tiempo transcurrido para enviar los datos desde el proceso maestro al proceso i, y el tiempo transcurrido para enviar los resultados desde el proceso i al proceso maestro. El tiempo de cálculo ($t\_calc_j(p_i)$) es el tiempo real consumido por el proceso para realizar los cálculos. Durante el tiempo ocioso ($t\_idle_j(p_i)$) no se realizan cálculos ni comunicaciones. Por tanto, el tiempo de comunicación y de cálculo debe ser ajustado de tal forma que el tiempo ocioso sea minimizado. En la figura 3.9 se puede apreciar de forma gráfica la definición de los tiempos.

El tiempo de comunicación y el tiempo de cálculo puede ser estimado en función de parámetros relacionados con el rendimiento de las redes y los computadores, y la cantidad de trabajo a procesar. En un sistema heterogéneo típico, el coste de la comunicación depende tanto de las características o capacidades de comunicación del computador (interfaz de red) como también del rendimiento de la red [BRP99]. Por tanto, el tiempo de comunicación puede ser modelado por la expresión clásica lineal en función de la latencia de comunicación y el volumen de datos a enviar y recibir. En nuestro caso, matemáticamente puede ser expresado como sigue, para cualquier iteración del programa:

$$t\_comm(p_i) = 2 \times t_{lat}(p_i) + \frac{n\_unit \times (sizeof(data\_u) + sizeof(result\_u))}{B} \qquad (3.4)$$

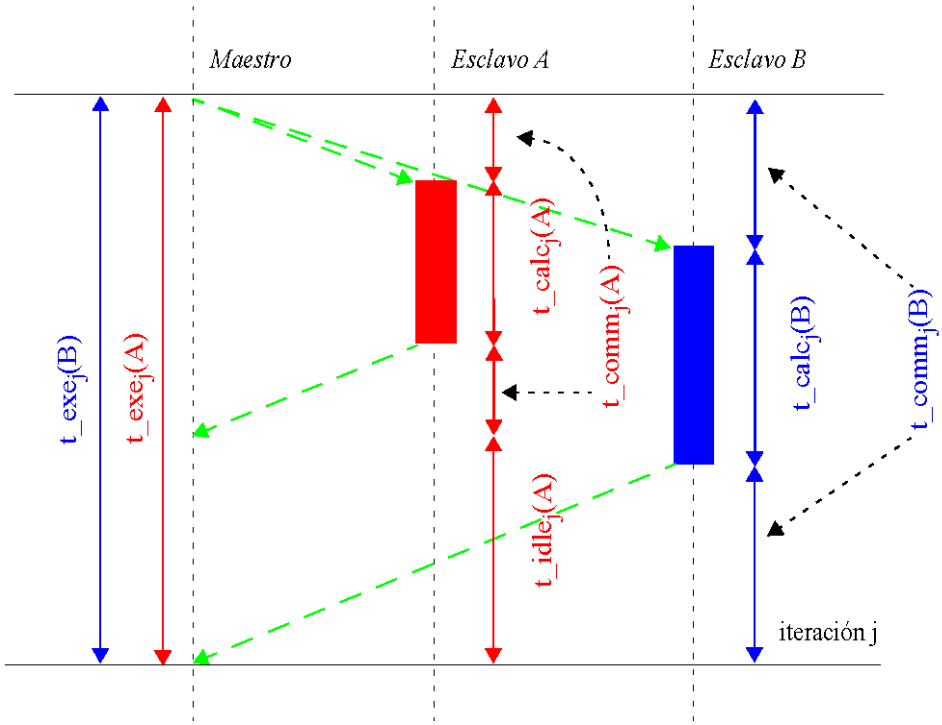


Figura 3.9 Tiempo de ejecución, de comunicación, de cálculo y ocioso

Donde:

- $t_{lat}(p_i)$ es la latencia de red entre la entidad EM y la entidad donde se ejecuta el proceso esclavo *i*. Por simplicidad, se asume que este valor es igual para ambos sentidos de comunicación.
- *n_unit* es el número de unidades de datos a enviar al proceso esclavo. Una unidad de datos es la mínima cantidad de datos enviada para obtener un único resultado, el cual depende de la granularidad de la aplicación. Por ejemplo, si consideramos un resultado parcial como aquel elemento obtenido del producto matriz por vector, una unidad de datos sería una fila de la matriz, suponiendo que el vector está replicado en todos los procesos esclavos.
- *sizeof (data_u)* es el tamaño de una unidad de datos.
- *sizeof (result_u)* es el tamaño de un único resultado (el que se obtiene con los datos recibidos).

- B es la tasa transferencia de datos existente entre la entidad EM y la entidad donde se ejecuta el proceso esclavo i. Por simplicidad, se asume que este valor es igual para ambos sentidos de comunicación.

La métrica 3.4 es válida para aplicaciones en las que existe una relación lineal entre el volumen de datos enviados y los resultados obtenidos. Por eso, en dicha métrica, el parámetro *n_unit* multiplica al tamaño de una unidad de dato enviada y recibida. Sin embargo, en muchas aplicaciones esta relación no se cumple, y por tanto, en estos casos, el tiempo de comunicación definido en función del parámetro *n_unit* sólo se puede modelar de forma aproximada.

Por otro lado, el tiempo de cálculo en cualquier iteración puede ser expresado en función de la carga del procesador, su velocidad y el cálculo a realizar de la siguiente forma:

$$t\_calc(p_i) = \frac{n\_unit \times mips\_calc}{mips\_cpu} \times \frac{load + 1}{cpus} \qquad \text{si (load +1) > cpus}$$

$$t\_calc(p_i) = \frac{n\_unit \times mips\_calc}{mips\_cpu} \qquad \text{si (load +1) <= cpus} \qquad (3.5)$$

Donde:

- *mips_calc* es el número de millones de instrucciones máquinas ejecutadas en un proceso esclavo para procesar una unidad de datos y obtener un único resultado.
- *mips_cpu* representa la potencia de cálculo del procesador donde se ejecuta el proceso esclavo i y tiene por valor el número de millones de instrucciones por segundo que es capaz de ejecutar (MIPS).
- *cpus* indica el número de procesadores disponibles en la entidad donde se ejecuta el proceso esclavo i.
- *load* es el número medio de trabajos en el último minuto en la cola *run* del sistema donde se ejecuta el proceso esclavo i.

La expresión *(load+1)/cpus* se aplica si el numerador es mayor que el número de procesadores que posee la entidad donde se ejecuta el proceso p_i. Esta relación es necesaria debido a que el entorno de computación LAN-WLAN no está dedicado exclusivamente para la ejecución de los programas paralelos.

Partiendo de las métricas 3.3, 3.4 y 3.5 se deduce que el número óptimo de unidades de datos a enviar a cada proceso en la iteración j viene dado por la siguiente expresión:

$$n\_unit_j(p_i) = \frac{t\_exe\_est - (2 \times t_{lat}(p_i))}{\frac{mips\_calc}{mips\_cpu} \times \left\lceil \frac{load+1}{cpus} \right\rceil + \frac{sizeof(data\_u) + sizeof(result\_u)}{B}} \qquad (3.6)$$

Donde *t_exe_est* es el tiempo de ejecución calculado, según la ecuación 3.3, de un proceso que se desarrolla en el primer computador del que se tiene información en la memoria compartida, y donde se procesa un volumen de datos equivalente a una distribución de carga equitativa entre todos los procesos existentes al comienzo de la aplicación paralela-distribuida. Debido a que el volumen de datos a distribuir a cada proceso, obtenido con la ecuación 3.6 se calcula para un tiempo de ejecución *t_exe_est* calculado con la métrica 3.3, el cual generalmente no va coincidir con el tiempo de ejecución real de la iteración, el volumen definitivo a distribuir se obtiene al ponderar los datos de la métrica 3.6 sobre la cantidad total de datos a distribuir en la iteración.

Para aplicar esta métrica, los párametros *mips_cpu, load, cpus,* t_{lat} y B se obtienen a partir de los datos recopilados en cada recurso de computación, debido a que éstos dependen de las características físicas y del rendimiento de cada computador. Para ello, se utiliza la arquitectura basada en SNMP presentada en la sección 3.2, existiendo una correspondencia única entre estos parámetros y los objetos *lbBogomips, lbLoad, lbCpus, lbLatency* y *lbLamdSpeed* de LBGAC-MIB.

Además, como en la primera iteración no existe información del rendimiento previo de la ejecución de los procesos, la secuencia de distribución de datos fijado para la primera iteración se establece desde aquellos procesos que se ejecutan en los computadores más rápidos hasta los que se ejecutan en los computadores más lentos.

Esto es, al primer proceso al que se le envían los datos es aquel que se desarrolla en el computador cuyo parámetro *lbBogomips* es mayor.

Resto de iteraciones

En el resto de iteraciones del programa paralelo, y salvo vinculación al entorno de un nuevo computador, se dispone de información sobre el tiempo de ejecución empleado y el número de unidades de datos computadas por cada proceso en la iteración previa. Por tanto, la estimación de carga a distribuir a cada proceso se calcula en base al rendimiento previo de cada uno, es decir, se toma como referencia el tiempo de ejecución de cada proceso en la iteración previa para decidir la siguiente distribución de carga, al igual que se realiza en [ZLP97][SMS03c]. Para ello, la primitiva *Equilibrar* realiza los tres pasos siguientes, los cuales se muestran en la figura 3.10.

Inicialmente, en el paso 1, se estima el rendimiento que ha obtenido cada proceso en la iteración previa mediante la siguiente métrica:

$$rend_j(p_i) = \frac{n\_unit\_efectivas_{j-1}(p_i)}{t\_exe_{j-1}(p_i)} \qquad (3.7)$$

Donde $n\_unit\_efectivas_{j-1}(p_i)$ es el número de unidades de datos procesadas por el proceso i, y $t\_exe_{j-1}(p_i)$ es el tiempo de ejecución del proceso. Ambos parámetros se estiman al finalizar la iteración previa.

El valor obtenido para cada proceso i en la iteración j ($rend_j(p_i)$) nos indica cuánto de eficaz es cada proceso para la aplicación concreta que se ejecuta en el computador utilizado y en la iteración previa. El cálculo del rendimiento de los procesos establece el orden o secuencia de envío de datos en la siguiente distribución de carga. Esta secuencia se establece en un orden descendente, esto es, de mayor a menor rendimiento. Por tanto, los datos se envían primero a los procesos con mayor rendimiento.

A continuación, en el paso 2, se estima si se ha alcanzado el equilibrio o no en la iteración previa. Esto se realiza de la siguiente forma: primero, se calcula la media del

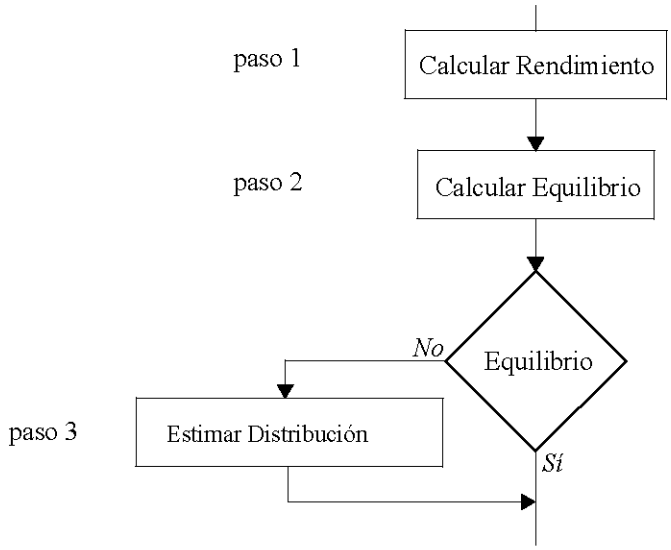


Figura 3.10 Diagrama de flujo de la primitiva Equilibrar en cualquier iteración excepto la primera

tiempo de ejecución de los procesos. Luego, si los resultados de todas las diferencias calculadas entre este valor medio y el tiempo de ejecución de cada proceso están dentro de un margen (porcentaje fijado por el programador), entonces se considera que en la iteración previa se ha alcanzado el equilibrio en los tiempos de ejecución de todos los procesos. Es decir, la carga comunicada a cada proceso implica que todos los procesos finalicen dentro del margen establecido. Por tanto, si no hay variaciones en la VM_LAM y no se producen eventos significativos en algún computador, podemos predecir que los tiempos de ejecución de los procesos en la iteración actual serán similares a los de la iteración previa si la distribución de carga es la misma que la realizada en la última distribución. De este modo, siempre que se alcance el equilibrio en la iteración previa, en la iteración actual se distribuye la misma proporción de carga que la realizada en la iteración anterior.

Por otro lado, en el caso de que en la iteración previa no se alcance el equilibrio en los tiempos de ejecución, se debe llevar a cabo una nueva estimación de carga para cada proceso (paso 3). Debido a que el objetivo de la primitiva *Equilibrar* es que el tiempo de ejecución de todos los procesos sean similares o estén dentro de un margen establecido,

el número de unidades de datos a distribuir se calcula en base al tiempo de ejecución medio de todos los procesos. La métrica empleada es la siguiente:

$$n\_unit_j(p_i) = \mu\_t\_exe_{j-1} \times rend_{j-1}(p_i) \qquad (3.8)$$

Donde $\mu\_t\_exe_{j-1}$ es la media del tiempo de ejecución de los procesos esclavos. Los dos parámetros hacen referencia a la iteración previa (j-1).

La métrica anterior se aplica para estimar el volumen de datos a distribuir a cada proceso cuando no se produce ningún evento significativo relacionado con el rendimiento de los computadores. Si la entidad EM recibe alguna notificación del tipo *lbnLoad* o *lbnUpLink,* a partir de la última vez que se invocó la primtiva *Equilibrar*, es indicativo de que el rendimiento de la entidad que la envió puede haber variado bruscamente o que una entidad EP está disponible después de haber estado fuera de cobertura, repectivamente. Por tanto, para asumir dicho cambio, la carga a distribuir al proceso que se desarrolla en dicho computador se estima en base a la métrica 3.6, donde el parámetro *t_exe_est* se sustituye por la media de los tiempos de ejecución en la iteración previa $\mu\_t\_exe_{j-1}$.

También, la métrica 3.6 se aplica cuando un nuevo proceso se expande en un nuevo computador durante la ejecución de la aplicación paralela-distribuida, es decir, como el proceso maestro tiene que distribuirle datos por primera vez y no posee información acerca del rendimiento de dicho proceso, la estimación se realiza en base al rendimiento del computador y de la red. Por ello, el volumen de datos a distribuir al nuevo proceso se estima en base al tiempo de ejecución medio de los procesos en la iteración previa $\mu\_t\_exe_{j-1}$.

Debido a que la estimación de carga realizada puede no coincidir con el volumen de datos a distribuir, una vez calculada la cantidad de datos a enviar a cada proceso, ésta se pondera a la cantidad total a distribuir en la iteración.

Por otro lado, durante la evolución de la aplicación paralela-distribuida, la estrategia de equilibrio de carga va ajustando los tiempos de ejecución de todos los procesos al tiempo de ejecución medio. Por tanto, una vez alcanzado el equilibrio, se puede predecir de antemano cuál es el instante de tiempo aproximado a partir del cual comienzan a llegar al proceso maestro los resultados calculados por los procesos esclavos. De esta forma, se puede programar la recepción de resultados justo antes del instante de su llegada. Esta predicción es sólo válida en la ejecución de aplicaciones donde el número de operaciones de cálculo realizadas en cada iteración sea igual. En este sentido, en la primitiva *Equilibrar*, a petición del usuario, se puede programar una señal del tipo SIGALRM para ser enviada al proceso maestro un tiempo después de ser programada. El tiempo tomado para programar dicha señal es el tiempo de ejecución menor de todos los procesos en la iteración previa. El proceso maestro puede permanecer bloqueado si se invoca la primitiva *Comprobar_Batería_y_Enlace* y no se ha recibido la señal. Una vez recibida la señal, continúa la ejecución y se conocerá si existe algún computador situado en una zona donde no hay enlace de comunicación o su nivel de energía en la batería es bajo. Así, se pueden programar las rutinas de recepción de aquellos resultados que se tiene la certeza que serán enviados.

3.4.6 Primitiva Comprobar_Batería_y_Enlace

Debido a que los computadores portátiles que forman parte del entorno de computación pueden desaparecer de forma inesperada de la VM_LAM debido a la falta de energía en la batería o debido a que están ubicados en una zona de cobertura limitada, es necesario conocer esta situación antes de enviar datos a los procesos o recibir los resultados desde éstos.

Para conocer el estado de estos computadores, esta primitiva simplemente comprueba el estado del campo *BattWlink* de la estructura de datos *t_infoproc* (figura 3.7). Un valor igual a uno de este campo indica la existencia de problemas relacionados con la batería o enlace inalámbrico. Este campo se actualiza cuando se comprueba que no es posible la comunicación con la entidad EP, por medio de la hebra *Snmp_Ping*, o

cuando se recibe una notificación del tipo *lbnLowBattery* o del tipo *lbnBatteryStatus* y se estima que la batería se agotará antes de finalizar la iteración. También, una notificación del tipo *lbnAppStart* puede indicar la existencia de un computador con problemas dependiendo de los valores de los parámetros adjuntos referentes a la batería o calidad del enlace inalámbrico. Por tanto, esta primitiva tiene que acceder en exclusión mutua a la memoria compartida a través del semáforo *Sem_Gn* para obtener la información necesaria. Esta primitiva devuelve el número de procesos con problemas de batería y/o cobertura y el rango de cada uno de ellos.

Por otro lado, y dependiendo de la implementación, esta primitiva puede bloquear temporalmente la ejecución del proceso maestro si se programa el envío de la señal SIGALRM en la primitiva *Equilibrar*, y en el momento de invocar la primitiva *Comprobar_Batería_y_Enlace* no se ha recibido la señal.

3.4.7 Primitiva Almacenar_Info

El proceso maestro se encarga de calcular el tiempo de ejecución de cada proceso, es decir, el tiempo transcurrido desde que se comenzó a enviar los datos al primer proceso esclavo hasta que se reciben los resultados de cada uno. Por tanto, esta primitiva se invoca cada vez que se reciben los resultados desde un proceso esclavo. El tiempo de ejecución de cada proceso y el número de unidades sobre las que se realizaron cálculos tienen que ser almacenados para poder estimar el número de unidades a distribuir en la siguiente iteración, métrica 3.8. De esta forma, la primitiva *Almacenar_Info* almacena los tiempos de ejecución y además, la carga efectiva computada por cada proceso.

Para realizar esta tarea, el programador debe tomar una marca de tiempo, *tiempo_inicial*, la cual sirve como referencia de inicio del tiempo de ejecución, antes de iniciar la distribución de datos. Para estimar el tiempo de ejecución de cada proceso, cada vez que se reciban resultados calculados por un proceso, se calcula la diferencia entre el tiempo actual y el *tiempo_inicial*. Respecto a la carga efectiva computada por cada proceso, es el programador el que debe facilitar dicho valor a la primitiva. En el caso de aplicaciones donde la carga distribuida no está correlada con la cantidad de

trabajo realizado, el programador de la aplicación paralela debe modificar el código de los procesos esclavos para calcular el número de unidades que han sido evaluadas y comunicarlas al proceso maestro.

3.4.8 Primitiva Finalizar_Cooperante

Los procesos esclavos invocan la primitiva *Finalizar_Cooperante* cuando quieren dejar de formar parte de un entorno de computación donde la estimación de la carga a distribuir se realiza con el mecanismo de equilibrio de carga. Dicha invocación provoca que se inicie una sesión SNMP para actualizar el valor del objeto *lbProcFinish*, de LBGAC-MIB, con el rango del proceso que la invoca. La actualización de este objeto provoca que el proceso *Colector* invoque a la primitiva *Enviar_Notificación* para enviar una notificación del tipo *lbnAppFinish* al proceso *Gestor*, realizando éste último la eliminación de la información del proceso que invocó la primitiva *Finalizar_Cooperante*, de la memoria compartida existente en la entidad EM.

3.4.9 Primitiva Finalizar

Esta primitiva termina la ejecución del proceso *Gestor* enviándole la señal *SIGKILL*. También libera los recursos compartidos con el proceso maestro, como la información sobre computadores y procesos, así como el semáforo que sincroniza el acceso en exclusión mutua a estos recursos.

4. Biblioteca y esquema de computación

En este capítulo se presentan las nuevas funciones añadidas a la biblioteca LAMGAC que implementan las primitivas del protocolo de equilibrio de carga, así como las modificaciones realizadas a las funciones ya existentes en LAMGAC para permitir la compatibilidad. Por otro lado, debido al control realizado sobre la batería y la calidad del canal inalámbrico, se define un mecanismo de recepción controlada de datos utilizando las nuevas funciones de LAMGAC. Además, se presentan las ideas básicas del modelo de computación propuesto, que consiste en el diseño de tres esqueletos, uno para cada tipo de entidad del entorno de computación. Finalmente, se evalúa la eficiencia del protocolo y el rendimiento de estas funciones en un entorno de computación LAN-WLAN.

4.1 Extensión de la biblioteca LAMGAC

En el capítulo anterior se presentaron las primitivas de nuestro protocolo de equilibrio de carga, las cuales pueden ser aplicadas en programas paralelos en los que existen dependencias de datos entre iteraciones. La implementación de este tipo de aplicaciones con las primitivas desarrolladas es compleja, debido a la combinación de diversos protocolos y mecanismos, como son: el *protocolo de gestión de red SNMP*, los IPC soportados por LINUX [Web-8], y el *middleware* de programación paralela con paso de mensajes *LAM/MPI*. Por este motivo, se ha implementado un conjunto de funciones que implementan las operaciones del protocolo, y por tanto, facilitan enormemente la programación de este tipo de aplicaciones cuando se ejecutan sobre entornos heterogéneos donde los recursos de computación portátiles tienen un comportamiento aleatorio y dinámico. Estas funciones han sido integradas en la biblioteca LAMGAC [Mac01] para permitir al programador controlar tanto el volumen de la distribución de datos entre los procesos como la gestión de la variación dinámica de computadores en la VM_LAM en tiempo de ejecución.

La implementación de las operaciones del protocolo proporcionan al usuario un conjunto de cinco nuevas funciones. Estas funciones se utilizan para calcular la distribución óptima de datos en cada iteración, en función de los parámetros de rendimiento de los computadores y de la red, para conocer si hay computadores portátiles no disponibles debido a su batería o al estar situados en una zona de no cobertura, y también, se utilizan para liberar los recursos utilizados por el mecanismo de equilibrio de carga. Además, cuatro funciones de la implementación original de la biblioteca LAMGAC han sido modificadas para adaptarse a la especificación del protocolo de equilibrio de carga. Las funciones han sido implementadas en el lenguaje de alto nivel C, y hacen uso del protocolo SNMP para obtener información relevante acerca del rendimiento y estado de los recursos de computación, del *middleware* MPI para la comunicación entre los procesos del programa paralelo, del entorno LAM para el envío/recepción de señales de usuario entre distintos procesos y de los mecanismos de comunicación entre procesos como semáforos y memoria compartida.

Tabla 4.1 Funciones de LAMGAC, entidad que la invoca y primitivas del protocolo que implementa

Funciones	*Entidad*	*Primitivas*
void **LAMGAC_Init_an** * (. . .)	*EM*	*Iniciar_EM*
void **LAMGAC_Init_fn** * ()	*EF, EM*	*Iniciar_EF, Enviar_Notificación*
void **LAMGAC_Init_pn** * (. . .)	*EP*	*Iniciar_EP, Enviar_Notificación*
void **LAMGAC_Balance** (. . .)	*EM*	*Equilibrar*
void **LAMGAC_Store_info** (. . .)	*EM*	*Almacenar_Info*
void **LAMGAC_Test_battery_beacon** (. . .)	*EM*	*Comprobar_Batería_y_Enlace*
void **LAMGAC_Itest_battery_beacon** (. . .)	*EM*	*Comprobar_Batería_y_Enlace*
void **LAMGAC_Finalize_slave** (. . .)	*EP, EF*	*Finalizar_Cooperante, Enviar_Notificación*
void **LAMGAC_Finalize** * ()	*EM*	*Finalizar*

En la tabla 4.1 se muestran los nombres de las funciones de la biblioteca que utilizan el protocolo de equilibrio de carga, los tipos de entidades que pueden invocarla y las primitivas del protocolo que implantan. Las funciones que necesitan parámetros se muestran con puntos suspensivos entre paréntesis después de su nombre. En esta tabla, se indica con un asterisco aquellas funciones que se han modificado partiendo de la implementación original de la biblioteca LAMGAC. El resto de funciones son nuevas implementaciones que se han añadido a la biblioteca.

4.1.1 Función LAMGAC_Init_an

Esta función se utiliza exclusivamente en la entidad EM. Se invoca una sóla vez en el proceso con rango cero, P0, o proceso maestro que se ejecuta en dicha entidad, y se debe llamar antes que cualquier otra función de la biblioteca LAMGAC. Sin embargo, la invocación de esta función no tiene que ser la primera sentencia ejecutable del programa.

Para extender las capacidades de la biblioteca LAMGAC con funcionalidades de equilibrio de carga, esta función se ha ampliado a partir de su implementación original. En dicha implementación, esta función se encargaba de crear dos procesos,

manejo_vinculaciones y *manejo_desvinculaciones*, que almacenan, por un lado, la información de los computadores que se suscriben al entorno de computación, y por otro lado, la información de los computadores que solicitan dejar de forma controlada pertenecer a la VM_LAM.

En la nueva implementación, además de realizar las tareas originales, esta función crea un tercer proceso, denominado proceso *Gestor*, cuyo objetivo es recibir y procesar las notificaciones enviadas por el proceso *Colector* ubicado en cada entidad EF y EP, y también, verificar la disponibilidad de las entidades EP que están localizadas en zonas de cobertura reducida. El procesado de las notificaciones lleva consigo el almacenamiento o eliminación de la información acerca de las características y rendimiento actual de los procesos y computadores suscritos o que dejan de formar parte del entorno de computación. La ampliación implantada se corresponde con las tareas realizadas por la primitiva *Iniciar_EM*.

Su declaración es la siguiente:

```
void LAMGAC_Init_an (char *argv[] /*in*/)
```

El argumento de esta función es el segundo parámetro de la función principal *main()* y se utiliza para crear la memoria compartida y los semáforos de sincronización entre el maestro y los procesos hijos: *manejo_vinculaciones*, *manejo_desvinculaciones* y *Gestor*.

4.1.2 Función LAMGAC_Init_fn

Esta función es de uso exclusivo de los procesos que se ejecutan en las entidades EF, y se invoca una sóla vez, antes que cualquier otra función de la biblioteca LAMGAC. En el caso de que el proceso maestro también realice cálculos paralelos, excepcionalmente éste también tiene que invocar esta función después de la llamada a *LAMGAC_Init_an()*. La invocación de esta función se realiza después de invocar a la función *MPI_Init()* que permite que la biblioteca MPI pueda ser utilizada. Sin embargo, esta función no tiene que ser la primera sentencia ejecutable del programa.

Esta función se ha extendido para considerar las especificaciones del protocolo de equilibrio de carga. El objetivo de la nueva implementación es indicar al proceso *Colector* que se ejecuta en el sistema local que un nuevo proceso esclavo MPI ha comenzado su ejecución en el computador, y por tanto, desea ser considerado por el protocolo de equilibrio de carga en las siguientes distribuciones de datos. Para ello, debe enviar una notificación al proceso *Gestor* con la información acerca de las características y rendimiento actual de dicho computador. Esta extensión de la función se corresponde con las primitivas *Iniciar_EF* y *Enviar_Notificación.*

Su declaración es la siguiente:

void ***LAMGAC_Init_fn()***

4.1.3 Función LAMGAC_Init_pn

Esta función es de uso exclusivo de los procesos que son expandidos por el proceso maestro en las entidades EP, y se invoca una sóla vez, antes que cualquier otra función de la biblioteca LAMGAC. La invocación de esta función se realiza después de llamar a las funciones *MPI_Init()* y *MPI_Comm_get_parent()*, ambas pertenecientes a la biblioteca MPI, que permiten que la biblioteca MPI pueda ser utilizada y que se obtenga un intercomunicador con el proceso maestro, respectivamente.

En su implantación original, esta función obtiene el identificador del proceso MPI expandido y el número de procesos de la aplicación paralela que se ejecutan en los computadores fijos y los computadores portátiles.

En la nueva implantación, además de realizar las tareas originales, esta función le indica al proceso *Colector* que se ejecuta en el sistema local, que un nuevo proceso esclavo MPI ha comenzado su ejecución en el computador, y por tanto, desea ser considerado por el protocolo de equilibrio de carga en las siguientes distribuciones de datos. Para ello, debe enviar una notificación al proceso *Gestor* con la información acerca de las características y rendimiento actual de dicho computador. Esta extensión de

la función se corresponde con las primitivas *Iniciar_EP* y *Enviar_Notificación*.

Su declaración es la siguiente:

```
void LAMGAC_Init_pn (
        int awareness /*in*/,
        int *my_id /*out*/,
        int *NA /*out*/,
        int *NI /*out*/,
        MPI_Comm COMM_PARENT /*in*/)
```

El primer parámetro puede tomar solamente dos valores (0 ó 1), y en función de éste, los datos devueltos en los parámetros *my_id*, *NA* y *NI* serán diferentes:

- 0, el proceso que la invoca no realiza invocaciones a la función *LAMGAC_Awareness_update()* [Mac01]. La función devuelve el identificador del proceso MPI en el parámetro *my_id*, y en los parámetros *NA* y *NI* un -1.
- 1, el proceso que la invoca realiza invocaciones a la función *LAMGAC_Awareness_update()*. La función devuelve el identificador del proceso MPI en el parámetro *my_id*, y el número de procesos que se ejecutan en las entidades EF y EP en los parámetros *NA* y *NI*, respectivamente.

En ambos casos, el parámetro COMM_PARENT contiene el intercomunicador devuelto por la función *MPI_Comm_get_parent()* llamada antes de realizar la invocación a esta función.

4.1.4 Función LAMGAC_Balance

Esta función sólo puede ser invocada por el proceso maestro, el cual se ejecuta en la entidad EM. El objetivo de esta función es estimar la cantidad de datos que tiene que ser distribuida a cada proceso esclavo vinculado al protocolo de equilibrio de carga para que dichos procesos finalicen al mismo tiempo en cada iteración, es decir, calcula una

distribución de datos tal que exista un equilibrio de carga entre todos los procesos paralelos que colaboran en la solución de un problema. Por tanto, esta función implementa la primitiva *Equilibrar*.

Su declaración es la siguiente:

```
float LAMGAC_Balance (
                int first /*in*/,
                int signalarm /*in*/,
                float percent /*in*/,
                int nproc /*in*/,
                float micalc [] /*in*/,
                int sizedata /*in*/,
                int sizeresults /*in*/,
                int distdata /*in*/,
                int order [] /*out*/,
                int ndata [] /*out*/)
```

El primer parámetro de esta función, *first*, puede tomar dos valores:

- 0, indica que la iteración actual no es la primera iteración de la aplicación paralela. En esta situación, la estimación del volumen de datos a distribuir se realiza en base al rendimiento de cada proceso en la iteración previa, aplicando la métrica 3.8. Excepcionalmente, en el caso de la existencia de un evento significativo y reciente en algún computador esclavo, o la vinculación al entorno de una nueva entidad EP, el volumen de datos a comunicar al proceso que se ejecuta en dicho computador se estima en base a parámetros relacionados con las características y rendimiento actual del computador y la red, siguiendo la métrica 3.6.
- 1, indica que la estimación del volumen de datos a enviar a cada proceso esclavo se corresponde con la primera distribución de datos que se realiza en el programa paralelo, es decir, la iteración actual es la primera iteración de la aplicación. En esta situación, no se dispone de ninguna información acerca del

rendimiento de los procesos en los computadores, por tanto, la estimación de datos se calcula en base a parámetros relacionados con las características y rendimiento actual de cada computador que compone el entorno de computación (métrica 3.6).

El segundo parámetro de esta función, *signalarm*, también puede tomar dos posibles valores:

- 0, indica que la señal SIGALRM no se programa en la ejecución de la función.
- 1, indica que la señal SIGALRM se programa para ser enviada un cierto tiempo después de finalizar la ejecución de la función. El tiempo de disparo de la señal coincide con el tiempo de ejecución empleado por el proceso más rápido en la iteración previa. De esta forma, en una situación de equilibrio y siempre que el número de operaciones de cálculo en cada iteración sean similares, los resultados calculados por los procesos esclavos deberían llegar de inmediato después de enviar la señal. En el caso de utilizar este valor en el parámetro *signalarm*, la función *LAMGAC_Test_battery_beacon()* debe ser utilizada para provocar el bloqueo del proceso maestro hasta que reciba la señal SIGALRM. Cuando se recibe una señal de este tipo, si el proceso está bloqueado, se desbloquea y se comprueba si existe alguna entidad EP que pueda desaparecer de la máquina paralela virtual en breve debido a un nivel bajo de energía en la batería, o que la entidad EP está situada fuera del área de cobertura del punto de acceso asociado, y por tanto, no hay enlace de comunicación posible.

El significado del resto de parámetros es el siguiente:

- *percent*. Este parámetro fija el grado de desequilibrio que se permite. Su valor está comprendido entre 0 y 99.9. Un valor cercano a cero indica un equilibrio muy ajustado, es decir, los tiempos de ejecución de los procesos esclavos deben ser similares al tiempo de ejecución medio de los mismos. Si este parámetro se fija al valor máximo, se acepta como estado de equilibrio una variación del

99.9% del tiempo de ejecución de cada proceso con respecto al tiempo de ejecución medio. La elección del valor de este parámetro depende del orden de magnitud del tiempo de ejecución de los procesos y del grado de equilibrio que se desee obtener. Si el tiempo de ejecución es elevado, el valor del parámetro *percent* debe ser pequeño para evitar fuertes variaciones del mismo entre los procesos. Por otro lado, si el tiempo de ejecución de los procesos es pequeño, el valor de este parámetro no debe ser muy bajo para evitar que nunca se alcance el equilibrio. Nótese que cuando en una iteración se alcanza el equilibrio, es decir los tiempos de ejecución de todos los procesos están dentro del porcentaje fijado respecto al tiempo de ejecución medio, en la siguiente iteración se vuelve a distribuir el mismo porcentaje de datos a cada proceso, según la especificación definida en la primitiva *Equilibrar*. Esto último introduce la ventaja de que se realizan menos cálculos en la primitiva, y por consiguiente, la ejecución de la primitiva *Equilibrar* es más rápida cuando los tiempos de ejecución en la iteración previa están equilibrados.

- *nproc*. Representa el número actual de procesos que colaboran en los cálculos paralelos vinculados al protocolo de equilibrio de carga, tanto en las entidades EF como en las entidades EP. La información correspondiente a todos los procesos implicados en la solución del problema tiene que ser accessible por el proceso maestro a través de la memoria compartida que posee con el proceso *Gestor*. En caso contrario, la función permanece bloqueada hasta que el número de procesos de los que se tiene información sea igual a *nproc*.

- *micalc*. Vector de números reales que representa el número de millones de instrucciones máquinas ejecutadas en cada procesador para procesar una unidad de dato y obtener un único resultado. Este valor debe ser calculado con la relación existente entre el tiempo de cálculo obtenido para procesar una única unidad de dato y los millones de intrucciones por segundo del procesador (MIPS). Este vector tiene que ser suministrado por el programador a la aplicación paralela.

- *sizedata*. Representa el tamaño en bytes de una única unidad de dato. Este dato se utiliza solamente en la métrica 3.6, es decir, en las siguientes situaciones: en la primera iteración del programa paralelo, cuando se recibe una notificación del tipo *lbnLoad* o *lbnUpLink* o cuando se expande un nuevo proceso en un computador portátil y se recibe la notificación *lbnAppStart*.

- *sizeresults*. Representa el tamaño en bytes de un único resultado. Esta dato se utiliza solamente en la métrica 3.6, es decir, en las siguientes situaciones: en la primera iteración del programa paralelo, cuando se recibe una notificación del tipo *lbnLoad* o *lbnUpLink* o cuando se expande un nuevo proceso en un computador portátil y se recibe la notificación *lbnAppStart*.

- *distdata*. Representa el número total de unidades de datos a distribuir en la iteración actual.

- *order*. Vector de enteros de dimensión igual al número de procesos, donde se almacena la secuencia de envío a seguir cuando se realiza la distribución de datos. Este orden se establece de mayor a menor según la relación entre el tiempo de ejecución empleado y el número de unidades de datos procesadas en la iteración previa por cada proceso (métrica 3.7). El valor almacenado en cada elemento se corresponde con el rango de un proceso, de esta forma, el primer elemento del vector contiene el identificador del primer proceso al que se le envían datos, el segundo elemento contiene el identificador del segundo proceso al que se le envían datos, y así sucesivamente.

- *ndata*. Vector de enteros de dimensión igual al número de procesos, donde se almacena el número de unidades de datos a distribuir a cada proceso esclavo. La cantidad correspondiente a cada proceso está ubicada en el elemento del vector con número igual al rango de dicho proceso dentro de la aplicación paralela-distribuida, es decir, en la posición cero del vector está el número de unidades de datos a comunicar al proceso con rango 0, en la posición uno los datos correspondientes al proceso con rango 1, y así sucesivamente. Si en una iteración no se envían datos a un proceso, la entrada del vector *ndata*

correspondiente al identificador del proceso se inicia a cero.

Además, la función devuelve el tiempo de ejecución medio de los procesos en la iteración previa.

El proceso maestro debe invocar esta función en cada iteración antes de realizar la distribución de datos para tener conocimiento del volumen de datos a enviar a cada proceso esclavo, y en qué orden se debe realizar el envío.

4.1.5 Función LAMGAC_Store_info

Esta función sólo puede ser invocada por el proceso maestro de la aplicación paralela-distribuida. El objetivo de esta función es almacenar el tiempo de ejecución empleado por el proceso esclavo que envió los resultados y el número efectivo de unidades de datos sobre los cuales se realizaron cálculos. Debido a que hay aplicaciones donde la cantidad de datos distribuidos a cada proceso no está siempre correlado con el cómputo realizado, es decir, el número de unidades de datos enviadas es diferente al número de unidades de datos procesadas, es necesario tener en cuenta dicha información para realizar una correcta distribución de datos en la siguiente iteración. Por tanto, esta función implementa la primitiva *Almacenar_Info*.

Su declaración es la siguiente:

```
void LAMGAC_Store_info (
                int rank /*in*/,
                int ndata /*in*/,
                double initime /*in*/)
```

El significado de los parámetros es el siguiente:

- *rank*. Representa el rango o identificador del proceso esclavo que envió los resultados.
- *ndata*. Representa el número efectivo de unidades de datos sobre los cuales fueron realizados los cálculos.

- *initime*. Representa el instante de tiempo de comienzo de la distribución de datos en la iteración actual.

El proceso maestro debe invocar esta función en cada iteración después de recibir los resultados de cada proceso, es decir, se invoca tantas veces como resultados se han recibido.

4.1.6 Función LAMGAC_Test_battery_beacon

La función *LAMGAC_Test_battery_beacon()* se invoca únicamente en el proceso maestro. Su ejecución provoca que dicho proceso permanezca bloqueado hasta que una señal del tipo SIGALRM se reciba, la cual se debió programar durante la llamada a la función *LAMGAC_Balance()* si el parámetro *signalarm* de ésta se fijó a 1. Es obvio que esta función se tiene que invocar después de la llamada a la función *LAMGAC_Balance()*. Cuando se recibe la señal, se comprueba si existe alguna entidad EP con un nivel bajo de energía de la batería o que está situado en una zona donde no hay enlace de comunicación inalámbrica con la entidad EM. Si la señal SIGALRM se recibe antes de invocar la función o no se programa el envío de esta señal en la función *LAMGAC_Balance()*, se comprueba el estado de las entidades EP sin bloquear el proceso maestro. Esta función implementa la primitiva *Comprobar_Batería_y_Enlace*.

Su declaración es la siguiente:

```
void LAMGAC_Test_battery_beacon (
        int slaveprocs /*out*/,
        int procrank [] /*out*/)
```

El significado de los parámetros es el siguiente:

- *slaveprocs*. Indica el número de procesos esclavos localizados en las entidades EP cuyo nivel de energía en la batería no es suficiente para acabar la iteración actual y/o no tienen comunicación inalámbrica con el proceso maestro en el instante de la llamada a la función.

- *procrank*. Vector de enteros de dimensión *slaveprocs* que almacena el rango de los procesos con las características mencionadas.

Esta función se debe utilizar cuando el parámetro *signalarm* de la función *LAMGAC_Balance()* es igual a 1. El proceso maestro debe invocar esta función en cada iteración antes de iniciar la recepción de resultados. De esta forma, se tiene conocimiento de aquellas entidades que pueden tener problemas para enviar los resultados, y por tanto, no se debe implementar la recepción para ellos.

4.1.7 Función LAMGAC_Itest_battery_beacon

La función *LAMGAC_Itest_battery_beacon()* se invoca únicamente en el proceso maestro, y ésta implanta la primitiva *Comprobar_Batería_y_Enlace*. Por tanto, tiene el mismo objetivo descrito para la función anterior, *LAMGAC_Test_battery_beacon()*. La única diferencia existente con esta última es que no provoca el bloqueo del proceso maestro hasta recibir una señal del tipo SIGALRM, es decir, cuando se invoca esta función directamente se comprueba si hay alguna entidad EP con problemas para acabar la iteración actual debido a la poca energía remanente en la batería y/o que no exista enlace de comunicación inalámbrico entre la entidad EP y la entidad EM. De esta forma, esta función se puede invocar en cualquier parte del programa paralelo.

Su declaración es la siguiente:

```
void LAMGAC_Itest_battery_beacon (
                int slaveprocs /*out*/,
                int procrank [] /*out*/)
```

Los parámetros son los mismos que los utilizados en la función anterior, y por tanto, tienen el mismo significado.

Idealmente, esta función se podría invocar antes de comenzar la distribución de datos para evitar el envío de información a aquellas entidades que no están disponibles, y puede ser utilizada en un mecanismo de recepción de datos controlada para verificar si

de alguna entidad EP no se pueden recibir los resultados. La utilización de esta función no limita la invocación de la versión bloqueante, ambas pueden ser utilizadas en el mismo programa paralelo.

4.1.8 Función LAMGAC_Finalize_slave

La función *LAMGAC_Finalize_slave()* se invoca en cada proceso esclavo para indicar al proceso *Colector* que se ejecuta en el sistema local que no va a realizar más cálculos paralelos vinculado al protocolo de equilibrio de carga. Por tanto, el proceso *Colector* debe notificar al proceso *Gestor* que la información referente al rendimiento de dicho proceso MPI no es necesaria, y debe ser eliminada de la memoria compartida. Por tanto, esta función implementa la primitiva *Finalizar_Cooperante* y *Enviar_Notificación*.

Su declaración es la siguiente:

```
void LAMGAC_Finalize_slave (int rank /*in*/)
```

El parámetro *rank* representa el rango o identificador del proceso que invoca la función. La invocación de esta función no tiene que ser necesariamente la última sentencia ejecutable del programa. Se debe invocar en algún punto del programa a partir del cual el proceso no va a recibir distribuciones de carga de forma equilibrada con el resto de procesos.

4.1.9 Función LAMGAC_Finalize

La función *LAMGAC_Finalize()* se invoca únicamente en el proceso maestro. Para extender las capacidades de la biblioteca, esta función ha sido ampliada a partir de su versión original. En la implantación inicial, esta función finalizaba las acciones de control de vinculaciones y desvinculaciones de los procesos expandidos en las entidades EP. En la implantación actual, además de realizar las tareas originales, esta función se encarga de finalizar las acciones de control sobre las notificaciones enviadas por cada proceso *Colector*. Después de la invocación a esta función, la ejecución del programa

paralelo continúa en los procesos que colaboraban en la ejecución del programa antes de invocar a la función *LAMGAC_Finalize()*, no produciéndose ninguna vinculación o desvinculación posterior de procesos en las entidades EP, y ninguna recepción de las notificaciones enviadas por cualquier proceso *Colector* del entorno de computación, así como tampoco se crea ninguna hebra de ejecución para controlar las entidades EP que estén localizadas en zonas de cobertura limitada.

Su declaración es la siguiente:

```
void LAMGAC_Finalize()
```

La invocación a esta función no tiene que ser necesariamente la última sentencia ejecutable del programa. Debe ser invocada en algún punto del programa a partir del cual no se invoquen funciones de la biblioteca LAMGAC. Esta función se corresponde con la primitiva *Finalizar*.

4.2 Recepción controlada de datos con LAMGAC extendida

En un entorno de computación LAN-WLAN donde los computadores portátiles pueden entrar y salir de cobertura en tiempo de ejecución y la batería de éstos es un recurso finito de energía, es necesario establecer un mecanismo para controlar la recepción de resultados enviados desde estos computadores. Si no se establece ningún mecanismo sobre estos parámetros, puede darse la situación de que el proceso maestro quede bloqueado, temporal o indefinidamente, esperando por resultados provinientes de computadores portátiles que nunca van a llegar por estar éstos fuera de cobertura o su batería se haya agotado en el transcurso de la iteración. Incluso, aunque se supiese a priori que el computador va a regresar a la zona de cobertura después de un tiempo finito, la espera por la recepción de los resultados calculados en éste provocaría que el sistema quede desequilibrado, con el consecuente aumento del tiempo de ejecución de la aplicación paralela-distribuida.

Combinando las funciones de control de la batería y cobertura,

LAMGAC_Test_battery_beacon o *LAMGAC_Itest_battery_beacon()*, y la función *LAMGAC_Store_info()* con algunas funciones de la biblioteca MPI-2, se puede diseñar un mecanismo de recepción de datos que controle el estado de los computadores portátiles. El mecanismo desarrollado se puede dividir en seis pasos, los cuales se detallan a continuación:

1. *Comprobar estado de los computadores portátiles.* Antes de iniciar la recepción, se invoca una de las funciones de control de batería y cobertura de *LAMGAC* (*LAMGAC_Test_battery_beacon()* o *LAMGAC_Itest_battery_beacon()*). De esta forma se conoce antes de iniciar la recepción de datos si existen computadores portátiles en disposición de enviar resultados.

2. *Inicio de la recepción de datos.* Se inicia una recepción no bloqueante para cada uno de los procesos a los que se les envió datos y que están disponibles en el instante actual. La recepción de datos puede ser iniciada con la función *MPI_Irecv()*.

3. *Comprobación de recepciones completadas.* Invocando la primitiva *MPI_Testsome()* verificamos si alguna de las operaciones de recepción iniciadas en el paso anterior ha sido completada.

4. *Almacenar información.* Por cada una de aquellas operaciones de recepción que han sido completadas, se invoca la función *LAMGAC_Store_info()* para almacenar el tiempo de ejecución y el número de unidades de datos procesadas por el proceso esclavo que envió los resultados. Además, en este paso el programador debe implementar las operaciones que debe realizar con los datos recibidos.

5. *Comprobar variación de estado de los computadores portátiles.* En este paso se invoca la función no bloqueante *LAMGAC_Itest_battery_beacon()* para conocer si durante el tiempo transcurrido del paso 2 al paso 4, uno o varios computadores se han situado fuera de cobertura o su batería se ha agotado.

6. *Cancelación de recepciones.* Si la comprobación del estado de los computadores portatiles generó como resultado el rango de uno o varios procesos no disponibles, entonces se procede a la cancelación de la recepción no bloqueante iniciada para dichos procesos en el paso 2. La cancelación de cada recepción se realiza con la función *MPI_Cancel()*.

La secuencia de pasos del 3 al 6 se repite hasta que no queden más operaciones de recepción por completar. En la figura 4.1 se muestra un diagrama de flujo del mecanismo de recepción controlada con *LAMGAC*.

Por otro lado, el programador debe ser consciente que los resultados que debían ser obtenidos con las recepciones no iniciadas y/o canceladas deben ser calculadas por otro proceso. El programador puede optar por dos soluciones: a) los resultados son calculados por el proceso maestro, o b) se realiza una iteración extra del programa paralelo con los datos que dan lugar a los resultados no obtenidos, es decir, se realiza una distribución de datos entre los procesos disponibles y se vuelve a implementar la recepción controlada. El programador debe estudiar cuál es la solución menos costosa en función del volumen de datos a comunicar y los cálculos a realizar.

4.3 Esqueletos de programación con LAMGAC extendida

En este apartado se presenta el modelo de computación que utiliza la biblioteca *LAMGAC* extendida, desarrollada en este trabajo de investigación. Se ha elegido el esquema MPMD porque las acciones realizadas en el programa difieren en función del tipo de proceso y entidad de que se trate. En general, el proceso maestro se encarga de estimar y realizar la distribución dinámica de datos entre todos los procesos esclavos que ejecutan la aplicación paralela, de almacenar los resultados calculados por éstos y de controlar el estado de los computadores portátiles. Opcionalmente, este proceso puede también colaborar en los cálculos. Entre las acciones básicas de los procesos esclavos se incluyen la comunicación de datos (la recepción de nuevos datos para calcular y el envío de resultados) y la realización de cálculos con los datos recibidos. El código que ejecuta

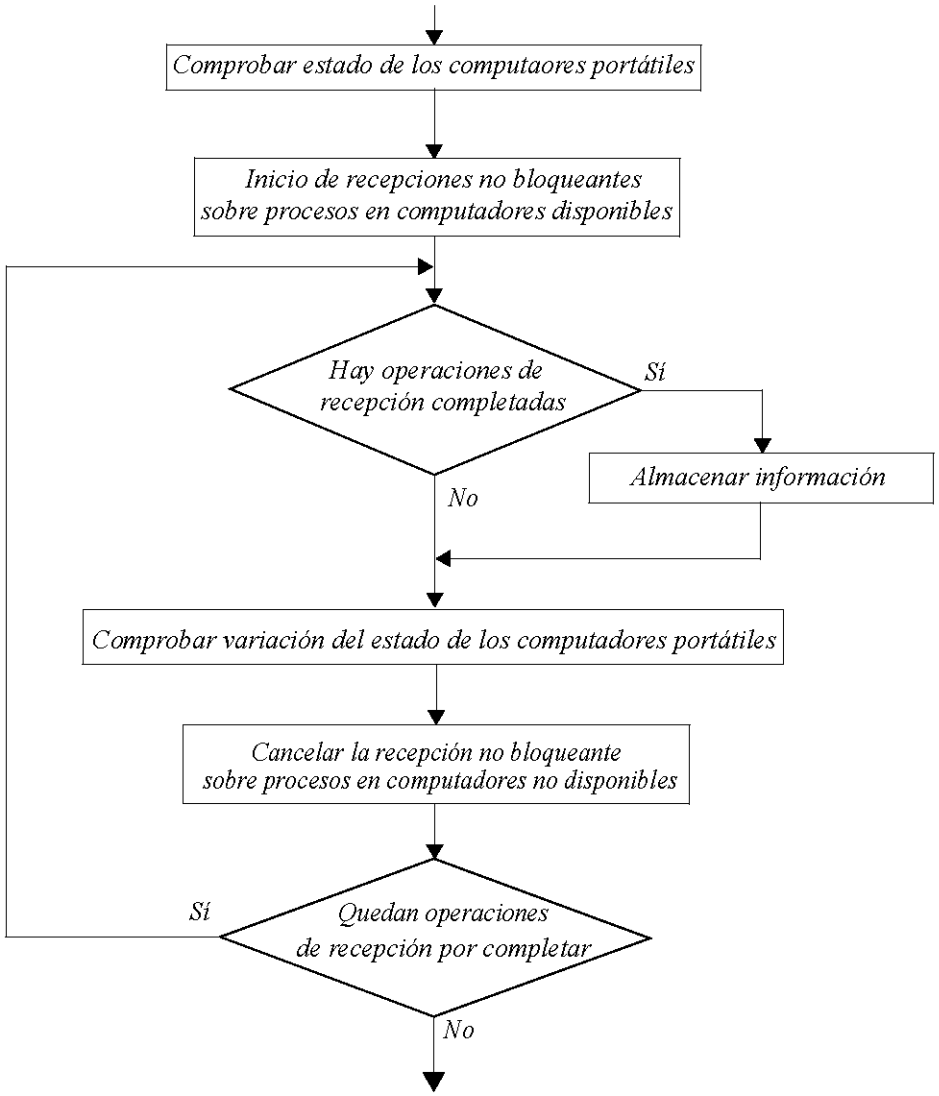


Figura 4.1 Recepción controlada de datos con LAMGAC

un programa en un computador fijo y otro que se ejecuta en un computador portátil difiere, básicamente, en el tipo de comunicaciones empleadas (comunicaciones colectivas frente a comunicaciones punto a punto), debido a las características de los intercomunicadores de MPI cuando los procesos son expandidos en tiempo de ejecución.

4.3.1 Esquema de programación

Este esquema de computación se utiliza en aquellas aplicaciones donde la distribución de datos no se realiza de forma distribuida entre todos los procesos, sino que existe un

proceso, denominado maestro, que se encarga de realizar una distribución de datos dinámica en cada iteración del programa paralelo. En nuestro modelo, el proceso maestro, conociendo el estado del resto de procesos es capaz de calcular y realizar una distribución de datos equilibrada.

Este esquema se compone de tres esqueletos (figura 4.2), que incluyen la directiva *#include "lamgac.h"*. Este archivo almacena las definiciones y declaraciones necesarias para compilar un programa MPI que utiliza la biblioteca LAMGAC. También, es necesario incluir en cada esqueleto el archivo de cabecera *mpi.h* de la biblioteca MPI, para poder realizar invocaciones a las funciones definidas en dicha biblioteca.

El esqueleto para el proceso maestro sigue la siguiente estructura. Inicialmente, se invoca la rutina *INITIALIZE* que realiza la llamada a las funciones *MPI_Init()*, *MPI_Comm_rank()* y *MPI_Comm_size()*. Antes de invocar a cualquier función de la biblioteca LAMGAC, el proceso maestro llama a la función *LAMGAC_Init_an()*. En el caso de que el proceso maestro fuese a realizar cálculos paralelos, éste también tiene que invocar a la función *LAMGAC_Init_fn()* después de llamar a la función *LAMGAC_Init_an()*.

Periódicamente, el proceso maestro invoca la función *LAMGAC_Update()* para detectar y actualizar los cambios que se produzcan en la VM_LAM. La frecuencia con la que esta comprobación se realiza depende del programador, pero normalmente se debe hacer una vez en cada iteración del programa paralelo. También, en lugar de la función *LAMGAC_Update()* se podría llamar a la función *LAMGAC_Awareness_update()*. Sin embargo, la sincronización existente con los procesos esclavos en cada llamada y la decisión de tener un balanceador de carga centralizado hace que la utilización de esta función sea sólo rentable cuando sea necesario conocer por todos los procesos el número de procesos que se ejecutan en los computadores.

Una vez conocidos los cambios que se han producido y antes de realizar la distribución de datos, se invoca la función *LAMGAC_Balance()* para estimar la cantidad de unidades de datos que se distribuye a cada proceso. Además, en la llamada a esta

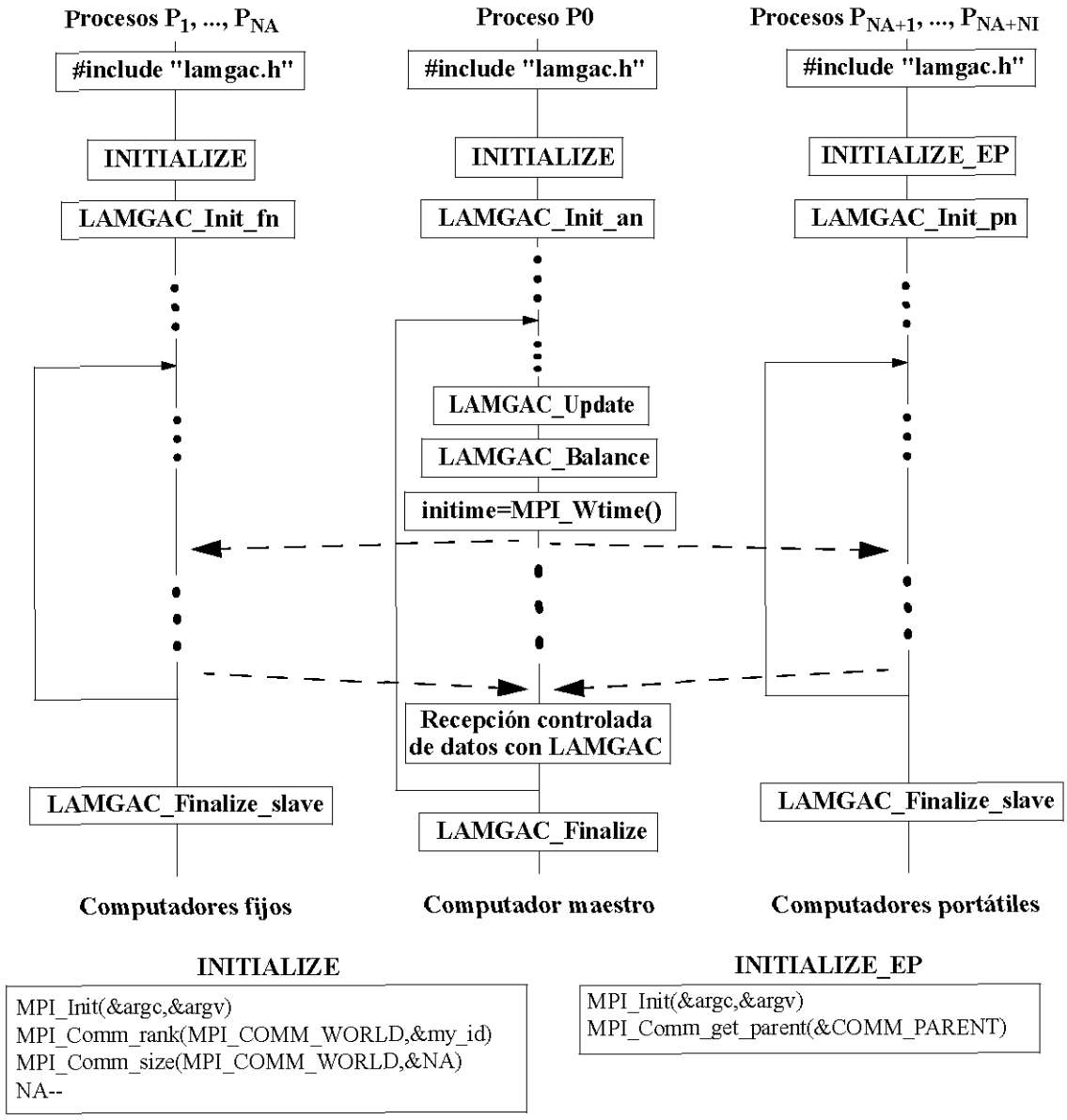


Figura 4.2 Esquema de programación

función, se puede programar el envío de la señal SIGALRM para comprobar con la función *LAMGAC_Test_battery_beacon()* el estado de los computadores portátiles justo antes de iniciar la recepción de resultados. La programación de esta señal sólo se debe realizar en aplicaciones en las que los cálculos de todas la iteraciones sean similares. Antes de realizar la distribución de datos, se obtiene una marca de tiempo, denominada *initime*, la cual nos servirá para determinar el tiempo de ejecución de cada proceso

cuando se reciban los resultados.

Una vez llevada a cabo la distribución de datos, se implementa el mecanismo de recepción controlada de datos con *LAMGAC* explicado en la sección 4.2. En el caso de existir uno o varios procesos ejecutándose en computadores portátiles para los que no se implementó la recepción o para los que las recepciones iniciadas fueron canceladas, la carga inicialmente asignada a estos procesos debe ser reasignada al resto de procesos, o llevar a cabo los cálculos correspondientes de forma local.

Opcionalmente, y en cualquier instante de la iteración puede invocarse la función *LAMGAC_Itest_battery_beacon()* para tener conocimiento de aquellos posibles computadores que pueden abandonar la VM_LAM de forma abrupta. Por último, y una vez acabadas las iteraciones del programa paralelo, se invoca la función *LAMGAC_Finalize()*, no pudiendo después realizar ninguna llamada a otras funciones de la biblioteca *LAMGAC*.

El esqueleto para los procesos con identificador desde 1 hasta NA (procesos en computadores fijos) es igual al de los procesos con identificador desde NA+1 hasta NA+NI (procesos en computadores portátiles), con la excepción de la invocación de diferentes funciones MPI en la inicialización, la llamada a la función *LAMGAC_Init_fn()* en los computadores fijos y la función *LAMGAC_Init_pn()* en los computadores portátiles. Antes de hacer cualquier invocación a las funciones de la biblioteca LAMGAC, se llama a la función *LAMGAC_Init_fn()* o *LAMGAC_Init_pn()*, dependiendo del recurso donde se ejecute el proceso. Al igual que en el proceso maestro, al inicio del programa, los procesos con identificador desde 1 hasta NA invocan la rutina *INITIALIZE* obteniendo así el identificador del proceso (*my_id*) y el número de procesos ejecutándose en los computadores fijos. Por otro lado, los procesos expandidos en los computadores portátiles comienzan su ejecución como cualquier programa MPI, invocando la función *MPI_Init()*, y éstos acceden a su intercomunicador llamando a la función *MPI_Comm_get_parent()*. Esta funciones son invocadas al llamar a la rutina *INITIALIZE_EP*. Después de invocar la función *LAMGAC_Init_pn()*, se obtiene el identificador del proceso (*my_id*) y comienza la ejecución del programa propiamente

dicho. Tanto los procesos que se ejecutan en los computadores portátiles como los que se ejecutan en los computadores fijos finalizan con la función *LAMGAC_Finalize_slave()*. Después de la ejecución de esta última función, la información del proceso se elimina de la memoria compartida existente entre el proceso maestro y el *Gestor* en el computador maestro, y el proceso esclavo deja de formar parte del protocolo de equilibrio de carga.

4.4 Programando con LAMGAC extendida

A título de ejemplo, en este apartado se presenta una aplicación que utiliza el esquema de programación presentado en el apartado anterior. Esta aplicación resuelve el producto matriz por vector utilizando las nuevas funciones añadidas a la biblioteca *LAMGAC* para equilibrar los tiempos de ejecución de los procesos que colaboran en la solución del problema.

El número de procesos que inician la aplicación es igual a $NA+1$. El primer término de la expresión corresponde al número de procesos que se ejecutan en los computadores fijos y el segundo término tiene en cuenta al proceso maestro. Este número de procesos se mantiene constante durante la ejecución del programa, puesto que se supone que los procesos en los computadores fijos nunca solicitan desvincularse. El número de procesos inicial puede aumentar durante la ejecución del programa debido a la expansión de nuevos procesos en los computadores portátiles que se suscriben a la VM_LAM. Esta expansión se realiza cuando el proceso maestro invoca la función *LAMGAC_Update()*, y existen peticiones de vinculación pendientes. El número de procesos que colaboran en la resolución del problema también puede decrecer en tiempo de ejecución. Esto sucede siempre que el proceso maestro invoque a *LAMGAC_Update()* y existan peticiones de desvinculación pendientes desde algún computador portátil.

4.4.1 Multiplicación matriz por vector

Dada una matriz A de N filas y M columnas y un vector b de M filas, el problema del álgebra lineal del cálculo de la multiplicación matriz por vector se resuelve de la

siguiente forma:

$$A \times b = c$$

Donde:

- c es el vector resultado de la multiplicación. Su dimensión es igual al número de filas de la matriz A.

Los elementos de las matrices son números reales, esto es:

$$\begin{array}{l} A[i][j] \in \Re \\ b[j] \in \Re \qquad \forall i, j \in \aleph \qquad 0 \le i < N \qquad 0 \le j < M \\ c[i] \in \Re \end{array}$$

Donde:

- *A[i][j]* es un elemento de la matriz A,
- *b[j]* es un elemento del vector B, y
- *c[i]* es un elemento del vector resultado.

El elemento *c[i]* se obtiene sumando los productos parciales de los elementos de la fila i de la matriz A por los elementos del vector b.

En la figura 4.3 se muestra el código secuencial de la multiplicación matriz por vector. Con respecto al código paralelo, en cada iteración se distribuye un conjunto de filas entre todos los procesos paralelos. Cuando el proceso maestro recibe los resultados calculados por todos los procesos, vuelve a distribuir otro conjunto de filas. Este proceso se repite hasta que no queden más filas por distribuir. A continuación describimos el funcionamiento de los tres esqueletos exponiendo las acciones principales que efectúan.

Esqueleto para el proceso maestro

El programa del proceso maestro realiza, básicamente, las siguientes acciones:

- *Acción 1*: *distribución inicial de datos.* Inicialmente, el proceso maestro replica el vector b en todos los procesos con los que se inicia la aplicación.
- *Acción 2*: *actualización del número de procesos.* Periódicamente, se invoca la

```
for (i=0; i<N; i++) {
        C[i]= 0;
        for (j=0; j<M; j++)
            C[i] = C[i] + A[i][j]*B[j];
}
```

Figura 4.3 Código secuencial para el producto matriz por vector

función *LAMGAC_Update()* para actualizar el número de procesos en los computadores portátiles. Cuando un nuevo proceso se expande en un computador, se le envía el vector *b* para que pueda realizar los cálculos. Si existe algún proceso que se quiere desvincular, se le envía un mensaje para que acabe su ejecución. Por simplicidad, supongamos que esta comprobación se realiza al inicio de cada iteración del programa paralelo.

- *Acción 3*: *estimación del volumen de datos a distribuir.* Después de conocer los cambios producidos en la VM_LAM, se invoca la función *LAMGAC_Balance()* para obtener el volumen de datos a distribuir a cada proceso y la secuencia de envío a éstos. Para la aplicación en cuestión, la distribución de datos se corresponde con el reparto de un número determinado de filas de la matriz *A* fijado por el programador. Un aspecto importante a tener en cuenta en la llamada a esta función es la posibilidad de conocer el estado de los computadores portátiles antes de recibir los resultados de los procesos esclavos. Asignando un valor igual a 1 al parámetro *signalarm*, se programa el envío de la señal SIGALRM. Se aconseja utilizar esta opción siempre y cuando el cálculo realizado en cada iteración sea similar. En la multiplicación de matriz por vector, los cálculos a realizar son iguales en todas las iteraciones, lo cual no significa que los procesos tardan lo mismo en calcular, de ahí la importancia de realizar el balanceo de la carga.

- *Acción 4*: *distribución de filas.* Después de conocer el volumen de datos a enviar a cada proceso, éste se distribuye según la secuencia de envío obtenida

en la acción anterior. La distribución de filas se realiza con rutinas de envío de datos no bloqueantes. Si en una iteración dada no existen filas para distribuir a uno o varios procesos, a éstos se les envía un mensaje para que finalicen su ejecución. También, si uno o varios computadores portátiles solicitan desvincularse, a los procesos esclavos en dichos computadores se les envía un mensaje para que finalicen el programa.

En la figura 4.4 se presenta un ejemplo de distribución de filas: en la iteración i-ésima, el número de filas que quedan por distribuir es 9 y el número de procesos en los computadores fijos y portátiles es NA=2 (P1 y P2) y NI=1 (P3), respectivamente. En esta iteración, la llamada a la función *LAMGAC_Balance()*, con el parámetro *distdata* igual a 6, devuelve como resultado que a los procesos P1, P2 y P3 se les debe enviar tres, una y dos filas de la matriz *A*, respectivamente (vector *ndata*). La secuencia de envío de datos se realiza en el orden siguiente P1, P3, P2 (vector *order*). En la siguiente iteración, el número de filas que queda por distribuir es 3, y la invocación a *LAMGAC_Balance()* devuleve un número de dos filas a enviar al proceso P1 y una fila al proceso P3, por lo que el proceso P2 no recibirá datos para calcular. En este caso, al proceso P2 se le envía un mensaje para que finalice su ejecución.

- *Acción 5*: *recepción controlada de resultados.* En esta acción se implementa el mecanismo de recepción controlada explicado en la sección 2 de este capítulo. Antes de recibir los resultados calculados por los procesos, se debe comprobar el estado de los computadores portátiles para evitar esperas por datos que nunca van a llegar. Para conocer si existen recursos en esta situación, se puede invocar a la función *LAMGAC_Test_battery_beacon()* o a la función *LAMGAC_Itest_battery_beacon()*. Nótese que la utilización de la primera función implica el bloqueo del proceso maestro hasta que transcurra un espacio de tiempo contado a partir de la finalización de la función *LAMGAC_Balance()*, siempre y cuando se programe el envío de la señal

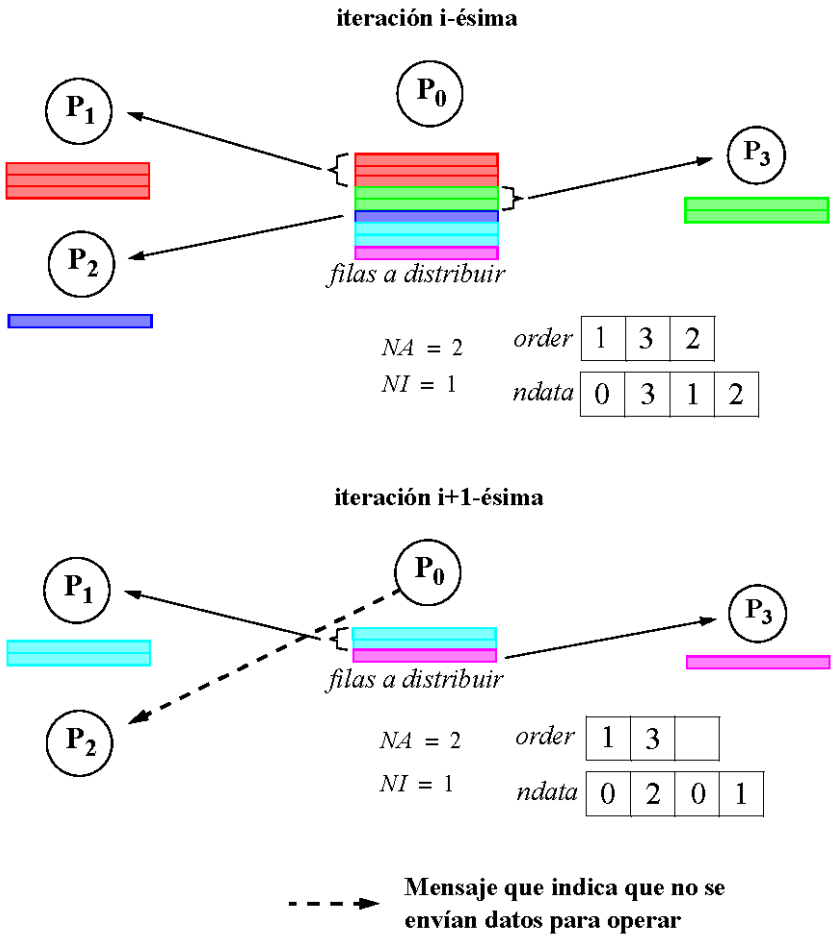


Figura 4.4 Ejemplo de distribución dinámica de filas

SIGALRM. Dicho tiempo viene determinado por el tiempo de ejecución del proceso más rápido en la iteración previa. Una vez comprobado el estado de los computadores portátiles, se implementa una recepción de datos no bloqueante para todos los procesos, excepto para aquellos que se ejecutan en los computadores portátiles con problemas. Una vez iniciadas las recepciones, el proceso maestro entra en un bucle esperando por los resultados de los cálculos paralelos. En primer lugar, se comprueba si alguna operación de recepción ha finalizado. Si es así, se invoca la función *LAMGAC_Store_info()* para calcular y almacenar el tiempo de ejecución del proceso que ha enviado los resultados y la carga efectiva computada. En el caso de la aplicación en cuestión, la carga

efectiva computada coincide con el número de filas distribuidas al proceso. A continuación, se vuelve a comprobar el estado de los computadores portátiles mediante la invocación de la función *LAMGAC_Itest_battery_beacon()*. Si esta función devuelve el rango de algún proceso, entonces se cancela la recepción iniciada anteriormente para dicho proceso. Esta secuencia de pasos se repite hasta que no queden más operaciones de recepción por completar.

- *Acción 6*: *contabilización de filas no procesadas*. Si al finalizar la acción anterior existen uno o varios procesos para los cuales no se inició la recepción o ésta fue cancelada, las filas no procesadas por éstos son calculadas por el resto de procesos en la siguiente iteración. Es decir, la iteración se vuelve a repetir, pero sólo se distribuyen las filas para las que no se obtuvieron resultados.

En la figura 4.5 se presenta el esqueleto completo para el proceso maestro.

Esqueleto para los procesos en los computadores fijos

El programa que se ejecuta en estos computadores realiza, básicamente, las siguientes acciones:

- *Acción 1*: *recepción inicial de datos*. Al inicio, el proceso recibe el vector *b*.
- *Acción 2*: *recepción de filas*. Por regla general, y en cada iteración, el proceso recibe el conjunto de filas de la matriz *A* enviadas por el proceso maestro. Si se recibe un señal indicando que no hay datos para calcular, el proceso finaliza.
- *Acción 3*: *calcular*. Se realiza la multiplicación de las filas recibidas por el vector de datos inicial *b*.
- *Acción 4*: *enviar resultados*. Una vez concluidos los cálculos, cada proceso envía los resultados obtenidos al proceso maestro.

El proceso en un computador fijo finaliza cuando recibe un mensaje de finalización enviado por el proceso maestro. En la figura 4.6 se presenta un esqueleto para los procesos que se ejecutan en los computadores fijos.

```
#include "lamgac.h"
int main (int argc, char *argv[])
{
    int my_id, NA, NI= 0, ND, error, pvez=1, salrm=1, n_filas=100;
    float micalc[]={...};
    double initime;

    // ... Sentencias del bloque INITIALIZE
    LAMGAC_Init_an (argv);
    //. . Acción 1: distribución inicial de datos
    N= ... // Número de filas a distribuir
    do {
        do {
            //. . Acción 2: actualización del número de procesos
            error= LAMGAC_Update (..., &NI, &ND, ...);
            // Comunicar a los procesos nuevos los datos iniciales
            // Comunicar a los procesos que se van a desvincular que finalicen el algoritmo
            // . . Acción 3: estimación del volumen de datos a distribuir
            LAMGAC_Balance (pvez, salrm, percent, NA+NI, micalc, sizeof(int)*col_A,
                                    sizeof(int), n_filas, order, ndata);
            pvez=0; inittime=MPI_Wtime();
            //. . Acción 4: distribución de filas
            //. . Acción 5: recepción controlada de resultados
            // .. Acción 6: contabilización de las filas no procesadas
        } while (filas no procesadas);
        N= ... // Actualizar el número de filas que quedan por distribuir
    } while (N>0);
    // .. Comunicar a los procesos que finalicen el algoritmo
    LAMGAC_Finalize();
    MPI_Finalize();
    return(0);
}
```

Figura 4.5 Esqueleto de ejecución para el proceso maestro

Esqueleto para los procesos en los computadores portátiles

El programa que se ejecuta en estos computadores realiza, básicamente, las siguientes acciones:

```
#include "lamgac.h"
#define DATA 1
int main (int argc, char *argv[]) {
      int my_rank;

      // ... Sentencias del bloque INITIALIZE
      LAMGAC_Init_fn ();
      // .. Acción 1: recepción inicial de datos
      do {
          // .. Acción 2: recepción de filas
          if (tag_recibido==DATA) {
                    // .. Acción 3: calcular
                    // .. Acción 4: enviar resultados
          }
      } while (señal_c==DATA);
      LAMGAC_Finalize_slave (my_rank);
      MPI_Finalize();
      return(0);
}
```

Figura 4.6 Esqueleto que ejecutan los procesos en los computadores fijos

- *Acción 1*: *recepción inicial de datos*. El proceso recibe el vector *b*.
- *Acción 2*: *recepción de filas*. Por regla general, y en cada iteración, el proceso recibe el conjunto de filas de la matriz *A* enviadas por el proceso maestro. Si se recibe una señal indicando que no hay datos para calcular, el proceso finaliza. También, si el computador solicitó desvincularse, el proceso recibirá un mensaje para que acabe la ejecución.
- *Acción 3*: *calcular*. Se realiza la multiplicación de las filas recibidas por el vector de datos inicial *b*.
- *Acción 4*: *enviar resultados*. Una vez concluidos los cálculos, cada proceso envía el resultado obtenido al proceso maestro.

El proceso finaliza cuando recibe un mensaje de finalización enviado por el proceso

```
#include "lamgac.h"
#define DATA 1
int main (int argc, char *argv[]) {
      int my_rank,NA, NI;
      MPI_Comm COMM_PARENT;

      // .. Sentencias del bloque INITIALIZE_EP
      // .. Acción 1: recepción inicial de datos
      LAMGAC_Init_pn (0,&my_rank,&NA,&NI,COMM_PARENT);
      do {
            // .. Acción 2: recepción de filas
            if (tag_recibido==DATA) {
                  // .. Acción 3: calcular
                  // .. Acción 4: enviar resultados
            }
      } while (señal_c ==DATA);
      LAMGAC_Finalize_slave (my_rank);
      MPI_Finalize();
      return(0);
}
```

Figura 4.7 Esqueleto que ejecutan los procesos en los computadores portátiles

maestro. En la figura 4.7 se presenta un esqueleto para los procesos que se ejecutan en los computadores portátiles.

4.5 Análisis del esquema de computación

En este apartado se presentan algunas consideraciones útiles que el programador debería tener en cuenta para diseñar programas con la *LAMGAC* extendida.

En primer lugar, se analizan las características del esquema de computación debido a la interacción entre el programa paralelo y la arquitectura SNMP. Cuando un nuevo proceso se expande en un computador portátil, utilizando la función *LAMGAC_Update()* o *LAMGAC_Awareness_update()*, ésta envía una notificación del tipo *lbnAppStart* hacia el computador maestro. Llegados a este punto, es necesario recordar que el envío de esta

notificación está asociada a la invocación de la función *LAMGAC_Init_pn()*, y que la función *LAMGAC_Balance()* permanece bloqueada hasta que la información referente a todos los procesos y computadores que realizan cálculos paralelos esté almacenada en la memoria compartida declarada en el computador maestro. Por tanto, el tiempo de ejecución de la función *LAMGAC_Balance()* puede verse afectado por el tiempo transcurrido desde que se invoca a la función *LAMGAC_Init_pn()* (computador portátil) hasta que la notificación llega y se procesa en el proceso *Gestor* (computador maestro). En este sentido, sería conveniente, siempre que sea posible, invocar la función *LAMGAC_Update()* con bastante antelación sobre *LAMGAC_Balance()*. De esta forma, el tiempo de espera o bloqueo en esta última función se minimiza cuando se expande un nuevo proceso en un nuevo computador portátil.

Por otro lado, al invocar la función *LAMGAC_Balance()* se puede fijar el grado de equilibrio que se desea en cada iteración, parámetro *percent*. Además, según las especificaciones de la primitiva *Equilibrar*, cuando en una iteración se alcanza el equilibrio, la distribución de datos a realizar en la siguiente iteración es la misma que la realizada en la anterior. En esta situación, la sobrecarga introducida por la función es menor debido a que los cálculos realizados en la primitiva son menos que los cálculos realizados cuando no hay equilibrio. Por tanto, existe un compromiso en la llamada a esta función. Por un lado, sería conveniente fijar un grado de equilibrio no muy severo para así alcanzar el equilibrio cuanto antes y obtener la mínima sobrecarga debido a la función *LAMGAC_Balance()*, pero por otro lado, un grado de equilibrio muy ligero implica aceptar desequilibrios considerables en los tiempos de ejecución, y por tanto, se degrada el tiempo de ejecución global de la aplicación paralela-distribuida.

Por último, cabe destacar la importancia de realizar la distribución de datos inmediatamente posterior a la invocación de la función *LAMGAC_Balance()* cuando se invoca con el parámetro *signalarm* igual a uno. Al utilizar este valor, se programa una señal del tipo SIGALRM para ser enviada un instante de tiempo después igual al tiempo empleado por el proceso mas rápido en la iteración previa. Según las especificaciones de la función *LAMGAC_Test_battery_beacon()*, hasta que esta señal no se reciba, dicha

función permanece bloqueada, y por tanto, la ejecución del proceso maestro. Si las tareas de cálculo realizadas en cada iteración son similares, y el sistema tiene un grado de equilibrio aceptable, los resultados comenzarán a llegar inmediatamente después de que se envíe la señal del tipo SIGALRM. De esta forma, es aconsejable distribuir los datos después de invocar *LAMGAC_Balance()* para comprobar si existe algún computador portátil con problemas de batería y/o sin comunicación inalámbrica justo antes de iniciar la recepción de resultados. En cualquier otro caso, la comprobación del estado de los computadores portátiles se realizaría con bastante antelación a la recepción de datos, teniendo la incertidumbre de que un recurso puede fallar durante el tiempo transcurrido desde que se hace la comprobación hasta que se inicia la recepción de resultados. Si se da esta situación, el esquema de recepción controlada de datos tendrá que cancelar la recepción de los resultados procedentes de dicho recurso después de haber iniciado la recepción, con la consecuente pérdida de tiempo. Nótese la importancia de las funciones de control de batería y enlace, ya que existe un solapamiento entre los cálculos paralelos y el control de los computadores portátiles. Mientras se realizan los cálculos paralelos, la arquitectura SNMP monitoriza y controla el estado de los computadores portátiles.

4.6 Evaluación empírica del protocolo

En este apartado se presentan algunos de los experimentos realizados para evaluar la eficiencia del protocolo de equilibrio de carga, la sobrecarga introducida por la ejecución de las funciones *LAMGAC_Balance()*, *LAMGAC_Test_battery_beacon()*, *LAMGAC_Itest_battery_beacon()* y *LAMGAC_Store_info()* y la sobrecarga de la hebra de control (*Snmp_Ping*) sobre los computadores portátiles. Para realizar los experimentos se utilizaron los computadores mostrados en la tabla 4.2, y se utilizó un punto de acceso con la especificación IEEE 802.11b para establecer la red WLAN.

Los experimentos desarrollados fueron llevados a término sobre la aplicación de la multiplicación matriz por vector presentada en la sección 4.4, utilizando un tamaño de matriz de 10000 filas por 1000 columnas de números reales. Cada experimento se repitió veinte veces, obteniéndose una desviación típica pequeña en los tiempos de ejecución.

Tabla 4.2 Características de los computadores

Computador	Procesador	Memoria (MB)	Red (Mbps)
CM	Pentium IV 2.4 Ghz	512	100
CF1	Pentium IV 2.4 Ghz	512	100
CF2	Pentium III 450 Mhz	128	100
CP1	Pentium Mobile 1.4 Ghz	1024	11
CP2	Pentium IV 2.4 Ghz	512	11

4.6.1 Eficiencia del protocolo

Para demostar la eficiencia del protocolo de equilibrio de carga se realizaron dos tipos de experimentos: a) comparar los tiempos de ejecución en una situación estacionaria, b) observar los tiempos de ejecución por iteración en una situación dinámica, donde existen computadores portátiles que se vinculan y desvinculan del entorno de computación en tiempo de ejecución. A continuación, se muestran los resultados obtenidos.

a) Situación estacionaria

El algoritmo secuencial se lanzó en el computador CM y en el computador CF2 para obtener el tiempo de ejecución más rápido y más lento, respectivamente. Para la ejecución paralela se desarrollaron dos algoritmos: el primero de ellos utiliza el protocolo de equilibrio de carga para estimar la ditribución de datos, y el segundo realiza una distribución equitativa entre todos los procesos esclavos en todas las iteraciones, es decir, en este último caso no se tiene en consideración el rendimiento de los computadores ni de las redes de comunicación. Los programas paralelos se ejecutaron utilizando todos los computadores indicados en la tabla 4.2. Los CP se vincularon a la VM_LAM al comienzo de la aplicación, y en cada computador se lanzó un proceso paralelo. Por tanto, hay cuatro procesos que realizan cálculos paralelos y uno, el proceso maestro, distribuye los datos y recoge los resultados.

La función *LAMGAC_Balance()* permite calcular la distribución de datos para un volumen determinado especificado en el parámetro *distdata*. La cantidad especificada en

este parámetro puede influir en el tiempo de ejecución global de la aplicación, ya que de este valor puede depender el número de iteraciones que se realicen, y por tanto, el número de distribuciones y tráfico generado. Se realizaron diferentes experimentos con diferentes volúmenes de datos a distribuir en cada iteración, en concreto, se hicieron experimentos para un volumen de 200, 500 y 1000 filas de la matriz por iteración. Cada iteración finaliza cuando se reciben los resultados correspondientes al volumen de datos enviado.

En la figura 4.8 se muestran los resultados de los experimentos realizados. Claramente se observa que la ejecución secuencial, tanto en el recurso más rápido como en el más lento, es mucho más rápida que la ejecución de las aplicaciones paralelas. Esta situación era de esperar debido al tiempo excesivo que toman las comunicaciones para la aplicación matriz por vector, es decir, el tiempo empleado para enviar los datos y recibir los resultados es muy superior al tiempo empleado para realizar los cálculos. En el siguiente capítulo, se presentan otras aplicaciones cuyo programa paralelo-distribuido presenta un tiempo de ejecución inferior al programa secuencial.

Con respecto a las soluciones paralelas, se observa claramente que el tiempo de ejecución de la aplicación que utiliza la biblioteca *LAMGAC* es inferior, del orden de cinco veces, a la aplicación paralela no balanceada. Esto se debe a que en cada iteración el proceso maestro espera por los resultados de todos los procesos antes de pasar a la siguiente iteración, y por tanto y para esta aplicación, siempre esperará por los procesos que se ejecutan en los CP, debido a que al ser su *throughput* de red muy inferior al de los CF, los datos enviados y los resultados recibidos tomarán un tiempo excesivo en su transmisión. Por otro lado, se aprecia que a medida que aumenta el volumen de datos (número de filas) que se distribuye en cada iteración, el tiempo de ejecución decrece. Principalmente, esto se debe a que se realizan menos iteraciones, y por tanto, en cada iteración se distribuyen más datos y se realizan más cálculos.

En la figura 4.9 se muestra la desviación típica de la media aritmética de los tiempos de ejecución de los procesos por iteración para la ejecución de ambas aplicaciones paralelas-distribuidas cuando el volumen de filas a distribuir en cada iteración es igual a

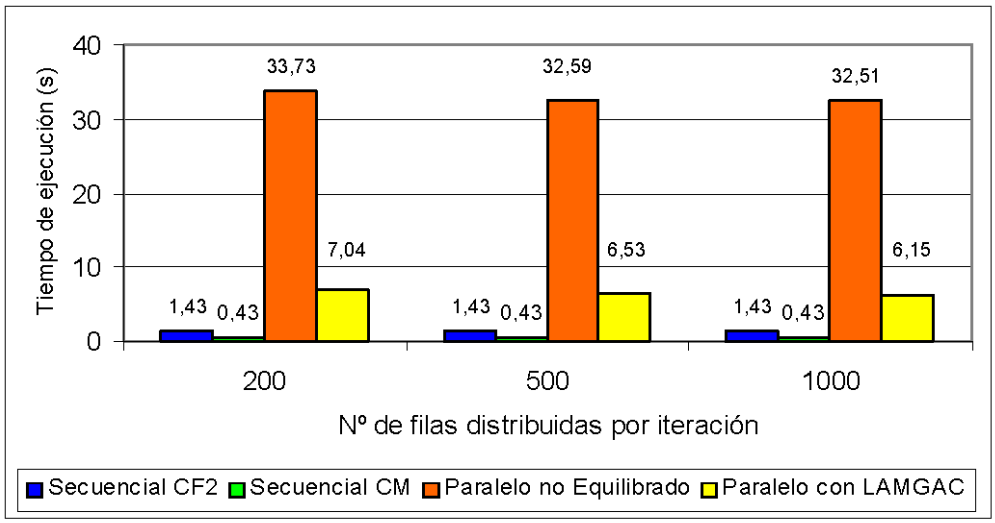


Figura 4.8 Eficiencia del protocolo de equilibrio de carga

500. Como se puede apreciar, las desviaciones típicas obtenidas en la solución paralela que utilizan el protocolo de equilibrio de carga son considerablemente inferiores que la aplicación que no utiliza dicho protocolo. Por tanto, se llega a la conclusión que los tiempos de ejecución de cada proceso en cada iteración son similares cuando se utiliza el mecanismo de equilibrio de carga, es decir, los tiempos de ejecución están equilibrados, logrando el objetivo propuesto.

b) Situación dinámica

En este experimento se pretende comprobar la eficiencia del protocolo de equilibrio de carga cuando existen vinculaciones y desconexiones de computadores portátiles durante la ejecución de la aplicación paralela. Para ello, se observa el tiempo de ejecución y la desviación típica por iteración. En el experimento llevado a cabo, el número de filas a distribuir por iteración es 500. La aplicación comienza su ejecución con el computador maestro y los dos CF. En la iteración número cinco, el computador CP1 se vincula al entorno y se expande un proceso en ella. La misma operación ocurre con el computador CP2 en la iteración 10. Finalmente, en el transcurso de la iteración 14, el proceso maestro detecta que los dos CP se sitúan fuera de cobertura respecto al punto de acceso,

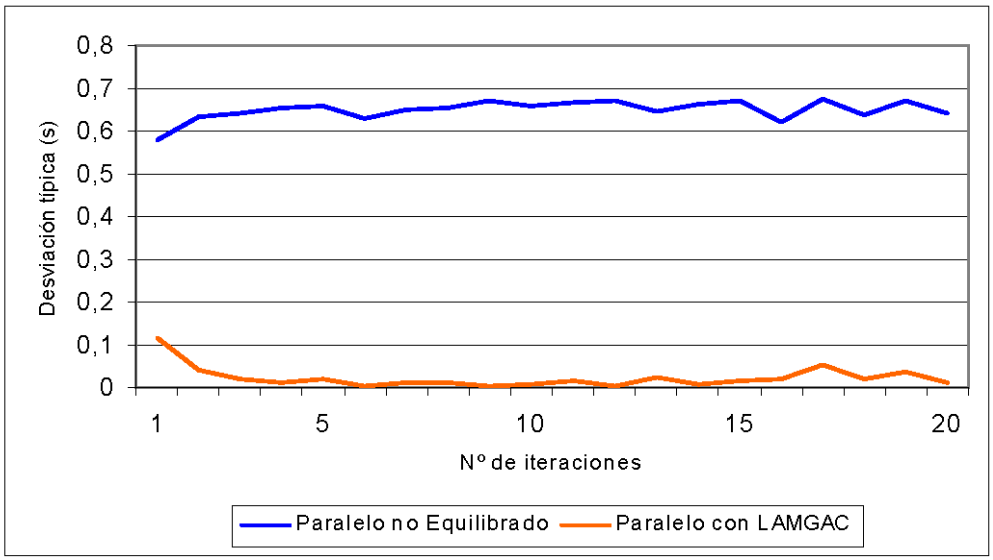


Figura 4.9 Desviación típica por iteración

y por tanto, no va a recibir los resultados calculados por éstos. Estos computadores permanecen fuera de cobertura hasta que la aplicación finalice. Como en esta iteración no se recibieron todos los resultados, en la iteración siguiente, la 15, se calculan los resultados no calculados anteriormente, es decir, se distribuye entre los CF las filas no procesadas por los computadores portátiles. Por este motivo, en esta simulación se realiza una iteración más.

En la figura 4.10 se muestra el tiempo medio de ejecución por iteración y la desviación típica. Como se puede observar, en la iteración 5 y 10, tanto la desviación típica como el tiempo de ejecución medio sufren un incremento debido a la incorporación de un nuevo proceso. Sin embargo, en las sucesivas iteraciones, estos parámetros vuelven a la normalidad concluyéndose que la situación de equilibrio se alcanza rápidamente. En la iteración 15, el tiempo de ejecución es muy pequeño debido a que se realizan los cálculos con las filas no procesadas en la iteración previa. El número de filas enviado en esta iteración es igual a 24. En las sucesivas iteraciones se sigue distribuyendo un conjunto de 500 filas entre los procesos disponibles.

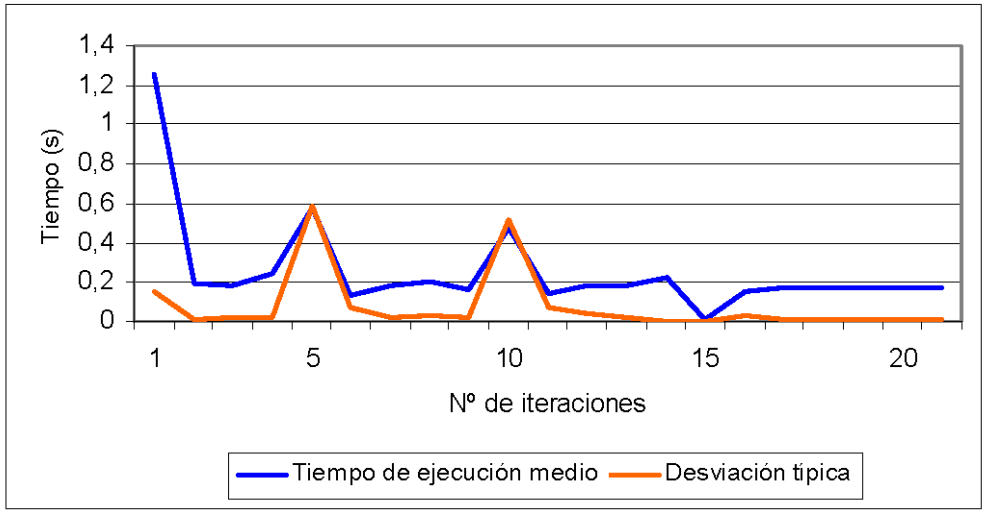


Figura 4.10 Tiempo de ejecución y desviación típica por iteración en una situación dinámica

4.6.2 Sobrecarga del tiempo de ejecución de las funciones

En el protocolo de equilibrio de carga descrito en este trabajo de investigación hay varias funciones que se invocan en cada iteración del programa paralelo: *LAMGAC_Update()*, *LAMGAC_Balance()*, *LAMGAC_Store_info()*, *LAMGAC_Test_battery_beacon()* y *LAMGAC_Itest_battery_beacon()*. Debido a la ejecución periódica de éstas, es necesario estudiar cómo afecta la sobrecarga que introducen las mismas en la ejecución del programa paralelo. El resto de funciones se invocan una sóla vez en la ejecución del programa, y además, el tiempo de ejecución consumido es despreciable frente al tiempo de ejecución global de la aplicación, por lo que no es necesario evaluarlas en este apartado.

El programa paralelo se ejecutó utilizando el computador CM, los dos CF y los dos CP. Estos últimos se vincularon a la VM_LAM al comienzo de la aplicación, y en cada uno se lanzaron varios procesos en función de las simulaciones realizadas. Los resultados que se muestran en este apartado representan el valor medio de la sobrecarga de cada función medido en cada iteración obtenido durante diez ejecuciones del programa paralelo.

La función *LAMGAC_Update()* pertenece a la implementación original de *LAMGAC* realizada en [Mac01]. En dicho trabajo se realizó la evaluación de esta función presentando una sobrecarga despreciable. Para analizar el tiempo de ejecución de la función *LAMGAC_Balance()* hay que considerar que ésta necesita la información acerca del rendimiento de todos los procesos y computadores para realizar la estimación de los datos a distribuir, y por tanto, el número de procesos que intervienen en la solución paralela influye en el tiempo de ejecución de la función. Cuando un proceso esclavo se inicia en un computador, el proceso *Colector* ubicado en dicho computador envía una notificación, del tipo *lbnAppStart*, hacia el computador maestro con la información acerca del rendimiento del computador. El tiempo transcurrido en llegar dicha notificación y ser procesada puede influir en el tiempo de ejecución de *LAMGAC_Balance()*, ya que ésta permanece bloqueada hasta que la información de todos los computadores esté disponible en la memoria compartida y sea accesible en exclusión mutua. Es por ello, que para estudiar la sobrecarga introducida por la función *LAMGAC_Balance()* se han realizado varios experimentos variando el número de procesos y teniendo en cuenta la llegada o no de notificaciones.

En la figura 4.11 se muestra la sobrecarga introducida por la función cuando no se espera la llegada de ninguna notificación, es decir, antes de invocar la función se tiene almacenada la información de todos los computadores donde se ejecutan los procesos. Además, debido a que la función *LAMGAC_Balance()* realiza cálculos diferentes en función de la iteración anterior, también se ha considerado esta situación. Recordemos que si en la iteración anterior hubo equilibrio, la función realiza la misma distribución de datos, sin embargo, si no hubo equilibrio, se calcula una nueva distribución utilizando la métrica 3.6. Como se puede apreciar, el tiempo de ejecución crece linealmente con el número de procesos, pero con una pendiente muy suave, existiendo un incremento alrededor de 0,5 ms en la ejecución de 20 procesos respecto a 2 procesos. Por tanto, se puede concluir que el número de procesos no influye de forma significativa en el tiempo de ejecución de la función (buena escalabilidad). Además, se aprecia una diferencia de aproximadamente 0,3 ms en la ejecución de la función cuando existe equilibrio en la

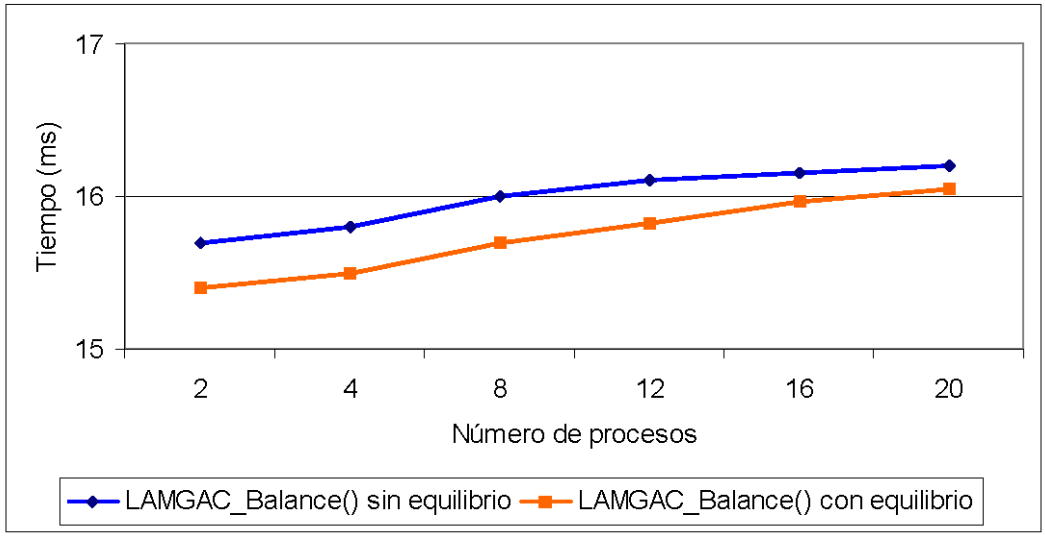


Figura 4.11 Tiempo de ejecución de LAMGAC_Balance() cuando no se esperan notificaciones

iteración anterior y cuando no lo hay.

Por otro lado, en la figura 4.12 se muestra la sobrecarga introducida por la función cuando se esperan notificaciones desde otros computadores, es decir, antes de invocar la función no se dispone de la información de todos los computadores. Para estudiar este tiempo, se ha medido el tiempo de ejecución de la función *LAMGAC_Balance()* en la primera iteración del programa paralelo. Se ha elegido esta iteración porque representa la situación más desfavorable. En esta iteración, el proceso *Gestor* recibe todas las notificaciones enviadas desde el resto de computadores, es decir, han de llegar tantas notificaciones como procesos esclavos se han iniciado. Como se puede apreciar, el tiempo de ejecución sigue una evolución ascendente y casi lineal con el número de procesos, desde 1/3 de segundo para dos procesos hasta un valor ligeramente inferior a los 2 segundos para veinte procesos. Esto se debe principalmente a que la función *LAMGAC_Balance()* permanece bloqueada debido a la llegada masiva de notificaciones que son procesadas secuencialmente por el proceso *Gestor*. Este tiempo varía en función de la distancia temporal entre la llamada a las funciones *LAMGAC_Init_fn()* y *LAMGAC_Init_pn()* en los CF y CP, respectivamente, y la invocación de

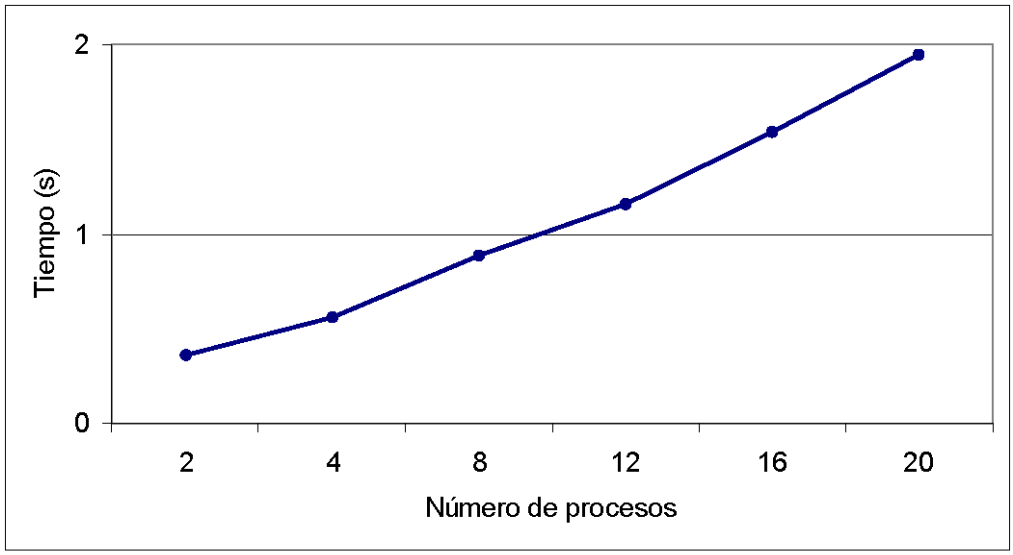


Figura 4.12 Tiempo de ejecución de LAMGAC_Balance() cuando se esperan notificaciones

LAMGAC_Balance() en el CM, como ya se analizó en el apartado 4.5. Resulta evidente que entre mayor es esta distancia temporal, menor es el tiempo de sobrecarga de la función, debido a que las notificaciones son recibidas con antelación. Incluso, si la distancia temporal es lo suficientemente grande para recibir todas las notificaciones antes de invocar la función *LAMGAC_Balance()*, la sobrecarga se aproximaría a los resultados mostrados en la figura 4.11. Cada vez que la función *LAMGAC_Balance()* tiene que esperar por la llegada de notificaciones, el tiempo de ejecución de ésta puede incrementarse. Sin embargo, esta situación solamente puede ocurrir al comienzo de la aplicación paralela-distribuida o cuando se expande un nuevo proceso en un computador portátil o cuando un computador desea desvincularse del entorno de computación. Por tanto, la frecuencia de repetición de esta situación suele ser bastante reducida. Además, la sobrecarga introducida se ve compensada con la mejora del tiempo de ejecución global de la aplicación paralela-distribuida.

Para analizar el tiempo de ejecución de las funciones *LAMGAC_Store_info()*, *LAMGAC_Test_battery_beacon()* y *LAMGAC_Itest_battery_beacon()* hay que considerar que la tarea realizada por éstas es muy sencilla (son muy ligeras), y por tanto, sus tiempos de ejecución no deberían ser influenciados de forma significativa por el

número de procesos. La primera actualiza la información referente al tiempo de ejecución empleado y la carga procesada por el proceso cuyo identificador se pasa por parámetros. Las otras dos funciones buscan en la estructura de datos de la memoria compartida si algún proceso en un computador portátil está situado fuera de cobertura o su batería se agota en breve. En la figura 4.13 se muestra su tiempo de ejecución, y como se aprecia permanecen casi constante con la variación del número de procesos (buena escalabilidad). Las funciones *LAMGAC_Test_battery_beacon()* y *LAMGAC_Itest_battery_beacon()* realizan la misma tarea con la salvedad de que una es bloqueante y la otra no. El tiempo que pueda permanecer bloqueada la función *LAMGAC_Test_battery_beacon()* no depende de la implementación de la propia primitiva *Comprobar_Batería_y_Enlace*, por tanto, la sobrecarga es igual en ambas funciones, y por eso sólo se muestra la primera. Hay que tener en cuenta que el orden de los tiempos de ejecución es de unidades de microsegundos, y aunque la función *LAMGAC_Store_info()* se invoca cada vez que se reciben los resultados enviados por un proceso paralelo, la sobrecarga introducida por la ejecución de estas funciones es despreciable.

4.6.3 Sobrecarga de la hebra de control de los computadores portátiles

Para evaluar la sobrecarga de la hebra de control de los computadores portátiles, se realizaron experimentos situándonos en el peor caso posible, es decir, se forzó la ejecución de las hebras desde el comienzo de la aplicación paralela hasta el final. Cada hebra consulta el valor del objeto *lbWirLink* cada segundo. Dado que para una distribución de filas por iteración igual a 500, el tiempo de ejecución de la aplicación paralela es aproximadamente 6,5 segundos, cada hebra consultará dicho objeto 6 veces durante la ejecución de la aplicación. Por tanto, y teniendo en cuenta que el tiempo de ejecución viene determinado por el tiempo de comunicación, la sobrecarga introducida por la ejecución de la hebra no es apreciable. Aún así, en la figura 4.14 se muestra el tiempo de ejecución cuando se ejecutan un número determinado de hebras. Para las ejecuciones con un número de hebras superior a dos, se ha supuesto que el número de

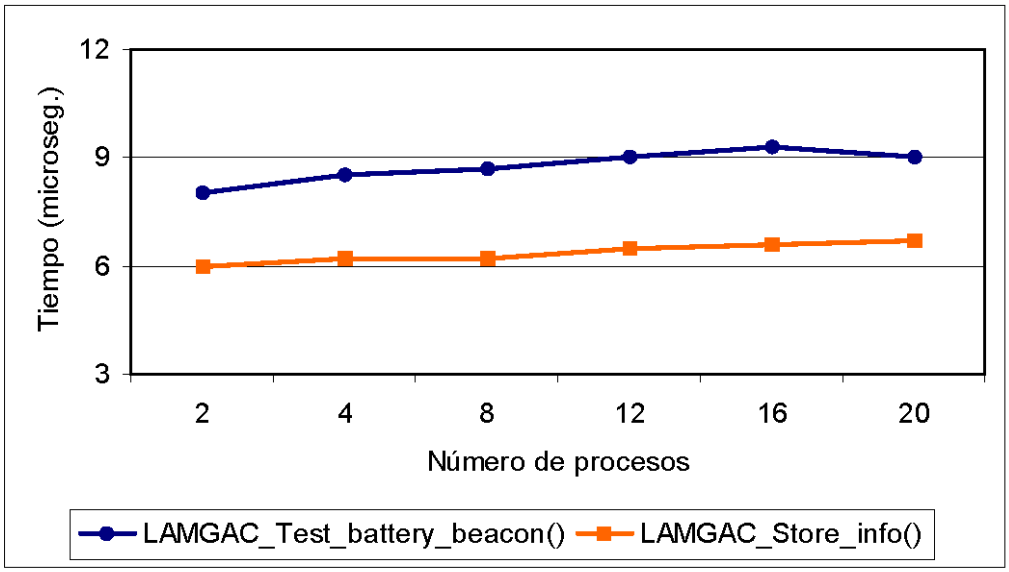


Figura 4.13 Tiempo de ejecución de LAMGAC_Test_battery_beacon() y LAMGAC_Store_info()

computadores portátiles a controlar es superior, y para ello se replicaron tantas hebras como número de CP a monitorizar. Como se puede apreciar, el tiempo de ejecución es prácticamente similar en todas las simulaciones realizadas, y por tanto, para esta aplicación, se puede concluir que es prácticamente independiente del número de hebras, no afectando al tiempo de ejecución de los procesos.

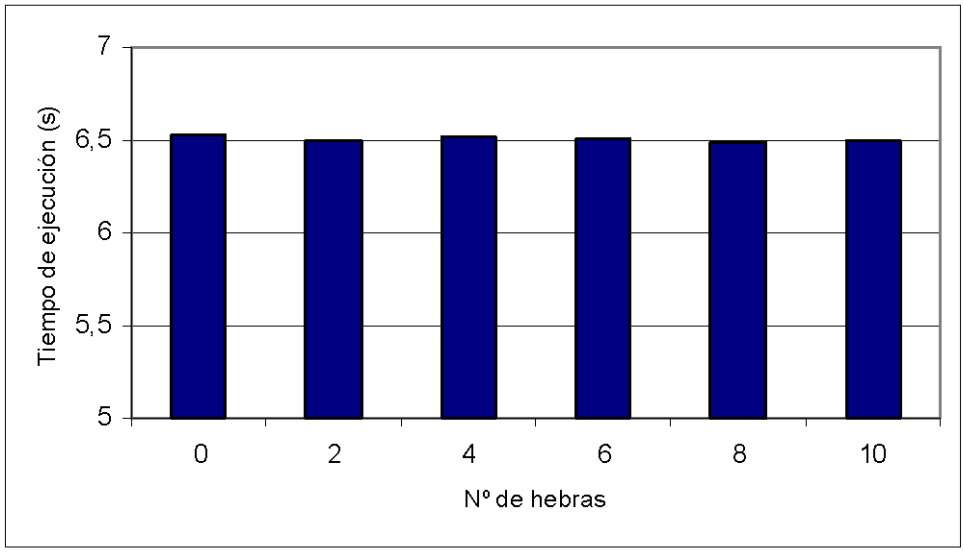


Figura 4.14 Sobrecarga de la hebra de control sobre los computadores portátiles

5. Resultados experimentales

En este capítulo se muestran los resultados más significativos obtenidos al resolver cuatro aplicaciones diferentes sobre un entorno de computación LAN-WLAN. Para cada aplicación se demuestra la eficiencia del protocolo de equilibrio de carga desarrollado en este trabajo de investigación.

5.1 Introducción

En el capítulo anterior se presentó la ampliación del *middleware LAMGAC* para permitir al programador realizar tanto distribuciones de datos equilibradas entre los procesos paralelos como la gestión de la variación dinámica de computadores en tiempo de ejecución. Para verificar la eficiencia del protocolo de equilibrio de carga desarrollado en este trabajo de investigación, en este capítulo se presentan los resultados experimentales más importantes obtenidos al ejecutar cuatro aplicaciones paralelas-distribuidas utilizando nuestro mecanismo de equilibrio de carga sobre un entorno de computación y comunicación heterogéneo LAN-WLAN. En concreto, en primer lugar se analiza la aplicación del producto de dos matrices. A continuación se expone el método de relajación de Jacobi. En tercer lugar se presentan los resultados obtenidos sobre la fase de estimaciones de la herramienta de codiseño Hw/Sw desarrollada en nuestro grupo de investigación. Por último, se presentan los resultados obtenidos al resolver el problema de la simulación de un canal infrarrojo difuso en interiores.

5.2 Entorno de computación LAN-WLAN

Los experimentos fueron llevados a término en una red formada por CF y CP bajo el sistema operativo LINUX, tal como se muestra en la figura 1 del capítulo 2. Los primeros utilizan para la comunicación de datos el protocolo IEEE 802.3u (*Fast-Ethernet*), y los computadores portátiles utilizan el protocolo IEEE 802.11b en modo infrastructura. La comunicación entre los CP y el CM se realiza a través de un punto de acceso inalámbrico. En este entorno se dispone de cinco CF, de los cuales uno ejerce de maestro, y de tres CP. Las características de los recursos de computación y de comunicación se muestran en la tabla 5.1 y 5.2, respectivamente. Se utilizaron dos agrupaciones diferentes de computadores para realizar los experimentos, las cuales se especifican en la tabla 5.3, donde en cada computador sólo se ejecuta un proceso del programa paralelo. Los CP especificados en cada agrupación pueden vincularse y desvincularse del entorno de computación en tiempo de ejecución.

Tabla 5.1 Características de los recursos de computación

Computador	Procesador	Memoria (MB)	Red (Mbps)	Kernel
CM	Pentium IV 2.4 Ghz	512	100	2.4.20
CF1	Pentium IV 2.4 Ghz	512	100	2.4.20
CF2	Pentium IV 2.4 Ghz	512	100	2.6.12
CF3	Pentium III 1 Ghz	256	100	2.6.10
CF4	Pentium III 450 Mhz	128	100	2.2.19pre17
CP1	Pentium Mobile 1.4 Ghz	1024	11	2.6.12
CP2	Pentium Mobile 1.4 Ghz	1024	11	2.6.12
CP3	Pentium IV 2.4 Ghz	768	11	2.4.21

Tabla 5.2 Características de los recursos de comunicación

Recurso	Modelo	Características
Conmutador	Ovislink Ethernet Switch Live FSH8G	9 puertos RJ-45: 8 puertos 10/100 + 1 Gigabit
Punto de acceso	HP Procurve Wireless Access Point 520wl	1 puerto RJ-45 10/100 Mbps 2 slots PCMCIA para tarjetas IEEE 802.11a/b/g

Tabla 5.3 Agrupaciones de computadores

Agrupación	Computadores
C1	CM, CF1, CF2, CF3, CF4, CP1, CP2, CP3
C2	CM, CF1, CF4, CP1, CP3

Respecto al software utilizado, en cada máquina se ha instalado la distribución de MPI-2, LAM/MPI 6.5.9 [Web-8], para implementar la comunicación entre procesos, y la distribución de SNMP, Net-SNMP 5.0.8 [Web-4], con las especificaciones del agente SNMP (proceso *Colector*) mostradas en la sección 3.2.1 para implementar la arquitectura SNMP de control y monitorización.

Para demostrar la eficiencia del protocolo de equilibrio de carga se realizaron diferentes simulaciones que abarcan los siguientes aspectos: a) comparación de los tiempos de ejecución secuencial, paralelo con mecanismo de equilibrio de carga y paralelo con técnica de planificación equilibrada por bloques, b) estudio de la desviación

típica del tiempo de ejecución por iteración, c) adaptación del mecanismo de equilibrio de carga frente a vinculaciones de nuevos procesos y no disponibilidad de CP, y d) sobrecarga de la hebra de control *SnmpPing*. Cada una de estas simulaciones se realizó veinte veces, obteniéndose una desviación típica muy pequeña para todos los experimentos.

5.3 Multiplicación de matrices

Dada una matriz A de N filas y M columnas y, otra matriz B de M filas y P columnas, el problema del álgebra lineal del cálculo de la multiplicación matriz por matriz se resuelve de la siguiente forma:

$$A \times B = C$$

Donde:

- C es la matriz resultado de la multiplicación. Su dimensión es igual al número de filas de la matriz A por el número de columnas de la matriz B (N * P).

Los elementos de las matrices son números reales, esto es:

$$\begin{aligned} &A[i][j] \in \Re \\ &B[j][k] \in \Re \qquad \forall i,j \in \aleph \\ &C[i][k] \in \Re \end{aligned}$$

Donde:

- *A[i][j]* es un elemento de la matriz A,
- *B[j][k]* es un elemento del matriz B, y
- *C[i][k]* es un elemento de la matriz resultado.

El elemento *C[i][k]* se obtiene sumando los productos parciales de los elementos de la fila i de la matriz A por los elementos de la columna k de la matriz B.

En la figura 5.1 se muestra el código secuencial de la multiplicación matriz por matriz. Con respecto al programa paralelo, varios autores han propuesto soluciones para resolver de forma óptima la multiplicación de matrices sobre una red de computadores

```
for (i=0; i<N; i++) {
      for (k=0; k<P; k++) {
          C[i][k]=0;
          for (j=0; j<M; j++)
                    C[i][k]=C[i][k] + A[i][j]*B[j][k];
      }
}
```

Figura 5.1 Código secuencial para el producto matriz por matriz

heterogéneos [BBR01][KaL01][TDQ01]. En ninguno de estos trabajos se considera la red de comunicación como heterogénea y tampoco es posible la variación dinámica de procesos en tiempo de ejecución. Además, realizan una asignación de carga estática al comienzo de la aplicación en función de la potencia de procesamiento de cada computador. Por tanto, en un entorno LAN-WLAN no dedicado donde la variación de rendimiento de la red inalámbrica es aleatoria y significativa y el número de CP puede variar en tiempo de ejecución, se obtendrán desequilibrios de carga a lo largo de la ejecución de la aplicación paralela-distribuida. En este sentido, en este trabajo no estamos interesados en obtener una solución óptima de la multiplicación de matrices, sino verificar la eficiencia de protocolo de equilibrio de carga diseñado en este trabajo de investigación. De hecho, la ejecución paralela de la multiplicación de matrices es muy interesante en lo que se refiere al equilibrio de carga debido a que el tiempo de comunicación es mucho mayor que el tiempo de cálculo si se trabajan con matrices de órdenes elevados. Por este motivo, para esta aplicación se han realizado tres implantaciones diferentes cuya diferencia principal reside en la forma de distribuir la carga desde el proceso maestro hacia los procesos esclavos. Una breve descripción de las implantaciones se muestra a continuación:

- *Implantación A*. En cada iteración, el proceso maestro replica una columna de la matriz *B* en todos los procesos esclavos. A continuación, se distribuyen las filas de la matriz *A* entre los procesos esclavos teniendo en cuenta el número de filas que le toca a cada uno según el resultado devuelto por la función

LAMGAC_Balance(). Este proceso se repite hasta que no queden columnas de la matriz B por procesar. Esta versión es la menos óptima de todas ya que en cada iteración se envía una porción de información de la matriz A a cada proceso, y dicha información puede ya haber sido recibida parcial o completamente en iteraciones previas. Sin embargo, con esta implantación se demuestra la eficiencia del mecanismo de equilibrio de carga cuando el tiempo empleado para las comunicaciones es considerablemente mayor que el empleado para el cálculo.

- *Implantación B*. El proceso maestro replica la matriz A en todos los procesos esclavos al comienzo de la aplicación y a cada proceso que se ejecute en un CP cuando éste se vincule al entorno de computación. A continuación, y en cada iteración, el proceso maestro replica una columna de la matriz B en todos los procesos esclavos, y después de invocar a la función *LAMGAC_Balance()* se le indica a cada proceso el subconjunto de filas con el que debe operar. Esta versión es más óptima que la anterior debido a que los datos se comunican sólo una vez. Sin embargo, pueden existir computadores que posean al completo toda la matriz A, pero luego sólo realizan cálculos con un subconjunto reducido de filas, con la consecuente pérdida de tiempo en el envío inicial de filas.

- *Implantación C*. Esta versión es una mezcla de las dos versiones anteriores. En cada iteración, el proceso maestro replica una columna de la matriz B en todos los procesos esclavos. A continuación, y después de invocar a la función *LAMGAC_Balance()* el proceso maestro conoce el número de filas con las que debe operar cada proceso esclavo, y por tanto, les indicará el subconjunto de filas con las que deben operar si han sido distribuidas en iteraciones previas, y si no, les enviará las filas correspondientes. De las tres implantaciones, ésta es la más eficiente porque cada proceso esclavo dispone de las filas necesarias para operar y cada fila sólo se comunica una vez a un mismo proceso.

A continuación, y siguiendo el modelo de programación presentado en la sección 3

del capítulo anterior, se describe para cada una de las implantaciones descritas el funcionamiento de los tres esqueletos exponiendo las acciones principales que efectúan.

5.3.1 Implantación A

Para esta implantación se presentan las acciones principales que realiza cada proceso.

Esqueleto para el proceso maestro

El programa del proceso maestro realiza, básicamente, las siguientes acciones:

- *Acción 1*: *replicación de una columna de la matriz B*. Al comienzo de cada iteración el proceso maestro comunica una columna de la matriz *B* a todos los procesos vinculados al entorno de computación.
- *Acción 2*: *actualización del número de procesos*. Se invoca la función *LAMGAC_Update()* para actualizar el número de procesos en los CP. A continuación, y si un nuevo proceso se expande en un CP, hay que comunicarle los datos iniciales necesarios para que pueda realizar los cálculos. Entre otros, se le comunica la columna correspondiente de la matriz *B*. Si existe algún proceso en un CP que se quiere desvincular, se le envía un mensaje para que acabe su ejecución.
- *Acción 3*: *estimación del volumen de filas a distribuir*. Después de conocer los cambios producidos en la VM_LAM, se invoca la función *LAMGAC_Balance()* para obtener el número de filas a distribuir a cada proceso y la secuencia de envío a éstos. El valor de los parámetros *sizedata* y *sizeresults* de esta función coincide con el tamaño de una fila de la matriz (*sizeof(float)*M*) y con el tamaño de un número real (*sizeof(float)*), respectivamente.
- *Acción 4*: *distribución de filas*. Después de conocer la cantidad de filas a enviar a cada proceso, éstas se distribuyen según la secuencia de envío obtenida en la acción anterior. El tamaño de esta comunicación de datos es la

suma del número de *bytes* correspondientes a los elementos de cada fila a enviar. Debido a la gran cantidad de datos a enviar, la distribución de filas se implementa con rutinas no bloqueantes para evitar el tiempo ocioso en los computadores con enlaces de comunicación rápidos mientras esperan que los datos sean recibidos en los computadores con enlaces con *throughput* bajo. Por otro lado, si en una iteración dada no se distribuyen filas a uno o varios procesos, a éstos se les envía un mensaje MPI con una etiqueta, que denominamos señal, para que no realicen operaciones de cálculo y pasen a la espera en la siguiente iteración. Esto es motivado porque podría ocurrir que durante algunas determinadas iteraciones la función *LAMGAC_Balance()* prescindiera de distribuir carga a un determinado proceso ya que en ese instante su capacidad de procesamiento está muy por debajo del resto. Sin embargo, si las condiciones del entorno varían dicho proceso puede ser considerado otra vez por el mecanismo de equilibrio de carga.

En la figura 5.2 se presenta un ejemplo de distribución de datos: el número de filas de la matriz *A* es nueve y en la iteración i-ésima el número de procesos en los CF y CP es NA=2 (P1 y P2) y NI=1 (P3), respectivamente. En esta iteración, la llamada a la función *LAMGAC_Balance()*, con el parámetro *distdata* igual a 9, devuelve como resultado que a los procesos P1, P2 y P3 se les debe enviar cinco, cero y cuatro filas de la matriz *A*, respectivamente (vector *ndata*). La secuencia de envío de datos se realiza en el orden siguiente: P1 y P3, y a P2 se le envía un mensaje diciéndole que no se le envían datos (vector *order*). En la siguiente iteración, el proceso P3 solicita desvincularse y por tanto, se le envía un mensaje para que acabe su ejecución. La invocación a *LAMGAC_Balance()* devuelve un número de seis filas a enviar al proceso P1 y tres filas al proceso P2. En esta última iteración, el mecanismo de equilibrio de carga consideró necesario mandar datos al proceso P2 para evitar que toda la carga sea procesada por el proceso P1.

- *Acción 5*: *recepción controlada de resultados*. Una vez realizada la

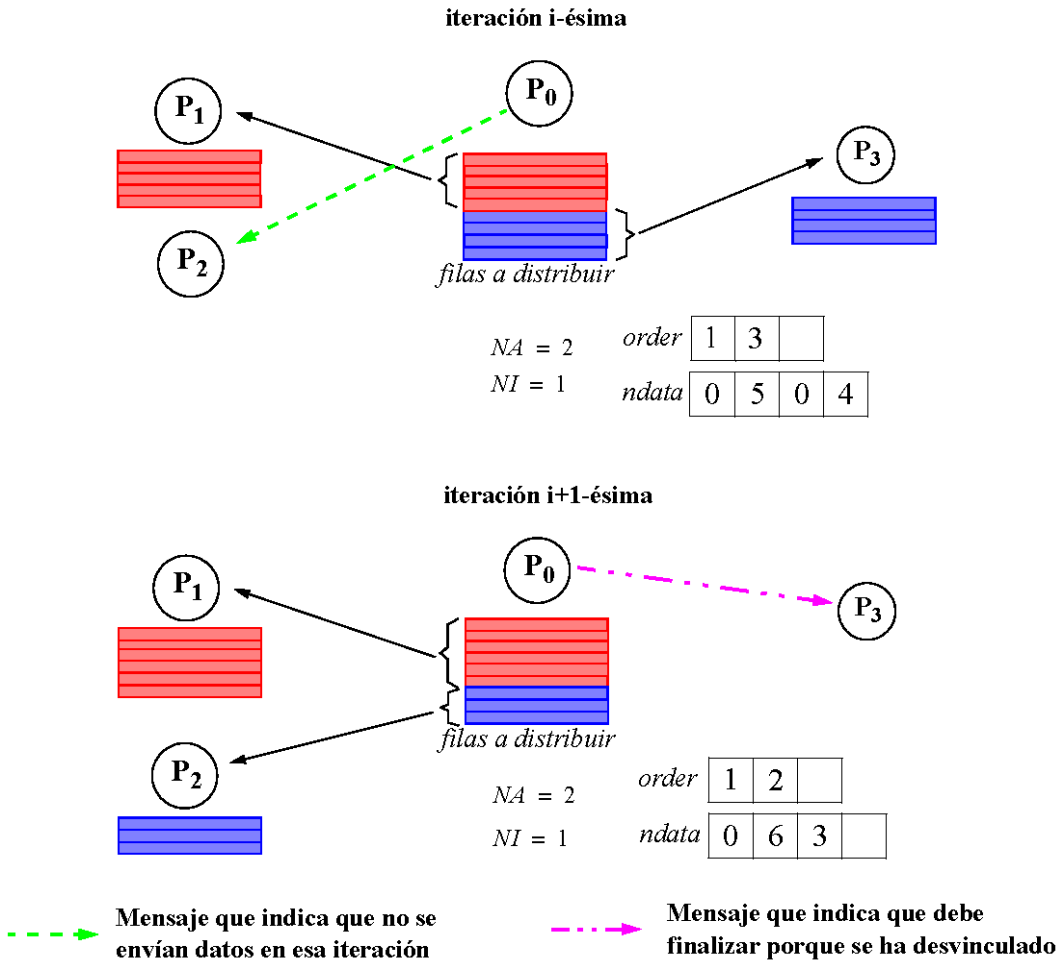


Figura 5.2 Ejemplo de distribución de filas

distribución de filas, se implementa el mecanismo de recepción controlada explicado en la sección 2 del capítulo anterior para recibir los datos calculados por los procesos esclavos. Durante esta recepción, el proceso maestro obtiene de los procesos esclavos la suma de los productos parciales de los elementos de cada fila por los elementos de la columna de la matriz *B*.

- *Acción 6*: *contabilización de filas no procesadas y envío de señal.* Una vez finalizada la acción anterior, se conoce si hay uno o varios CP de los cuales no se recibieron resultados porque estaban fuera de cobertura o la energía de la batería no era suficiente para acabar la iteración en curso. En el caso de existir

computadores en esta situación, los datos que estaban destinados para los procesos que se ejecutan en ellos serán redistribuidos entre el resto de procesos, es decir, la iteración se vuelve a repetir con la misma columna de la matriz B, pero sólo se distribuyen las filas para las que no se obtuvieron resultados. Por tanto, se envía una señal, mensaje MPI con una etiqueta, a todos los procesos esclavos que se ejecutan en el resto de computadores para indicarles que van a volver a recibir una distribución de filas para realizar cálculos con la columna actual de la matriz B o, en el caso de que se hayan procesado todas las filas, se les indica que deben pasar a la siguiente iteración.

En la figura 5.3 se presenta el esqueleto completo para el proceso maestro.

Esqueleto para los procesos en los CF

El programa que se ejecuta en estos computadores realiza, básicamente, las siguientes acciones:

- *Acción 1*: *recepción de la columna de la matriz B*. En cada iteración del programa paralelo el proceso recibe una columna de la matriz B.
- *Acción 2*: *recepción de filas*. En cada iteración, el proceso recibe un mensaje donde una señal le indica que van adjuntas un conjunto de filas de la matriz A para realizar los cálculos, o en caso contrario no se habrán enviado filas para calcular, y el proceso debe saltar a la siguiente iteración.
- *Acción 3*: *calcular*. Se realiza la multiplicación de las filas recibidas por la columna actual de la matriz B.
- *Acción 4*: *enviar resultados*. Una vez concluidos los cálculos, cada proceso envía los resultados obtenidos al proceso maestro.
- *Acción 5: recepción de señal sobre repetición de iteración*. El proceso recibe una señal del proceso maestro que le indica si hay que repetir los cálculos con la misma columna de la matriz B y una nueva distribución de filas o se pasa a la siguiente iteración. Las nuevas filas que se recibirán son parte del conjunto

```
#include "lamgac.h"
int main (int argc, char *argv[])
{
    int my_id, NA, NI= 0, ND;
    float micalc[]={...};

    // .. Sentencias del bloque INITIALIZE
    LAMGAC_Init_an (argv);
    // .. Comunicación de datos iniciales
    for (i=0;i<P;i++) {
    //. . Acción 1: replicación de una columna de la matriz B
        do {
        //. . Acción 2: actualización del número de procesos
            LAMGAC_Update (..., &NI, &ND, ...);
            // Comunicar a los procesos nuevos los datos iniciales y la columna actual de B
            // Comunicar a los procesos que se van a desvincular que finalicen el algoritmo
        //. . Acción 3: estimación del volumen de filas a distribuir
            LAMGAC_Balance (..., micalc, sizeof(float)*M, sizeof(float), N, order, ndata);
        //. . Acción 4: distribución de filas
        //. . Acción 5: recepción controlada de resultados
        //. . Acción 6: contabilización de las filas no procesadas y envío de señal
        } while (filas no procesadas);
    }
    LAMGAC_Finalize();
    MPI_Finalize();
    return(0);
}
```

Figura 5.3 Esqueleto de ejecución para el proceso maestro en la implantación A

de filas que uno o varios CP no procesaron o no pudieron enviar los resultados obtenidos con ellas.

En la figura 5.4 se presenta un esqueleto para los procesos que se ejecutan en los CF.

Esqueleto para los procesos en los CP

El programa que se ejecuta en estos computadores realiza, básicamente, las mismas acciones que los procesos que se desarrollan en los CP, con la salvedad de que se recibe la iteración actual del bucle principal y en la acción 2 se tiene en cuenta si el proceso ha

```
#include "lamgac.h"
int main (int argc, char *argv[]) {
      señal_c=CALCULAR;
      // .. Sentencias del bloque INITIALIZE
      LAMGAC_Init_fn ();
      // .. Recepción de datos iniciales
      for (i=0;i<P;i++) {
      // .. Acción 1: recepción de la columna de la matriz B
          do {
          // .. Acción 2: recepción de filas
              if (señal_c==CALCULAR) {
                // .. Acción 3: calcular
                // .. Acción 4: enviar resultados
                // .. Acción 5: recepción de señal sobre repetición de iteración
               }
          } while (repetir iteración);
      }
      LAMGAC_Finalize_slave (my_rank);
      MPI_Finalize();
      return(0);
}
```

Figura 5.4 Esqueleto de los procesos en los CF en la implantación A

solicitado su desvinculación. A continuación se describe dicha acción:

- *Acción 2*: *recepción de filas*. En cada iteración, el proceso recibe el conjunto de filas de la matriz *A* enviadas por el proceso maestro siempre y cuando la señal adjunta al mensaje indique que se tienen que realizar los cálculos. En caso contrario, pasará a la espera en la siguiente iteración. Sin embargo, si el proceso ha solicitado desvincularse del entorno, éste recibirá otra señal para que finalice su ejecución.

En la figura 5.5 se presenta un esqueleto para los procesos que se ejecutan en los CP.

```
#include "lamgac.h"
int main (int argc, char *argv[]) {
    señal_c=CALCULAR;
    // ... Sentencias del bloque INITIALIZE_EP
    LAMGAC_Init_pn (...);
    // .. Recepción de datos iniciales
    i=iter_actual;
    while((i<P) && (señal_c!=DESVINCULAR)) {
    // .. Acción 1: recepción de la columna de la matriz B
        do {
        // ... Acción 2: recepción de filas
            if (señal_c==CALCULAR) {
                // ... Acción 3: calcular
                // ... Acción 4: enviar resultados
                // ... Acción 5: recepción de señal sobre repetición de iteración
            }
        } while (repetir iteración);
        i++;
    }
    LAMGAC_Finalize_slave (my_rank);
    MPI_Finalize();
    return(0);
}
```

Figura 5.5 Esqueleto de los procesos en los CP en la implantación A

5.3.2 Implantación B

Para esta implantación se presentan las acciones principales que realiza cada proceso.

Esqueleto para el proceso maestro

Las acciones que realiza el proceso maestro son similares a las del proceso maestro de la implantación A. Sin embargo, para mayor claridad, éstas se describen a continuación:

- *Acción 1*: *replicación de la matriz A*. Inicialmente se comunica la matriz *A* a todos los procesos vinculados al entorno de computación.

- *Acción 2: replicación de una columna de la matriz B.* Al comienzo de cada iteración el proceso maestro comunica una columna de la matriz *B* a todos los procesos vinculados al entorno de computación.

- *Acción 3*: *actualización del número de procesos.* Se invoca la función *LAMGAC_Update()* para actualizar el número de procesos en los CP. A continuación, y si un nuevo proceso se expande en un CP, hay que comunicarle los datos iniciales necesarios para que pueda realizar los cálculos. Entre otros, se le comunica la matriz *A* y la columna correspondiente de la matriz *B*. Si existe algún proceso en un CP que se quiere desvincular, se le envía un mensaje para que acabe su ejecución.

- *Acción 4*: *estimación del número de filas que debe procesar cada esclavo.* Después de conocer los cambios producidos en la VM_LAM, se invoca la función *LAMGAC_Balance()* para obtener el número de filas que tiene que procesar cada esclavo y la secuencia de envío a éstos. Debido a que en esta implantación el proceso maestro tiene que indicar a cada proceso esclavo el rango de filas con las que debe operar (dos números enteros) y recibe de éstos un número entero por cada fila procesada, no existe una relación directa entre los parámetros *sizedata* y *sizeresults* de la función *LAMGAC_Balance()*. Por tanto, el parámetro *sizedata* se fija al valor 1 (mínimo valor), y *sizeresults* al valor que le corresponde, *sizeof(float)*.

- *Acción 5*: *distribución del rango de filas.* Después de conocer la cantidad de filas con las que debe operar cada proceso esclavo, se envía a cada uno el rango de filas de la matriz *A* con las que debe operar cada uno. El tamaño de esta comunicación de datos es el número de *bytes* ocupados por dos números enteros (límite inferior y superior del rango de filas). Al igual que en la implantación *A*, si en una iteración dada no se le va a asignar trabajo a uno o varios procesos, a éstos se les envía una señal para que no realicen operaciones de cálculo y pasen a la espera en la siguiente iteración.

- *Acción 6*: *recepción controlada de resultados.* En esta acción se implanta el

mecanismo de recepción controlada explicado en la sección 2 del capítulo anterior para recibir los datos calculados por los procesos esclavos.

- *Acción 7*: *contabilización de filas no procesadas y envío de señal.* Al igual que en la implantación A, en el caso de existir filas no procesadas debido a CP no disponibles, los cálculos correspondientes a éstas se distribuyen entre el resto de procesos disponibles. En este caso, se vuelven a repetir las acciones desde la acción 3 con las filas no procesadas. Cuando todas las filas sean procesadas se pasa a la siguiente iteración.

En la figura 5.6 se presenta el esqueleto completo para el proceso maestro.

Esqueleto para los procesos en los CF

El programa que se ejecuta en estos computadores computadores realiza, básicamente, las siguientes acciones:

- *Acción 1*: *recepción de la matriz A.* Al comienzo de la aplicación el proceso recibe la matriz A completa.
- *Acción 2*: *recepción de la columna de la matriz B.* En cada iteración del programa paralelo el proceso recibe una columna de la matriz B.
- *Acción 3*: *recepción de los límites del subconjunto de filas.* En cada iteración, el proceso recibe un mensaje donde una señal le indica que va adjunto el límite inferior y superior del subconjunto de las filas con las que debe realizar los cálculos o, en caso contrario, no se habrán enviado filas para calcular y el proceso debe pasar a la siguiente iteración.
- *Acción 4*: *calcular.* Se realiza la multiplicación de las filas por la columna actual de la matriz B.
- *Acción 5*: *enviar resultados.* Una vez concluidos los cálculos, cada proceso envía los resultados obtenidos al proceso maestro.
- *Acción 6: recepción de señal sobre repetición de iteración.* El proceso recibe

```
#include "lamgac.h"
int main (int argc, char *argv[])
{
      int my_id, NA, NI= 0, ND;
      float micalc[]={...};

      // .. Sentencias del bloque INITIALIZE
      LAMGAC_Init_an (argv);
      // .. Comunicación de datos iniciales
      //. . Acción 1: replicación de la matriz A
      for (i=0;i<P;i++) {
      //. . Acción 2: replicación de una columna de la matriz B
          do {
          //. . Acción 3: actualización del número de procesos
                LAMGAC_Update (..., &NI, &ND, ...);
                // Comunicar a los procesos nuevos los datos iniciales, matriz A y columna actual de B
                // Comunicar a los procesos que se van a desvincular que finalicen el algoritmo
          // . . Acción 4: estimación del número de filas que debe procesar cada esclavo
                LAMGAC_Balance (..., micalc, 1, sizeof(float), N, order, ndata);
          //. . Acción 5: distribución del rango de filas
          //. . Acción 6: recepción controlada de resultados
          //. . Acción 7: contabilización de las filas no procesadas y envío de señal
          } while (filas no procesadas);
      }
      LAMGAC_Finalize();
      MPI_Finalize();
      return(0);
}
```

Figura 5.6 Esqueleto de ejecución para el proceso maestro en la implantación B

una señal del proceso maestro que le indica si hay que repetir los cálculos con la misma columna de la matriz B y un nuevo rango de filas o se pasa a la siguiente iteración.

En la figura 5.7 se presenta un esqueleto para los procesos que se ejecutan en los CF.

Esqueleto para los procesos en los CP

Al igual que en la implantación A, las acciones realizadas por estos procesos son las

```
#include "lamgac.h"
int main (int argc, char *argv[]) {
      señal_c=CALCULAR;
      // .. Sentencias del bloque INITIALIZE
      LAMGAC_Init_fn ();
      // .. Recepción de datos iniciales
      // .. Acción 1: recepción de la matriz A
      for (i=0;i<P;i++) {
      // .. Acción 2: recepción de la columna de la matriz B
          do {
          // ... Acción 3: recepción de los límites del subconjunto de filas
              if (señal_c==CALCULAR) {
                  // ... Acción 4: calcular
                  // ... Acción 5: enviar resultados
                  // ... Acción 6: recepción de señal sobre repetición de iteración
              }
          } while (repetir iteración);
      }
      LAMGAC_Finalize_slave (my_rank);
      MPI_Finalize();
      return(0);
}
```

Figura 5.7 Esqueleto de los procesos en los CF en la implantación B

mismas que las realizadas por los procesos que se desarrollan en los CF, con la salvedad de que se recibe la iteración actual del bucle principal y en la acción 3 se tiene en cuenta la desvinculación del proceso. A continuación se describe esta acción:

- *Acción 3: recepción de los límites del subconjunto de filas.* En cada iteración, el proceso recibe el límite inferior y superior del rango de filas de la matriz *A* siempre y cuando la señal adjunta al mensaje indique que se tienen que realizar los cálculos. En caso contrario, pasará a la espera en la siguiente iteración. Sin embargo, si el proceso ha solicitado desvincularse del entorno, éste recibirá otra señal para que finalice su ejecución.

En la figura 5.8 se presenta un esqueleto para los procesos que se ejecutan en los CP.

```
#include "lamgac.h"
int main (int argc, char *argv[]) {
      señal_c=CALCULAR;
      // ... Sentencias del bloque INITIALIZE_EP
      LAMGAC_Init_pn (...);
      // .. Recepción de datos iniciales
      i=iter_actual;
      // .. Acción 1: recepción de la matriz A
      while((i<P) && (señal_c==DESVINCULAR)) {
      // .. Acción 2: recepción de la columna de la matriz B
          do {
          // ... Acción 3: recepción de los límites del subconjunto de filas
              if (señal_c==CALCULAR) {
                  // ... Acción 4: calcular
                  // ... Acción 5: enviar resultados
                  // ... Acción 6: recepción de señal sobre repetición de iteración
               }
          } while (repetir iteración);
          i++;
      }
      LAMGAC_Finalize_slave (my_rank);
      MPI_Finalize();
      return(0);
}
```

Figura 5.8 Esqueleto de los procesos en los CP en la implantación B

5.3.3 Implantación C

Para esta implantación se presentan las acciones principales que realiza cada proceso.

Esqueleto para el proceso maestro

Las acciones de este proceso son similares al correspondiente en las implantaciones anteriores. Sin embargo, para evitar confusiones, a continuación se describen todas las acciones:

- *Acción 1*: *replicación de una columna de la matriz B*. Al comienzo de cada

iteración el proceso maestro comunica una columna de la matriz B a todos los procesos vinculados al entorno de computación.

- *Acción 2*: *actualización del número de procesos*. Se invoca la función *LAMGAC_Update()* para actualizar el número de procesos en los CP. A continuación, y si un nuevo proceso se expande en un CP, hay que comunicarle los datos iniciales necesarios para que pueda realizar los cálculos. Entre otros, se le comunica la columna correspondiente de la matriz B. Si existe algún proceso que se quiere desvincular, se le envía un mensaje para que acabe su ejecución.

- *Acción 3*: *estimación del volumen de filas a procesar*. Después de conocer los cambios producidos en la VM_LAM, se invoca la función *LAMGAC_Balance()* para obtener el número de filas a distribuir a cada proceso y la secuencia de envío a éstos. Para establecer los valores de los parámetros *sizedata* y *sizeresults* hay que tener en cuenta que en esta implantación se comunican las filas a los procesos solamente si no han sido comunicadas previamente. Los valores de estos parámetros se utilizan en las tres siguientes situaciones: en la primera iteración del programa paralelo, o cuando se expande un nuevo proceso o cuando se recibe una notificación del tipo *lbnLoad o lbnUpLink*. En las dos primeras situaciones se comunican filas a los procesos, dado que no disponen de ellas, y en la tercera, se envían si no han sido comunicadas con anterioridad. Dado que esta última situación ocurre con menos frecuencia, en esta implantación se ha optado por fijar los parámetros *sizedata* y *sizeresults* a los valores correspondientes al envío de una fila y la recepción de un resultado, esto es, a *sizeof(float)*M* y *sizeof(float)*, respectivamente.

- *Acción 4*: *determinar la información a distribuir*. Después de conocer la cantidad de filas que se debe a enviar a cada proceso, éstas se deben enviar sólo si no están disponibles en el proceso esclavo. Para ello, el proceso maestro almacena y gestiona localmente el número de las filas que se le ha enviado a

cada proceso. Básicamente, la distribución se realiza de la siguiente forma: en la primera iteración se comunica a cada proceso un conjunto de filas según los resultados obtenidos en la acción anterior, y el proceso maestro registra el número de las filas comunicadas a cada proceso. En las sucesivas iteraciones y según el resultado de la llamada a la función *LAMGAC_Balance()*, cada proceso esclavo puede operar con las filas que tiene almacenadas o puede requerir operar con más filas. En el caso de calcular sólo con las filas que tiene almacenadas, el proceso maestro le comunica un vector en el que le indica el número de las filas con las que debe operar. En el caso de que necesite más filas, el proceso maestro le comunica las filas necesarias y el vector con el número de filas con las que debe operar. Por otro lado, el proceso maestro debe controlar que dos o más procesos no realicen cálculos con la misma fila de datos.

- *Acción 5*: *distribución de información.* Una vez conocida la información a enviar a cada proceso (filas y/o vector de filas) se procede a su envío. El tamaño de esta comunicación de datos es la suma del número de *bytes* del vector de enteros de dimensión igual a la cantidad de filas a procesar y los *bytes* correspondientes al número de filas a enviar cuando proceda el envío de éstas. Debido a la gran cantidad de datos a enviar, la distribución de filas se implementa con rutinas no bloqueantes para evitar el tiempo ocioso en los computadores con enlaces de comunicación rápidos mientras esperan que los datos sean recibidos por los computadores con enlaces con *throughput* bajo. Por otro lado, si en una iteración dada no se distribuyen filas a uno o varios procesos, a éstos se les envía una señal para que no realicen operaciones de cálculo y pasen a la espera en la siguiente iteración.

En la figura 5.9 se presenta un ejemplo: el número de filas de la matriz *A* es nueve y en la primera iteración el número de procesos en los CF y CP es NA=2 (P1 y P2) y NI=1 (P3), respectivamente. En esta iteración, la llamada a la función *LAMGAC_Balance()*, con el parámetro *distdata* igual a 9, devuelve

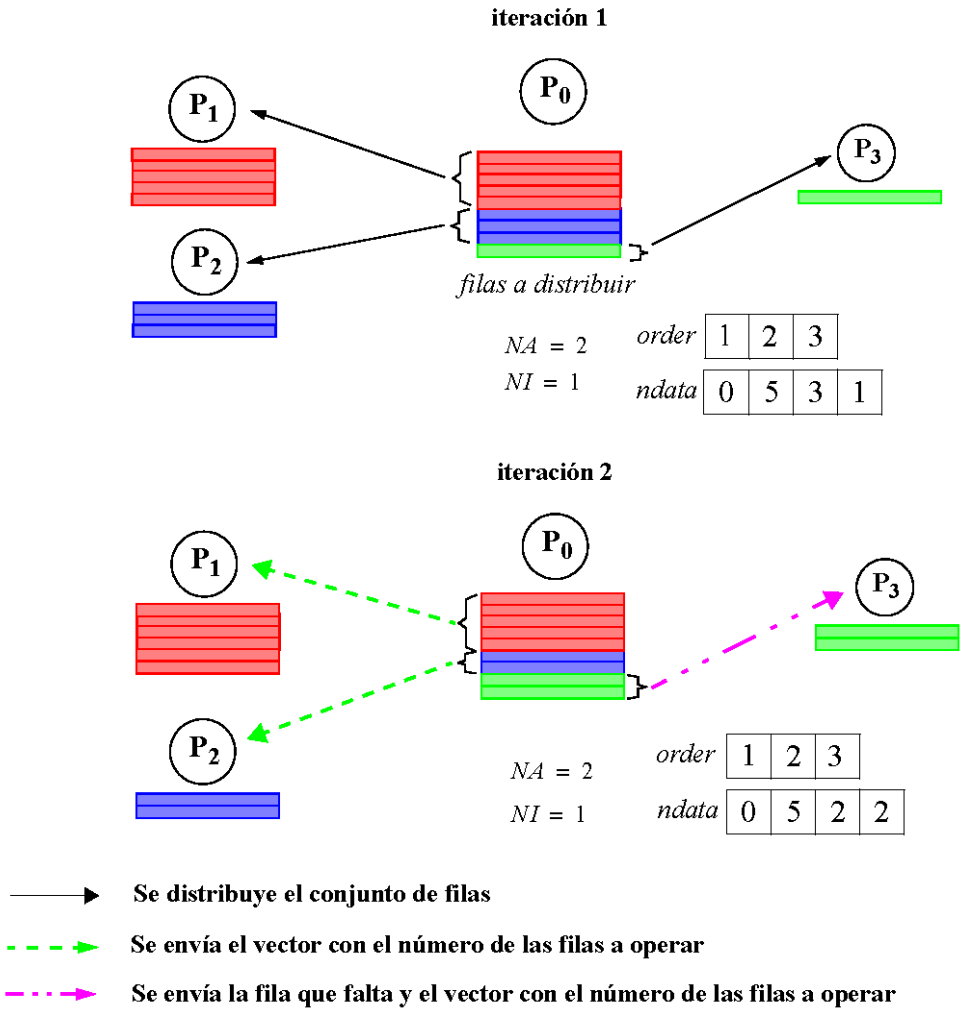


Figura 5.9 Ejemplo de distribución de filas en la implantación C

como resultado que a los procesos P1, P2 y P3 se les debe enviar cinco, tres y una fila de la matriz A, respectivamente (vector *ndata*). La secuencia de envío de datos se realiza en el orden siguiente: P1, P2 y P3 (vector *order*). En la siguiente iteración, la invocación a *LAMGAC_Balance()* devuelve un número de cinco filas a enviar al proceso P1, y dos a los procesos P2 y P3. Dado que tanto el proceso P1 como P2 tienen disponible en su memoria las filas con las que operar, no es necesario enviárselas otra vez, y por tanto, sólo se les envía un vector de índices indicándole el número de las filas con las que deben

operar. Sin embargo, el proceso P3 dispone de una fila y es necesario enviarle otra para continuar con los cálculos, por tanto, se le envía dicha fila y el vector donde se le indica el número de las filas con las que operar. Nótese, que a partir de esta iteración, tanto el proceso P2 como el P3 tienen en común los datos correspondientes a una misma fila, y por tanto, el proceso maestro debe considerarlo para evitar la repetición de cálculos.

- *Acción 6*: *recepción controlada de resultados*. Una vez realizada la distribución de filas, se implementa el mecanismo de recepción controlada explicado en la sección 2 del capítulo anterior para recibir los datos calculados por los procesos esclavos.

- *Acción 7*: *contabilización de filas no procesadas y envío de señal*. Al igual que en la implantación A, en el caso de existir filas no procesadas debido a CP no disponibles, los cálculos correspondientes a éstas se distribuyen entre el resto de procesos disponibles. En este caso, se vuelven a repetir las acciones desde la acción 2 con las filas no procesadas. Cuando todas las filas sean procesadas se pasa a la siguiente iteración.

En la figura 5.10 se presenta el esqueleto completo para el proceso maestro.

Esqueleto para los procesos en los CF

El programa que se ejecuta en estos computadores realiza, básicamente, las siguientes acciones:

- *Acción 1*: *recepción de la columna de la matriz B*. En cada iteración del programa paralelo el proceso recibe una columna de la matriz B.

- *Acción 2*: *recepción de información*. En la primera iteración, el proceso recibe un mensaje donde una señal le indica que van adjuntas un conjunto de filas de la matriz A para realizar los cálculos. En el resto de iteraciones, el proceso recibe el vector con el número de filas a operar, y si es necesario, también recibirá filas nuevas. En todas las iteraciones, el proceso maestro puede recibir

```
#include "lamgac.h"
int main (int argc, char *argv[])
{
      int my_id, NA, NI= 0, ND;
      float micalc[]={...};

      // .. Sentencias del bloque INITIALIZE
      LAMGAC_Init_an (argv);
      // .. Comunicación de datos iniciales
      for (i=0;i<P;i++) {
      //. . Acción 1: replicación de una columna de la matriz B
          do {
          //. . Acción 2: actualización del número de procesos
                LAMGAC_Update (..., &NI, &ND, ...);
                // Comunicar a los procesos nuevos los datos iniciales y la columna actual de B
                // Comunicar a los procesos que se van a desvincular que finalicen el algoritmo
          // . . Acción 3: estimación del volumen de filas a procesar
                LAMGAC_Balance (..., micalc, sizeof(int)*col_A, sizeof(int), filas_A, order,
ndata);
          //. . Acción 4: determinar la información a distribuir
          //. . Acción 5: distribución de información
          //. . Acción 6: recepción controlada de resultados
          //. . Acción 7: contabilización de las filas no procesadas y envío de señal
          } while (filas no procesadas);
      }
      LAMGAC_Finalize();
      MPI_Finalize();
      return(0);
}
```

Figura 5.10 Esqueleto de ejecución para el proceso maestro en la implantación C

una señal indicándole que no se le envía información, y por tanto, debe pasar a la siguiente iteración.

- *Acción 3*: *calcular*. Se realiza la multiplicación de las filas indicadas en el vector por la columna actual de la matriz *B*.
- *Acción 4*: *enviar resultados*. Una vez concluidos los cálculos, cada proceso envía los resultados obtenidos al proceso maestro.

- *Acción 5*: *recepción de señal sobre repetición de iteración.* El proceso recibe una señal del proceso maestro que le indica si hay que repetir los cálculos con la misma columna de la matriz B y una nueva distribución de filas o se pasa a la siguiente iteración. Las nuevas filas que se recibirán son parte de las filas que uno o varios CP no procesaron o no pudieron enviar los resultados obtenidos con ellas.

En la figura 5.11 se presenta un esqueleto para los procesos que se ejecutan en los CF.

Esqueleto para los procesos en los CP

Como ocurre en las implantaciones anteriores, las acciones realizadas por este tipo de procesos son las mismas que los procesos que se desarrollan en los CF, con la salvedad de que se recibe la iteración actual del bucle principal y en la acción 2 se tiene en cuenta si el proceso ha solicitado su desvinculación. A continuación se describe dicha acción:

- *Acción 2*: *recepción de información.* En la primera iteración de este proceso, se recibe un conjunto de filas inicial de la matriz A para realizar los cálculos. En el resto de iteraciones, el proceso recibe el vector con el número de filas a operar, y si es necesario, también recibirá filas nuevas. En todas las iteraciones se realizarán cálculos cuando la señal adjunta al mensaje indique que se tienen que realizar éstos. En caso contrario, pasará a la espera en la siguiente iteración. Sin embargo, si el proceso ha solicitado desvincularse del entorno, éste recibirá otra señal para que finalice su ejecución.

En la figura 5.12 se presenta un esqueleto para los procesos que se ejecutan en los CP.

5.3.4 Análisis de los resultados

Los programas de las diferentes implantaciones descritas en los apartados anteriores fueron llevados a término sólo en la configuración C2 de la tabla 5.3. Esto se debe a que

```
#include "lamgac.h"
int main (int argc, char *argv[]) {
    señal_c=CALCULAR;
    // .. Sentencias del bloque INITIALIZE
    LAMGAC_Init_fn ();
    // .. Recepción de datos iniciales
    for (i=0;i<P;i++) {
    // .. Acción 1: recepción de la columna de la matriz B
        do {
        // .. Acción 2: recepción de información
            if (señal_c==CALCULAR) {
              // .. Acción 3: calcular
              //.. Acción 4: enviar resultados
              //.. Acción 5: recepción de señal sobre repetición de iteración
             }
        } while (repetir iteración);
    }
    LAMGAC_Finalize_slave (my_rank);
    MPI_Finalize();
```

Figura 5.11 Esqueleto de los procesos en los CF en la implantación C

la configuración C1 utiliza cuatro CF con un rendimiento en las comunicaciones de datos muy superior al de los CP, y por tanto, el mecanismo de equilibrio de carga desestima desde la primera iteración la utilización de los CP para las implantaciones A y C. A modo de ejemplo, para la configuración C2, en la primera iteración de la implantación A, el proceso maestro comunica del orden de 520 y 420 filas a los procesos en los CP1 y CP4, respectivamente, y del orden de 30 filas a los procesos en los CP1 y CP3.

El tamaño de la matriz A utilizado es de 1000 filas por 5000 columnas de números reales, y el tamaño de la matriz B es de 5000 filas por 100 columnas. En cada iteración del algoritmo se calculan los resultados correspondientes al producto de la matriz A por una columna del matriz B, por tanto, el algoritmo paralelo realiza 100 iteraciones. Cada experimento se repitió veinte veces, obteniéndose una desviación típica baja en los tiempos de ejecución.

```
#include "lamgac.h"
int main (int argc, char *argv[]) {
      señal_c=CALCULAR
      // ... Sentencias del bloque INITIALIZE_EP
      LAMGAC_Init_pn (...);
      // .. Recepción de datos iniciales
      i=iter_actual;
      while((i<P) && (señal_c!=DESVINCULAR)) {
      // .. Acción 1: Recepción de la columna de la matriz B
          do {
          // ... Acción 2: recepción de información
              if (señal_c==CALCULAR) {
                  // ... Acción 3: calcular
                  //... Acción 4: enviar resultados
                  //... Acción 5: recepción de señal sobre repetición de iteración
               }
          } while (repetir iteración);
          i++;
      }
      LAMGAC_Finalize_slave (my_rank);
      MPI_Finalize();
      return(0);
}
```

Figura 5.12 Esqueleto de los procesos en los CP en la implantación C

En la figura 5.13 se muestran el tiempo medio de ejecución de la aplicación secuencial en el CM (SEQ_CM) y en el CF4 (SEQ_CF4), esto es, en la máquina más rápida y la más lenta de la configuración C2. También, se presenta el tiempo medio de la ejecución paralela utilizando una planificación equilibrada por bloques, esto es, se distribuye el mismo volumen de datos a todos los procesos (NLB_*x* en la figura, donde *x* representa la implantación del algoritmo) y de la ejecución paralela utilizando la técnica de equilibrio de carga (LB_*x*) para cada una de las diferentes implantaciones del producto de matrices. En la ejecución paralela, los procesos de los CP se expandieron en la primera iteración del algoritmo. Claramente se observa que la ejecución secuencial,

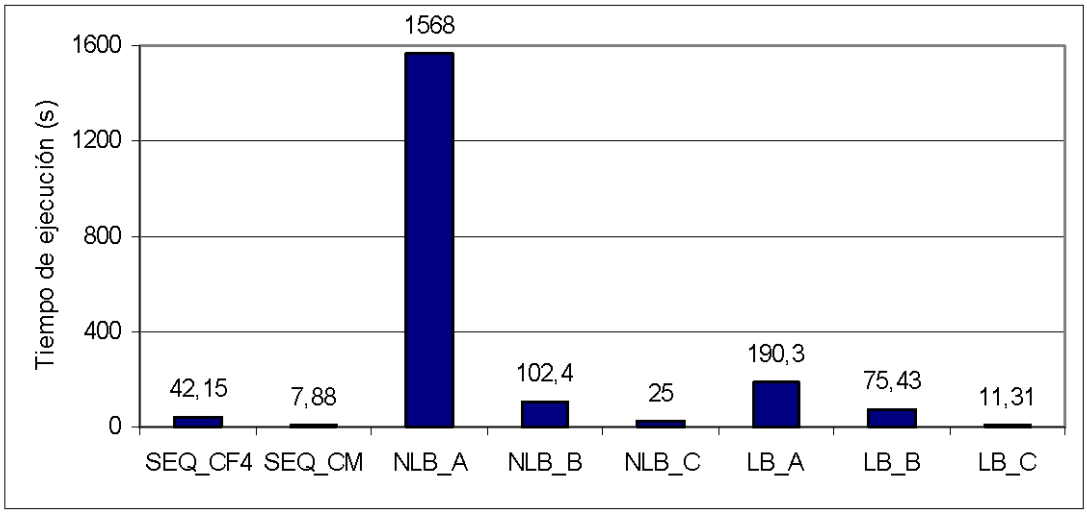


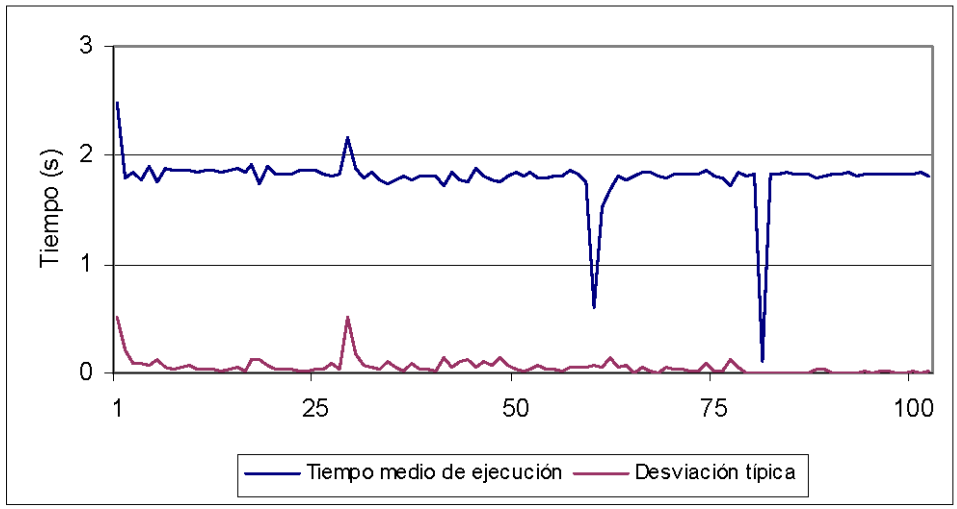
Figura 5.13 Eficiencia del protocolo de equilibrio de carga para las diferentes implantaciones

tanto en el recurso más rápido como en el más lento, es más rápida que la ejecución de las aplicaciones paralelas-distribuidas (a excepción del resultado obtenido para LB_C si lo comparamos con SEQ_CF4). Estos resultados se deben al tiempo excesivo que toman las comunicaciones con respecto al tiempo de cálculo.

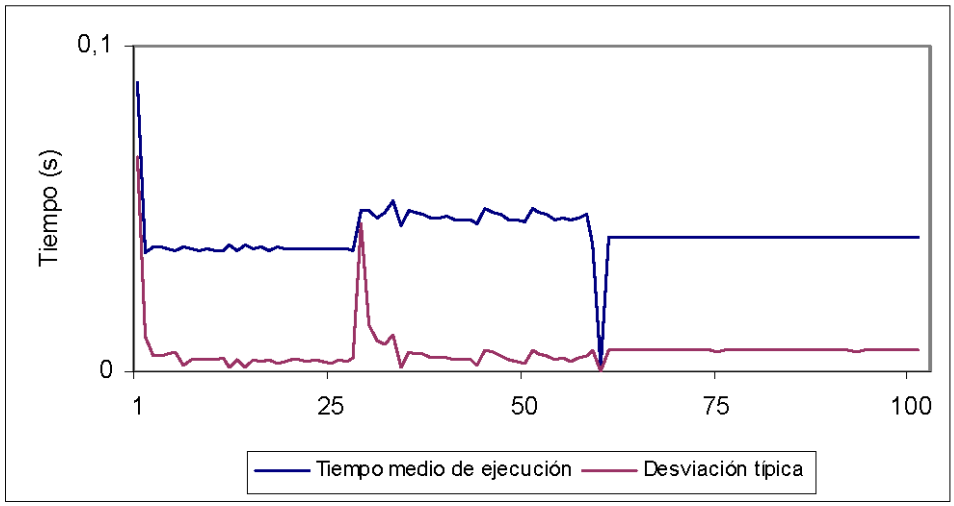
Con respecto a las soluciones paralelas, se observa claramente que el tiempo de ejecución de las implantaciones que utiliza el mecanismo de equilibrio de carga es inferior a las implantaciones no balanceadas, debido a que el proceso maestro espera siempre por la recepción de todos los resultados antes de pasar a la siguiente iteración. Además, se confirma que la implantación A es la menos eficiente, y la implantación C la que mejor resultados ofrece. Aunque en la implantación B, en cada iteración el proceso maestro distribuye sólo los límites del subconjunto de filas a procesar, el excesivo tiempo que toma esta implantación viene determinado por la comunicación de la matriz *A* a los procesos que se ejecutan en los CP. A modo de ejemplo, de los aproximadamente 76 segundos que tarda esta implantación, aproximadamente 69 segundos corresponde con la transmisión completa de la matriz A a los dos procesos de los CP (acción 1 del esqueleto del proceso maestro, figura 5.6).

En la figura 5.14 se muestra el tiempo medio de ejecución y la desviación típica de

a)



b)



c)

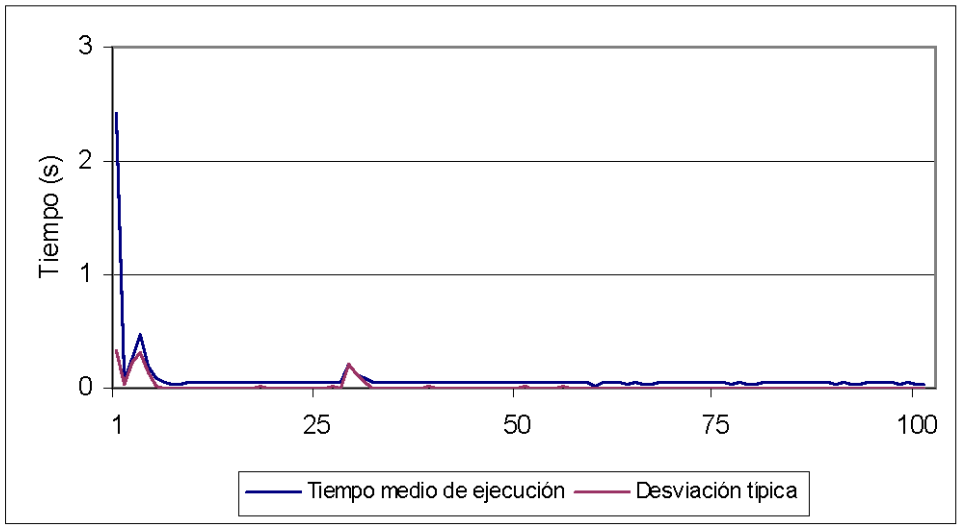


Figura 5.14 Tiempo de ejecución y desviación típica por iteración: a) implantación A, b) implantación B y c) implantación C

los procesos por iteración para las tres implantaciones utilizando el mecanismo de equilibrio de carga. Durante la ejecución, se producen vinculaciones de procesos y, por otro lado, los CP no están disponibles durante varias iteraciones: En la primera iteración se expande un proceso en CP1; en la iteración 30 se expande otro proceso en el CP3; en la iteración 60 y 81 y hasta el final de la aplicación, los CP1 y CP3 no están disponibles, respectivamente.

Como se puede apreciar, las desviaciones típicas obtenidas son próximas a cero, del orden de centésimas de segundo, con lo cual podemos afirmar que el equilibrio se alcanza. Por otro lado, se aprecia que cuando un proceso se expande en la iteración 30, la desviación típica aumenta debido al desequilibrio producido por la entrada del nuevo proceso. Sin embargo, el sistema se adapta rápidamente en las iteraciones sucesivas. Aunque en la implantación C, debido a que el proceso maestro envía filas en algunas iteraciones y en otras no, la desviación típica varía con mayor amplitud. Sobre todo esto se aprecia al comienzo de la aplicación, y en la iteraciones sucesivas a la 30 (cuando se expande un nuevo proceso). Por otro lado, en las iteraciones 60 y 81 el proceso maestro no recibe los resultados de un proceso del CP correspondiente, y por tanto, se realiza una iteración extra con las filas no procesadas. Por este motivo, el tiempo de ejecución en dichas iteraciones es bastante inferior al resto (menos cálculo que realizar), y la aplicación realiza 102 iteraciones en lugar de 100. En la figura 5.14 b), se aprecia que al incluir un nuevo proceso, en la iteración 30, el tiempo de ejecución medio aumenta aproximadamente en una céntesima de segundo hasta la iteración 60 que es a partir de la iteración cuando no está disponible el CP1. Esto es debido a que durante estas iteraciones se invocan más rutinas de envío y recepción de datos (*MPI_Isend* y *MPI_Irecv*) que en el resto, contribuyendo al incremento del tiempo de finalización de la iteración. Esta situación ocurre también en la implantación A y en la C, pero no se aprecia en las figuras porque el tiempo de ejecución es más elevado en la implantación A, y en la figura 5.14 c) la resolución no permite apreciarlo. Por otro lado, en la iteración 61 se realiza el cálculo de los resultados no obtenidos del CP no disponible. En el caso de la implantación B, y debido al número reducido de filas con las que calcular en dicha

iteración, el mecanismo de equilibrio de carga sólo considera los CF, y en las iteraciones sucesivas no vuelve a considerar el otro CP disponible.

Por último, en la figura 5.15 se muestra la sobrecarga que introduce la ejecución de la hebra *Snmp_Ping* en el tiempo de ejecución de la multiplicación de matrices. Para llevar a término esta evaluación, se forzó la ejecución de las hebras desde el comienzo de la aplicación paralela-distribuida hasta el final. Dado que en la configuración utilizada sólo se disponían de dos CP, para simular los experimentos con un número de hebras superior a dos, se replicó el número de éstas realizando las consultas sobre los CP de forma equitativa. Como se puede apreciar, el tiempo de ejecución de la aplicación paralela se incrementa ligeramente con el número de hebras. En el peor caso (implantación C), la sobrecarga supone un 0,87% del tiempo de ejecución. Por tanto, se demuestra que la sobrecarga que introduce la hebra de control de CP es despreciable. Además, en un escenario real es poco probable que las hebras permanezcan en ejecución desde el comienzo de la aplicación hasta el final, con lo que la sobrecarga real será inferior a la mostrada en la figura.

5.4 Método de Relajación de Jacobi

El método de relajación de Jacobi se emplea en la resolución de la ecuación diferencial de Laplace la cual se define de la siguiente forma:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

Para la solución de este método supongamos una lámina de metal conductora de dos dimensiones con una tensión aplicada en su contorno. Para calcular la tensión en cada punto interior de la lámina, se debe resolver la ecuación de Laplace para todos los puntos interiores, donde *V(i,j)* representa la tensión en cada punto (*i,j*) de la lámina de metal. Esta ecuación se puede transformar en la siguiente ecuación en diferencias aplicada a una

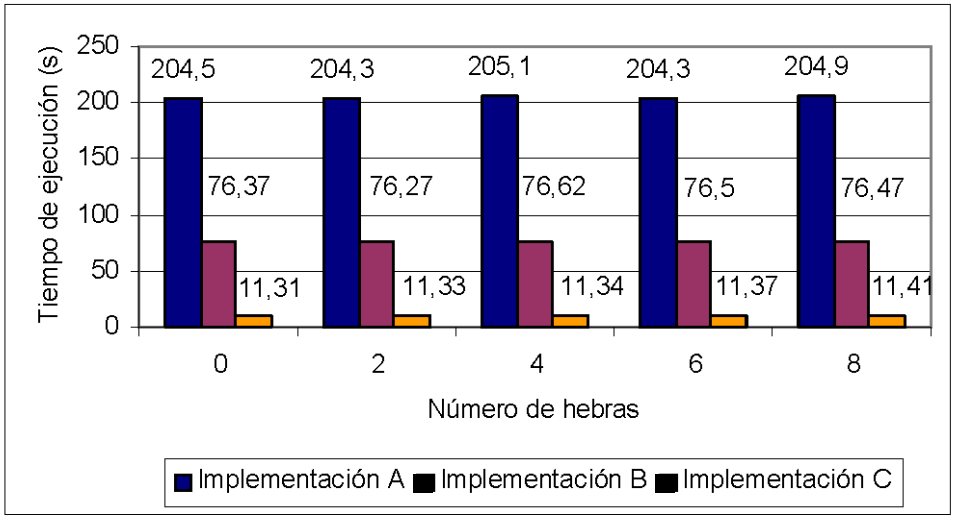


Figura 5.15 Sobrecarga de la hebra de control sobre los CP

malla bidimensional de puntos en la superficie de la lámina de metal.

$$V(i,j) = \frac{V(i-1,j) + V(i+1,j) + V(i,j-1) + V(i,j+1)}{4}$$

Como se observa, la tensión en cada punto se calcula como el promedio de las tensiones existentes en los cuatros puntos vecinos. Para ello, primero se deben seleccionar valores iniciales para la tensión en cada punto de la malla. La tensión es constante en los puntos del contorno, por tanto, estas tensiones son conocidas. Respecto a los puntos interiores, se suponen inicialmente igual a cero. En cada iteración del algoritmo se recalcula la tensión para cada punto interior como el promedio de las tensiones de los cuatros puntos vecinos. En cada iteración, la tensión de los puntos interiores varía mientras que la tensión en los puntos del contorno se mantiene constante.

5.4.1 Implantación

El algoritmo secuencial de relajación de Jacobi para resolver la ecuación de Laplace emplea una matriz bidimensional de números reales que representan la tensión en los puntos de la lámina de metal (puntos interiores y del contorno). Después de iniciar la matriz de puntos (a cero los puntos interiores y a un valor constante los puntos del

contorno), se recalcula el valor de cada punto de la matriz (excluyendo los puntos del contorno), computando el promedio de la tensión de sus vecinos. Este proceso se repite hasta que la solución converge hacia la precisión deseada o hasta un número de iteraciones dado.

En la figura 5.16 se muestra el algoritmo secuencial del método de relajación de Jacobi para resolver la ecuación de Laplace. Los datos de entrada al programa son: la tensión del contorno (*cte*), el número de iteraciones (*m*), el número de iteraciones máximo (*itermax*), el número de puntos en una fila o columna interior (*n*) y la tolerancia (*tolerance*). La matriz *A* contiene los valores iniciales para todos los puntos de la lámina de metal. En cada iteración del bucle con variable índice *k*, se calcula un nuevo valor para cada punto utilizando la ecuación en diferencias. El valor calculado se almacena en el elemento (*i,j*) de una matriz auxiliar (*B*) para evitar interferencias con el resto de los cálculos. Además, se calcula la diferencia en valor absoluto (*change*) del valor obtenido (*B[i][j]*) con el previamente calculado en la iteración anterior (*A[i][j]*). De esta forma, se obtiene la máxima diferencia (*maxchange*) entre los valores calculados en la iteración k con los obtenidos en la iteración k-1. Este cálculo es necesario para determinar si el algoritmo converge hacia una solución determinada. Al finalizar los cálculos de todos los puntos interiores de la matriz *A*, se copia el contenido de la matriz auxiliar a la matriz A antes de iniciar la siguiente iteración. El algoritmo iterativo finaliza cuando se han realizado un conjunto de iteraciones ($m \times \text{iter\_max}$) o bien cuando el máximo cambio (después de haber realizado m iteraciones) producido entre todos los valores de los puntos interiores sea inferior a un valor dado de tolerancia (*tolerance*).

Respecto a la implantación paralela, este algoritmo tiene bastante paralelismo explotable debido a que el cálculo en cada punto se realiza de forma independiente, y por tanto, puede calcularse en paralelo con los otros puntos de la matriz. Hay que tener en cuenta que las iteraciones del bucle de la variable *k* no son independientes, y por tanto, cada iteración depende de los valores calculados en la iteración previa. En la implantación de este algoritmo, se distribuye un conjunto de los puntos interiores de la matriz (filas de la matriz) a cada proceso esclavo. La dimensión de cada conjunto viene

```
do {
      maxchange= 0.0;
      for (k= 0; k< m; k++) {
          for (i= 1; i< n+1; i++) {
                    for (j= 1; j< n+1; j++){
                              B[i][j]= (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4 ;
                              change= fabs(fabs(B[i][j]) - fabs(A[i][j]));
                               if (change> maxchange) maxchange= change;
                    }
          }
          // ... Actualización de la matriz de puntos después de cada iteración
          for (i= 1; i< n+1; i++) for (j= 1; j< n+1; j++) A[i][j]= B[i][j];
      }
      iter++;
} while( (iter<= itermax) && (maxchange> tolerance));
if ( (iter> itermax) && (maxchange> tolerance) ) // ... No ha habido convergencia
```

Figura 5.16 Algoritmo secuencial del método de relajación de Jacobi

determinado por la aplicación de la técnica de equilibrio de carga desarrollada en este trabajo de investigación. Sin embargo, como para calcular la tensión de los puntos situados en los límites de cada conjunto es necesario conocer la tensión de los puntos contiguos, también hay que comunicar a cada proceso aquellas filas inmediatamente anterior y posterior al conjunto, es decir aquellas filas que hacen frontera con el conjunto. A éstas filas se les denomina *frontera.* La frontera que necesita un proceso para calcular la tensión de los puntos de las filas asignadas es calculada por otro proceso en la iteración previa. Por ello, y dado que la frontera que necesita un proceso se corresponde con las filas de puntos para los que otro proceso calcula su tensión, después de realizar los cálculos en cada iteración es necesario comunicar las fronteras actualizadas a cada proceso.

En la figura 5.17 se muestra un ejemplo de la frontera para tres procesos. Se ha supuesto que el número de filas interiores de la matriz es igual a 10 (desde el índice 1 hasta el 10), y que se distribuyen entre tres procesos según el resultado de la llamada a la función *LAMGAC_Balance()* (vector *ndata*). Por otro lado, las filas 0 y 11 de la matriz se

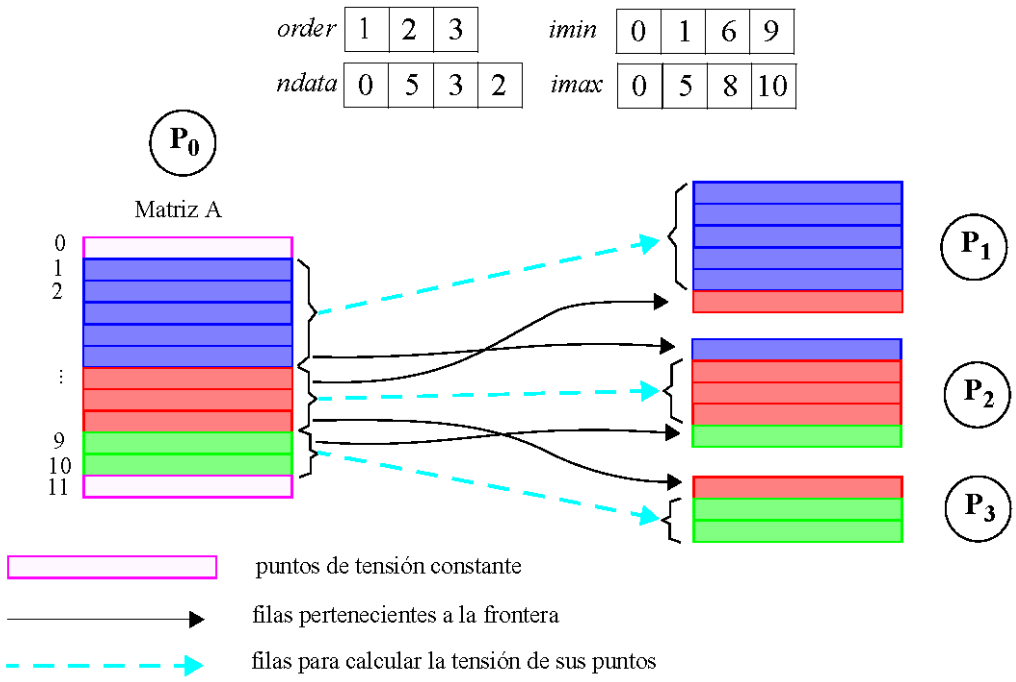


Figura 5.17 Ejemplo de frontera para los procesos del algoritmo paralelo

corresponde con la tensión de contorno constante de la lámina de metal. El proceso maestro almacena toda la matriz *A*, así como los límites del subconjunto de filas (vectores *imin* e *imax* en la figura) que modifican los procesos del programa paralelo en cada iteración del bucle. A cada proceso se le envía un conjunto de filas, y la frontera que le corresponde. Por ejemplo, al proceso P1, se le envía el conjunto correspondiente a las cinco primeras filas. Para modificar los elementos que se almacenan en estas filas, el proceso necesita conocer los valores que se corresponden con su frontera: elementos almacenados en la fila 0 y fila 6 de la matriz *A*. Dado que el proceso maestro asigna un valor constante a los elementos que se encuentran en la fila 0 y en la fila 11, y éstos no cambian durante su ejecución, diremos entonces que la frontera para el proceso P1 se corresponde con los elementos de la fila 6. De igual manera, el proceso con mayor identificador, P3, tiene como frontera una única fila: la fila 8 de la matriz *A*. Cualquier otro proceso tiene como frontera dos filas de la matriz *A*. En la figura se observa que para P2 serían los elementos de las filas 5 y 9. Nótese que la frontera de P1 es modificada, en

cada iteración del bucle de variable k, por el P2. Una parte de la frontera de P2 la modifica P1, y la otra parte de la frontera la modifica el proceso P3. Por último, la frontera de P3 es modificada por P2.

Por tanto y debido a que un proceso necesita los datos que se almacenan en su frontera para calcular los puntos interiores correspondientes a las filas que modifica, es necesario que los procesos reciban los valores de su frontera en cada iteración. Para ello, cada proceso comunica las filas almacenadas en su memoria local que necesitan ser conocidas por otros procesos (las que corresponderían a la frontera de estos procesos), y a continuación reciben su frontera actualizada con los valores calculados por otros procesos.

A continuación se describen las acciones que realizan los procesos dependiendo del computador donde se desarrollen.

Esqueleto para el proceso maestro

Básicamente, las acciones implementadas por el proceso maestro son las siguientes:

- *Acción 1*: *lectura de los datos de entrada e inicialización de estructuras.* El proceso recibe los datos iniciales: la tensión del contorno, el número de iteraciones del bucle de la variable *k*, el número máximo de iteraciones, el número de puntos en una fila o columna interior (suponiendo una lámina de metal cuadrada) y la tolerancia. A continuación, se inicia la matriz *A*: a cero los puntos interiores de la matriz, y a la tensión de contorno el contorno de la matriz.

- *Acción 2: actualización del número de procesos.* Dado que los procesos esclavos necesitan conocer el número total de procesos en ejecución y su identificador para saber cuales son las fronteras que tienen que enviar y recibir, en este algoritmo es necesario utilizar la función *LAMGAC_Awareness_update()* [Mac01] en lugar de la función *LAMGAC_Update()*. La frontera de cada proceso está compuesta por dos filas, excepto para el primer y último proceso. Por tanto, los procesos tienen que

saber su rango para determinar qué frontera envían y reciben. *LAMGAC_Awareness_update()* realiza las mismas acciones que la función *LAMGAC_Update()* y además, todos los procesos que la invocan conocen el número total de procesos en ejecución, tanto en los CF como en los CP. Por tanto, esta función tiene que ser invocada por todos los procesos. Además, en el caso de existir procesos recién vinculados, a éstos se les envía el número de iteración en curso (iter).

- *Acción 3: estimación del volumen de filas a distribuir.* Después de conocer los cambios producidos en la VM_LAM, se invoca la función *LAMGAC_Balance()* para obtener el número de filas de la matriz a distribuir a cada proceso y la secuencia de envío a éstos. El valor de los parámetros *sizedata* y *sizeresults* de esta función coincide con el tamaño de una fila de la matriz (*sizeof(float)*puntos de la fila*), dado que se envía y reciben todas las filas. Una vez conocido el subconjunto de elementos a enviar a cada proceso, se actualizan los vectores límite inferior (*imin*) y superior (*imax*).

- *Acción 4*: *distribución de las filas de la matriz A.* Se envía a cada proceso el subconjunto de filas de *A* que le corresponda y también la frontera. De forma general, al proceso *i* le corresponden las filas *imin[i]-1, imin[i], ..., imax[i], imax[i]+1* almacenadas en *A*, donde *imin[i]-1* e *imax[i]+1* es la frontera. Al proceso con identificador 1 le corresponde sólo *imax[1]+1*, y al proceso con mayor identificador (NA+NI) le corresponde sólo *imin[NA+NI]-1*, como fronteras.

- *Acción 5*: *recepción y envío de fronteras.* Dentro del bucle de la variable *k*, el proceso maestro recibe de cada proceso las filas que pertenecen a la frontera de otros procesos, y las va almacenando en las filas correspondientes de la matriz *A*. De forma general, del proceso *i* recibe las filas *imin[i]* e *imax[i]*. Del proceso con identificador 1 se recibe *imax[1]* y del mayor identificador se recibe *imin[NA+NI]*. Finalizada esta recepción, el proceso maestro envía a cada proceso su frontera actualizada con los valores enviados previamente por

todos los procesos.

- *Acción 6: recepción controlada y actualización de la matriz.* Dado que de una iteración a otra se puede asignar un subconjunto de filas diferentes a un mismo proceso, es necesario que cada proceso envíe el conjunto de filas que ha procesado para que el proceso maestro reconstruya la matriz, y vuelva a ser enviada en la siguiente iteración siempre y cuando no se alcance la convergencia deseada y por tanto, no finalice la aplicación. Para ello, se implementa el mecanismo de recepción controlada explicado en la sección 2 del capítulo anterior.
- *Acción 7*: *contabilización de filas no procesadas y cálculo, si procede.* En el caso de existir filas no procesadas debido a CP no disponibles, las tensiones de los puntos correspondientes a estas filas serán calculadas localmente por el proceso maestro.
- *Acción 8*: *comprobación de la convergencia del algoritmo.* Se determina la convergencia del algoritmo comprobando si se ha llegado a la tolerancia deseada o se ha superado el valor máximo de iteraciones permitida. Para el primer caso necesita conocer los cambios calculados por cada proceso después de que éstos hayan realizado *m* iteraciones. Para ello recolecta el máximo cambio calculado por cada proceso y determina el mayor de los valores recibidos. A continuación radía este valor a todos los procesos en ejecución.

En la figura 5.18 se muestra el esqueleto del proceso maestro.

Esqueleto para los procesos en los CF

Básicamente, las acciones que realizan los procesos que se desarrollan en los CF son las siguientes:

- *Acción 1*: *lectura de los datos de entrada.* Los datos iniciales se pasan por parámetros.
- *Acción 2*: *actualización del número de procesos.* Se invoca a la función

```
// .. Sentencias del bloque INITIALIZE
LAMGAC_Init_an (argv);
// ... Acción 1: lectura de los datos de entrada e inicialización de estructuras
do {
    // ... Acción 2: actualización del número de procesos
    LAMGAC_Awareness_update(&my_id, &NA, &NI,..., &ND, ...);
    // ... Acción 3: estimación del volumen de filas a distribuir
    LAMGAC_Balance (..., micalc, sizeof(float)*col_A, sizeof(float)*col_A, filas_A, order, ndata);
    // ... Acción 4: distribución de las filas de la matriz A
    change= 0.0;
    for (k= 0; k< m; k++) // ... Acción 5: Recepción y envío de fronteras
    iter++;
    // ... Acción 6: recepción controlada y actualización de la matriz
    // ... Acción 7: contabilización de filas no procesadas y cálculo, si procede
    // ... Acción 8: comprobación de la convergencia del algoritmo
} while( (iter<= itermax) && (maxchange> tolerance) );
LAMGAC_Finalize();
MPI_Finalize();
```

Figura 5.18 Esqueleto del proceso maestro para el método de relajación de Jacobi

LAMGAC_Awareness_update() y se conoce el número de procesos que se ejecutan en los CF y en los CP, NA y NI, respectivamente, y el identificador del proceso que le invoca.

- *Acción 3: recepción de filas de la matriz*. Cada proceso recibe el subconjunto de filas de la matriz *A* enviadas por el proceso maestro.
- *Acción 4: calcular*. En cada iteración del bucle de la variable k se calcula el valor promedio de cada punto de las filas interiores mediante la ecuación en diferencias y se actualiza la matriz de puntos.
- *Acción 5: envío y recepción de fronteras*. Cada proceso envía al proceso maestro la frontera que necesita conocer otros procesos para poder realizar cálculos en la siguiente iteración de la variable *k*. Además, recibe del proceso maestro su frontera actualizada.
- *Acción 6: enviar resultados*. El proceso envía al proceso maestro las filas que

ha modificado en la etapa de cálculo. Este paso es necesario debido a que en la próxima iteración, el mecanismo de equilibrio de carga puede asignar a cada proceso un conjunto diferente de filas, y por tanto, los datos deben estar actualizados en el proceso maestro.

- *Acción 7: comprobación de la convergencia del algoritmo.* Para determinar conjuntamente la convergencia del algoritmo, cada proceso envía al resto de procesos el máximo cambio producido entre todos los puntos calculados.

En la figura 5.19 se presenta el algoritmo para los procesos que se ejecutan en los CF.

Esqueleto para los procesos en los CP

Los procesos que se desarrollan en los CP realizan las mismas acciones que aquellos que se ejecutan en los CF, con las siguientes salvedades: se recibe el número de la iteración actual cuando se vinculan, y la acción 2 no se realiza en la primera iteración. Esto se debe a que la invocación de la función *LAMGAC_Awareness_update()* provoca la sincronización entre todos los procesos. Sin embargo, si un proceso se expande en un nuevo computador, la sincronización con este nuevo proceso se realiza en la función *LAMGAC_Init_pn()*, y por tanto, no se puede llamar a la *LAMGAC_Awareness_uptate()* después de invocar a *LAMGAC_Init_pn()*.

En la figura 5.20 se presenta el algoritmo paralelo para los procesos que se ejecutan en los CP.

5.4.2 Análisis de los resultados

El método de relajación de Jacobi se desarrolló en una agrupación de computadores como la mostrada en la configuración C2 de la tabla 5.3. Al igual que ocurre para la aplicación de multiplicación de matrices, si la aplicación paralela se desarrolla en la agrupación C1, el mecanismo desestima la utilización de los CP por existir CF con mayor rendimiento en las comunicaciones.

Los datos de entrada utilizados son : *n=1000, m=5, iter_max=10, y tolerance =0.1.*

```
// .. Sentencias del bloque INITIALIZE
LAMGAC_Init_fn ();
// ... Acción 1: lectura de los datos de entrada
do {
    // ... Acción 2: actualización del número de procesos
    LAMGAC_Awareness_update(&my_id, &NA, &NI, ..., &ND, ...);
    // ... Acción 3: recepción de filas de la matriz
    maxchange= 0.0;
    for (k= 0; k< m; k++) {
        // ... Acción 4: calcular
        for (i= 1; i<num_filas-1; i++) {
            for (j= 1; j<n+1; j++){
                B[i][j]= (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4 ;
                change= fabs(fabs(B[i][j]) - fabs(A[i][j]));
                if (change> maxchange) maxchange= change;
            }
        }
        // ... Actualización de la matriz de puntos después de cada iteración
        for (i= 1; i< num_filas-1; i++) for (j= 1; j< n+1; j++)  A[i][j]= B[i][j];
        // ... Acción 5: envío y recepción de fronteras
    }
    iter++;
    // ... Acción 6: enviar resultados
    // ... Acción 7: comprobación de la convergencia del algoritmo
} while( (iter<= itermax) && (maxchange> tolerance) );
LAMGAC_Finalize_slave (my_rank);
MPI_Finalize();
```

Figura 5.19 Esqueleto de los procesos en los CP para el método de relajación de Jacobi

En la figura 5.21 se muestra: el tiempo medio de ejecución para el algoritmo secuencial (SEQ_CM y SEQ_CF4), el tiempo medio del programa paralelo utilizando una planificación equilibrada por bloques (NLB), y utilizando la técnica de equilibrio de carga (LB). Los procesos en los CP se expandieron en la primera iteración del algoritmo. Claramente se observa que la ejecución secuencial, tanto en el recurso más rápido como en el más lento, es más rápida que la ejecución de las aplicaciones paralelas-distribuidas. Estos resultados se deben al tiempo excesivo que toman las comunicaciones al enviar la

```
// .. Sentencias del bloque INITIALIZE_EP
LAMGAC_Init_pn (1,&my_id,&NA,&NI,...);
// ... Acción 1: lectura de los datos de entrada
//Recibir número de la iteración actual (iter)
do {
    if (!primera_vez) {
        // ... Acción 2: actualización del número de procesos
        LAMGAC_Awareness_update(&my_id, &NA, &NI, ..., &ND, ...);
    }
    // ... Acción 3: recepción de filas de la matriz
    maxchange= 0.0;
    for (k= 0; k< m; k++) {
        // ... Acción 4: calcular
        for (i= 1; i<num_filas-1; i++) {
            for (j= 1; j<n+1; j++){
                B[i][j]= (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4 ;
                change= fabs(fabs(B[i][j]) - fabs(A[i][j]));
                if (change> maxchange) maxchange= change;
            }
        }
        // ... Actualización de la matriz de puntos después de cada iteración
        for (i= 1; i< num_filas-1; i++) for (j= 1; j< n+1; j++)  A[i][j]= B[i][j];
        // ... Acción 5: envío y recepción de fronteras
    }
    iter++;
    // ... Acción 6: enviar resultados
    // ... Acción 7: comprobación de la convergencia del algoritmo
} while( (iter<= itermax) && (maxchange> tolerance) );
LAMGAC_Finalize_slave (my_rank);
MPI_Finalize();
```

Figura 5.20 Esqueleto de los procesos en los CP para el método de relajación de Jacobi

matriz de puntos en cada iteración, y a la sincronización existente dentro de cada iteración al comunicarse las fronteras. Con respecto a las soluciones paralelas, se observa claramente que el tiempo de ejecución de la implantación que utiliza el mecanismo de equilibrio de carga es inferior a la implantación no balanceada.

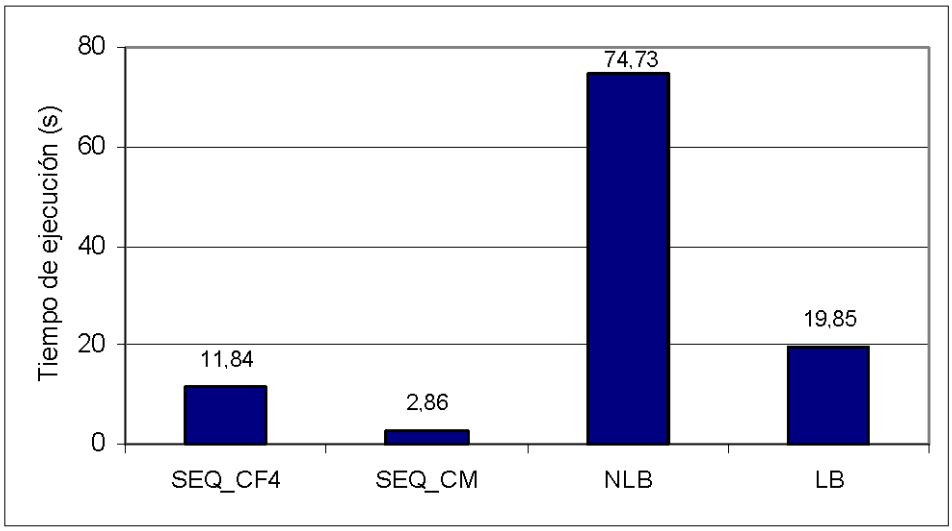


Figura 5.21 Eficiencia del protocolo de equilibrio de carga

Por otro lado, en la figura 5.22 se muestra la desviación típica del tiempo medio de ejecución de los procesos en cada iteración para la solución no balanceada (planificación equilibrada por bloques) y la que utiliza el mecanismo de equilibrio de carga. Claramente se observa, que a medida que avanza la aplicación, la desviación típica en la solución paralela balanceada va decreciendo, por tanto, los tiempos de ejecución de cada proceso van aproximándose entre sí. Por el contrario, en la solución no balanceada la desviación típica es elevada debido a que los tiempos de ejecución no están equilibrados.

Por último, en la figura 5.23 se muestra la sobrecarga que introduce la ejecución de la hebra *Snmp_Ping* en el tiempo de ejecución de la aplicación. Para llevar a término esta evaluación, se forzó la ejecución de las hebras desde el comienzo del porgrama paralelo hasta el final. Dado que en la configuración utilizada sólo se disponían de dos CP, para simular los experimentos con un número de hebras superior a dos, se replicó el número de éstas realizando las consultas sobre los CP de forma equitativa. Como se aprecia en la figura, el tiempo de ejecución de la aplicación se incrementa ligeramente con la ejecución de las hebras de control. En el peor caso, se incrementa aproximadamente 0,3 segundos, lo que supone un incremento alrededor del 1,5%. En cualquier caso, este

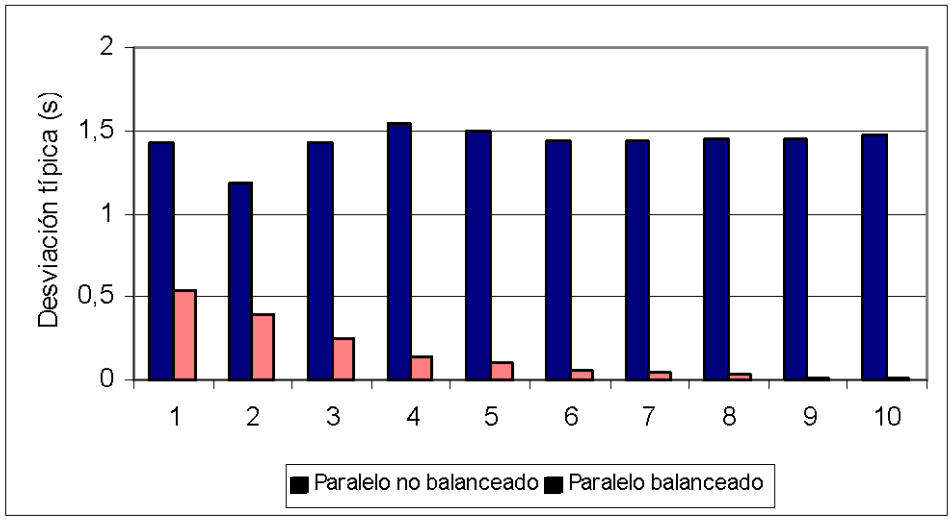


Figura 5.22 Desviación típica por iteración

incremento es despreciable frente a la ventaja que se obtiene controlando los CP, y teniendo en cuenta que en una situación real es poco probable que las hebras permanezcan en ejecución durante toda la ejecución de la aplicación.

5.5 Herramienta de Codiseño Hw/Sw

En el Grupo de Investigación de Arquitectura y Concurrencia del Departamento de Ingeniería Telemática de la Universidad de Las Palmas de Gran Canaria se ha desarrollado un entorno de codiseño denominado GACSYS (*GAC's Codesign System*) [CSC98][SaC99][SCS00]. En concreto, se han desarrollado las siguientes tareas:

- Diseño de un lenguaje de especificación denominado VSS (*VHDL-based System Specification*). VSS permite una codificación sencilla y soporta primitivas con un nivel de abstracción adecuado para especificar sistemas. Esto contribuye a disminuir el tiempo de diseño, los errores y el coste del mismo, así como las dificultades de mantenimiento y rediseño.
- Desarrollo de un compilador que genera código VHDL (*Very high speed integrated circuit Hardware Description Language*) a partir de VSS.

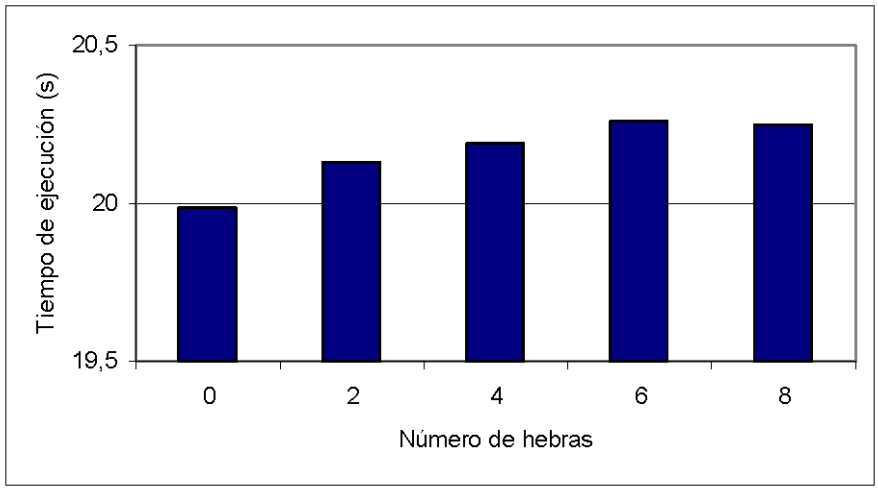


Figura 5.23 Sobrecarga de la hebra de control sobre los CP

- Desarrollo de una herramienta que permite obtener estimaciones de diversos parámetros relativos a la implantación de un sistema, tales como tiempo, área y potencia.
- Desarrollo de una herramienta automática de particionado que permite una mejor exploración del espacio de diseño, así como obtener un particionado mejorado respecto a un particionado manual.
- Diseño de una representación intermedia, basada en un grafo de flujo de datos denominado ASCIS, que soporta especificaciones VHDL. Esta representación permite el desarrollo posterior de herramientas de síntesis de hardware y/o software.
- Desarrollo de un compilador que genera la representación intermedia a partir de código VHDL.

En la figura 5.24 se muestra un diagrama de la herramienta desarrollada.

La tercera tarea genera las estimaciones del tiempo de ejecución, área y potencia consumidas por cada proceso VHDL especificado en la descripción del sistema. Estas estimaciones se generan para cada una de las posibles configuraciones *hardware* y

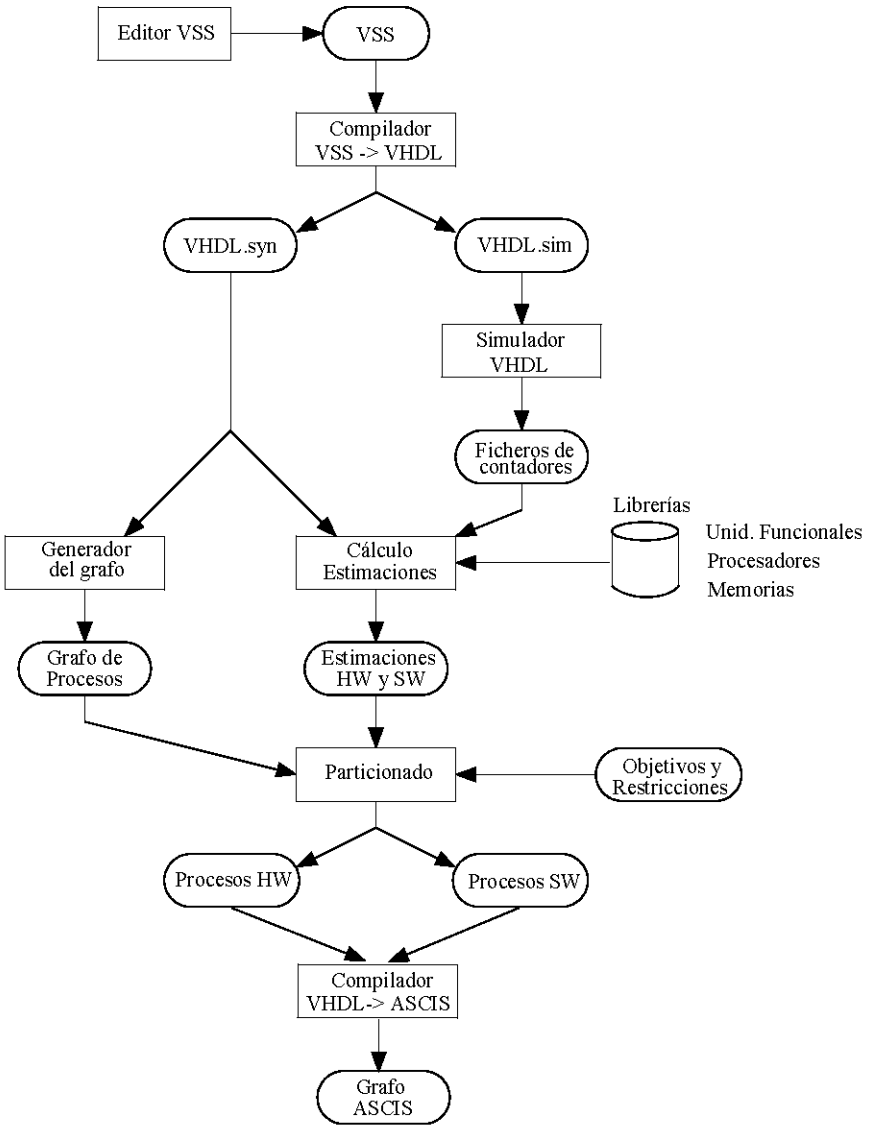


Figura 5.24 Herramienta de Codiseño GACSYS

software de los procesos. Como se aprecia en la figura 5.24, para generar las estimaciones se requiere: a) la descripción algorítmica del diseño, b) las características de las unidades funcionales, memorias y procesadores que se quieren utilizar, y c) un fichero de contadores que indica el número de veces que se ha ejecutado cada bloque básico de la descripción del sistema durante una simulación realizada previamente. Si el número de unidades funcionales disponibles es elevado, el número de configuraciones posibles para implementar un proceso también lo es, y

por tanto, la fase de evaluación de las estimaciones puede necesitar un tiempo de computación excesiva. En este sentido, para reducir su tiempo de ejecución es conveniente paralelizar su ejecución. A continuación se explica el proceso de evaluación de las estimaciones y su implantación paralela.

5.5.1 Implantación

Básicamente, la tarea de generación de estimaciones realiza las siguientes acciones:

- *Acción 1*: *lectura de datos de entrada.* Al comienzo del programa se realiza la lectura de los datos necesarios para realizar las estimaciones: fichero de especificación del sistema en VHDL, fichero de contadores y ficheros con las características de las unidades funcionales, memorias y procesadores a utilizar.
- *Acción 2*: *generación de representación interna y obtención de dependencias.* A partir de la especificación del sistema se genera una representación interna del mismo donde se almacena información necesaria de la especificación VHDL y las dependencias de datos existentes entre las sentencias de cada bloque básico de la especificación.
- *Acción 3*: *generación de combinaciones de recursos.* Esta acción consta de tres pasos y se ejecuta una vez para cada proceso VHDL. En primer lugar, y partiendo de las dependencias de datos se calcula el número máximo de operadores de cada tipo (sumas, restas, multiplicaciones, ...) que se pueden ejecutar en paralelo en cada bloque básico. A continuación, se generan todas las configuraciones posibles de recursos teniendo en cuenta el número máximo de operaciones de cada tipo que se pueden ejecutar en paralelo. Por último, para cada configuración obtenida se generan todas las combinaciones posibles de unidades funcionales según las unidades especificadas en el fichero de unidades funcionales. Este último paso se realiza de forma recursiva.
- *Acción 4*: *cálculo de las estimaciones.* Para cada proceso VHDL y para cada combinación de unidades funcionales se estima el área ocupada, el tiempo de

ejecución y la potencia consumida. El cálculo del área consiste en sumar el área de cada unidad funcional utilizada. Sin embargo, el cálculo de los otros dos parámetros se basa en la aplicación de un algoritmo de planificación de operaciones a cada bloque básico de la especificación del proceso. En concreto, se aplica el algoritmo de planificación basado en listas [GDW92]. Como resultado del algoritmo se obtiene el número de ciclos que necesita un bloque básico para su ejecución, y a partir de aquí se estima el tiempo y la potencia consumida. Para cada bloque básico, el algoritmo de planificación se ejecuta tantas veces como combinaciones de unidades funcionales se hayan generado. Nótese el elevado número de veces que se ejecuta el algoritmo durante el proceso de evaluación de las estimaciones. Por este motivo esta acción debe realizarse en paralelo.

- *Acción 5*: *generación de fichero de salida*. Por último se genera un fichero con las estimaciones realizadas para cada proceso y para cada combinación de recursos. Este fichero se utiliza como entrada de datos en la fase de particionado.

En la figura 5.25 se muestra el código secuencial utilizando las acciones mencionadas.

La fase de estimaciones tiene bastante paralelismo explotable debido a que el cálculo de estimaciones para cada combinación de recursos se realiza de forma independiente, y por tanto, puede calcularse en paralelo con otras configuraciones. Dado que la acción 3 tiene una etapa que se realiza de forma recursiva, y no es fácilmente paralelizable, sólo los cálculos de la acción 4 son realizados en paralelo. Para ello, y dado que cada proceso esclavo calcula todas las combinaciones posibles, el proceso maestro le indica a cada proceso el número de combinaciones para las que debe evaluar las estimaciones en paralelo y la combinación de partida. El número de combinaciones asignado a cada proceso viene determinado por la aplicación de la técnica de equilibrio de carga desarrollada en este trabajo de investigación.

```
// ... Acción 1: lectura de los datos de entrada
// ... Acción 2: generación de representación interna y obtención de dependencias
for (procesos VHDL) {
    // ... Acción 3: generación de combinación de recursos
    // ... Acción 4: cálculo de las estimaciones
}
// ... Acción 5: generación de fichero de salida
```

Figura 5.25 Esqueleto de la ejecución secuencial de la fase de estimaciones de GACSYS

A continuación se describen las acciones que realizan los procesos dependiendo del computador donde se desarrollen.

Esqueleto para el proceso maestro

Respecto a la ejecución secuecial, las acciones realizadas por el proceso maestro son las mismas, excepto la acción 4, dado que éste no realiza el cálculo de estimaciones. Además, se añaden cuatro nuevas acciones relacionadas con la biblioteca *LAMGAC*: actualización de número de procesos, estimación del número de combinaciones, distribución de datos, recepción de controlada de resultados y contabilización de combinaciones no evaluadas. Debido a que puede haber combinaciones cuyas estimaciones no sean recibidas en el proceso maestro debido a CP no disponibles, estas acciones se repiten hasta que no queden combinaciones por evaluar. A continuación se detallan estas acciones:

- *Acción 4.1*: *actualización del número de procesos*. Se invoca la función *LAMGAC_Update()* para actualizar el número de procesos. A continuación, y si un nuevo proceso se expande en un CP, hay que comunicarle los datos iniciales necesarios para que pueda realizar los cálculos. Entre otros, se le comunica el identificador del proceso VHDL a partir del cual debe realizar los cálculos. Por otro lado, si existe algún proceso del programa paralelo en un CP que se quiere desvincular, se le envía un mensaje para que acabe su ejecución.

- *Acción 4.2*: *estimación del número de combinaciones.* Después de conocer los cambios producidos en la VM_LAM, se invoca la función *LAMGAC_Balance()* para obtener el número de combinaciones que debe calcular cada proceso y la secuencia de envío a éstos. El tamaño del envío de datos es siempre de dos números enteros (número de combinaciones y combinación de partida), y por tanto, es independiente del número de combinaciones. Respecto a los datos recibidos, coincide con el tamaño de tres números reales (tiempo, potencia y área), y se recibe tantas series como número de combinaciones estime. Por tanto, el valor de los parámetros *sizedata* y *sizeresults* de esta función se fija a *1* (mínimo valor) y a *3*sizeof(float)*, respectivamente.
- *Acción 4.3*: *distribución de datos.* Se envía a cada proceso el número de combinaciones con las que debe operar y la combinación de partida.
- *Acción 4.4*: *recepción controlada de resultados.* Una vez realizada la distribución de filas, se implementa el mecanismo de recepción controlada explicado en la sección 2 del capítulo anterior para recibir las estimaciones realizadas por cada proceso.
- *Acción 4.5*: *contabilización de combinaciones no evaluadas.* En el caso de existir combinaciones no procesadas debido a CP no disponibles, la evaluación de las estimaciones correspondientes a éstas se distribuyen entre el resto de procesos disponibles. En este caso, se vuelven a repetir las acciones desde la acción 4.1 con las combinaciones no procesadas. Cuando todas las combinaciones sean procesadas se pasa a la siguiente iteración.

En la figura 5.26 se pueden ver las nuevas acciones que realiza el proceso maestro con respecto a la ejecución secuencial.

Esqueleto para los procesos en los CF

El esqueleto para estos procesos es el mismo que el código secuencial con la salvedad de

```
// .. Sentencias del bloque INITIALIZE
LAMGAC_Init_an (argv);
// .. Acción 1 y 2 de la ejecución secuencial
for (procesos VHDL) {
    // ... Acción 3: generación de combinación de recursos
    do {
        // ... Acción 4.1: actualización del número de procesos
            LAMGAC_Update (..., &NI, &ND, ...);
            // Comunicar proceso VHDL actual a los procesos recién vinculados
            // Comunicar a los porcesos que se van a desvincular que finalicen el programa
        // ... Acción 4.2: estimación del número de combinaciones
            LAMGAC_Balance (..., micalc, 1, 3*sizeof(float), total_combinacines, order, ndata);
        // ... Acción 4.3: distribución de datos
        // ... Acción 4.4: recepción controlada de resultados
        // ... Acción 4.5: contabilización de combinaciones no evaluadas
    }while (repetir_it);
}
// ... Acción 5: generación de fichero de salida
LAMGAC_Finalize();
MPI_Finalize();
```

Figura 5.26 Esqueleto del proceso maestro de la fase de estimaciones de GACSYS

que realizan las estimaciones para un conjunto limitado de combinaciones, y por ello, deben recibir el número de combinaciones y la combinación de partida, y además, deben enviar al proceso maestro las estimaciones obtenidas y recibir una señal para saber si deben repetir la iteración o pasar a la siguiente. Las acciones nuevas, con respecto al programa secuencial, son las siguientes:

- *Acción 3.1*: *recepción del número de combinaciones y combinación de partida.* En cada iteración, el proceso recibe el número de combinaciones a evaluar y la combinación de partida.
- *Acción 5*: *enviar resultados.* Una vez concluidas las estimaciones, cada proceso envía los resultados obtenidos al proceso maestro.
- *Acción 6: recepción de señal sobre repetición de iteración.* El proceso recibe una señal del proceso maestro que le indica si hay que repetir los cálculos con

```
// .. Sentencias del bloque INITIALIZE
LAMGAC_Init_fn ();
// .. Acción 1 y 2 de la ejecución secuencial
for (procesos VHDL) {
    // ... Acción 3: generación de combinación de recursos
    do {
        // ... Acción 3.1: recepción del número de combinaciones y combinación de partida
        for (i=comb_partida;i <=(comb_partida + num_combinaciones;i++)
                // ... Acción 4: Cálculo de estimaciones
        // .. Acción 5: enviar resultados
        // .. Acción 6: recepción de señal sobre repetición de iteración
    }while (repetir_it);
}
LAMGAC_Finalize_slave (my_rank);
MPI_Finalize();
```

Figura 5.27 Esqueleto de los procesos en los CF de la fase de estimaciones de GACSYS

el mismo proceso VHDL y una nuevo conjunto de combinaciones o se pasa a la siguiente iteración.

La figura 5.27 muestra las acciones de este tipo de procesos.

Esqueleto para los procesos en los CP

El esqueleto de los procesos que se desarrollan en los CP es el mismo que de aquellos que se ejecutan en los CF, con las siguientes salvedades: al comienzo de la ejecución se invoca la función *LAMGAC_Init_pn()* en lugar de *LAMGAC_Init_fn()*. Después de la acción 2 se recibe el identificador del proceso VHDL a partir del cual se evaluan las estimaciones, y por último, también hay que considerar la finalización del programa si el proceso ha solicitado desvincularse.

En la figura 5.28 se presenta el programa paralelo para los procesos que se ejecutan en los CP.

```
// ..  Sentencias del bloque INITIALIZE_EP
LAMGAC_Init_pn (0,&my_id,...);
// .. Acción 1 y 2 de la ejecución secuencial
// .. Recibir identificador de proceso VHDL actual
for (proceso VHDL actual hasta el último o DESVINCULAR) {
      // ... Acción 3: generación de combinación de recursos
      do {
          // ... Acción 3.1: recepción del número de combinaciones y combinación de partida
          if (señal != DESVINCULAR) {
                for (i=comb_partida;i <=(comb_partida + num_combinaciones;i++)
                    // ... Acción 4: cálculo de estimaciones
                // .. Acción 5: enviar resultados
                // .. Acción 6: recepción de señal sobre repetición de iteración
          }
      }while (repetir_it);
}
LAMGAC_Finalize_slave (my_rank);
MPI_Finalize();
```

Figura 5.28 Esqueleto de los procesos en los CP de la fase de estimaciones de GACSYS

5.5.2 Análisis de los resultados

Como descripción VHDL de un sistema, se ha utilizado un subsitema de reconocimiento de voz. El objetivo de este subsistema es la obtención de un conjunto de coeficientes (*cepstrum*) a partir de una señal de voz muestreada a 8 Khz y dividida en tramas de 128 muestras. La especificación del subsistema está formada por ocho procesos VHDL concurrentes, los cuales realizan entre otras operaciones, prenfatización de señal, escalado, transformada de Fourier, etc. Una descripción más detallada del sistema puede encontrarse en [SaC99]. Por otro lado, el número de unidades funcionales que pueden ser implementadas en paralelo se ha limitado a cuatro, estando disponibles dos tipos diferentes de multiplicadores y cuatro sumadores.

Para esta aplicación, se realizaron simulaciones en las dos agrupaciones, C1 y C2, mostradas en la tabla 5.3.

En la figura 5.29 se muestran: el tiempo medio de ejecución del programa secuencial en dos computadores, el tiempo medio de la ejecución del programa paralela utilizando una planificación equilibrada por bloques (NLB_Cx) y de la ejecución paralela utilizando la técnica de equilibrio de carga (LB_Cx). En las implantaciones paralelas, los procesos en los CP se expandieron en la primera iteración del algoritmo. En este caso, se observa que la ejecución secuencial es más lenta, tanto en el recurso más rápido como en el más lento, que la ejecución de las aplicaciones paralelas, salvo para la simulación NLB_C2, donde el menor número de procesadores implica que los computadores más lentos tengan que realizar más cálculos. Estos resultados se deben al elevado tiempo de cálculo frente al tiempo de comunicación. Con respecto a las soluciones paralelas, se observa claramente que el tiempo de ejecución de las implantaciones que utiliza el mecanismo de equilibrio de carga es inferior a las implantaciones no balanceadas. Sin embargo, el aumento de computadores en C1 no se ve reflejado holgadamente en el tiempo de ejecución frente a la agrupación C2, debido en parte a la acción 3, la cual es costosa computacionalmente y debe ser realizada en cada iteración por todos los procesos.

En la figura 5.30 a) se muestra la desviación típica del tiempo medio de ejecución de todos los procesos por iteración. Claramente se deduce que las implantaciones con el mecanismo de equilibrio de carga están balanceadas con respecto a las no balanceadas. Los picos obtenidos en la iteración 1, 4 y 8 son debido a que en dichas iteraciones se realizan más cálculos que en el resto y el mecanismo de equilibrio de carga no conoce este hecho de antemano y se realiza la estimación de la distribución en función del rendimiento de la iteración previa. A tenor de los resultados, este aumento en la desviación típica es despreciable.

Por otro lado, en la figura 5.30 b) se muestra la desviación típica del tiempo medio de ejecución para las implantaciones balanceadas con el mecanismo de equilibrio de carga cuando se producen vinculaciones y no disponibilidad de los CP para ambas configuraciones. La aplicación comienza con un proceso desarrollándose en cada CF, y en la primera iteración se expande un proceso en todos los CP, salvo en el CP3. En la

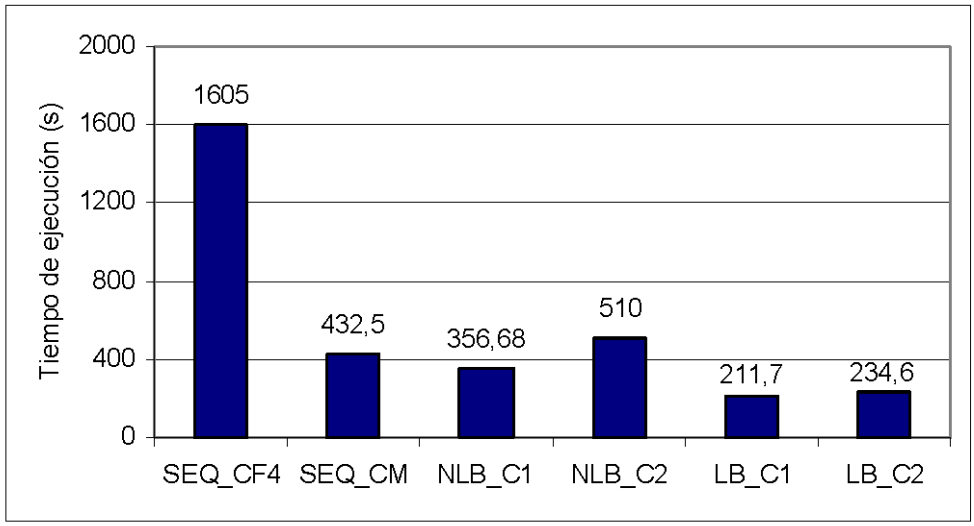
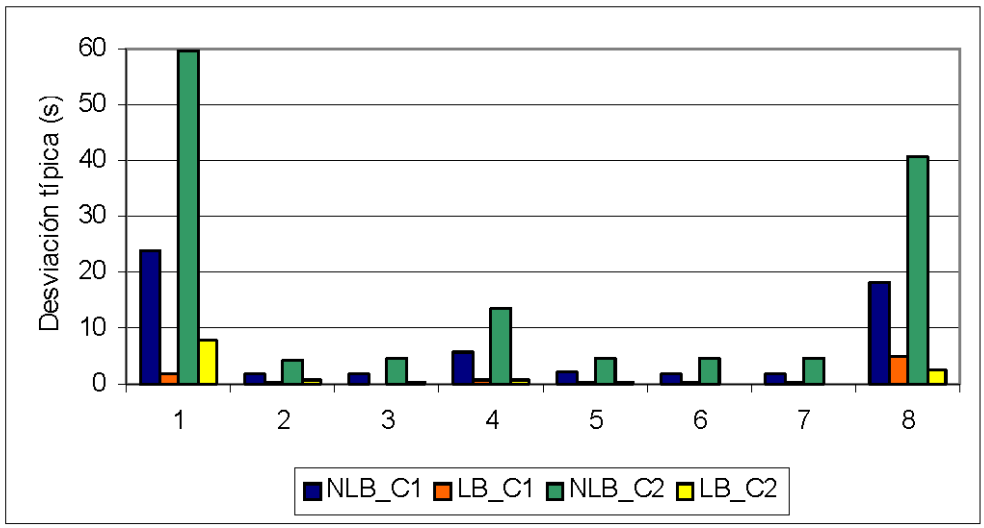


Figura 5.29 Eficiencia del protocolo de equilibrio de carga

iteración 3, se expande un nuevo proceso en el CP3, y en la iteración 5 y 8 se fuerza la no disponibilidad de los CP1 y CP3, respectivamente, hasta el final de la aplicación. En este caso la aplicación realiza 10 iteraciones porque en la iteración 6 y 9 se realizan cálculos con las combinaciones cuyas estimaciones no fueron obtenidas en la iteración anterior. Claramente se observa que los resultados obtenidos para la agrupación C1 son mejores que los de la C2, debido a que al repartir la misma carga entre más procesos los tiempos de ejecución de cada proceso son menores y por tanto, el desequilibrio será menor.

Por último, en la figura 5.31 se muestra la sobrecarga que introduce la ejecución de la hebra *Snmp_Ping* en el tiempo de ejecución de la aplicación. Para llevar a término esta evaluación, se forzó la ejecución de las hebras desde el comienzo del programa paralelo hasta el final. Dado que en la configuración utilizada (C1) sólo se disponían de tres CP, para simular los experimentos con un número de hebras superior a tres, se replicó el número de éstas realizando las consultas sobre los CP de forma equitativa. Como se observa el tiempo de ejecución de la aplicación se incrementa ligeramente con la ejecución de hebras de control. En el peor caso (12 hebras), se incrementa aproximadamente 0,4 segundos, lo que supone un incremento alrededor del 0,19%. En cualquier caso, este incremento es despreciable frente a la ventaja que se obtiene

a)



b)

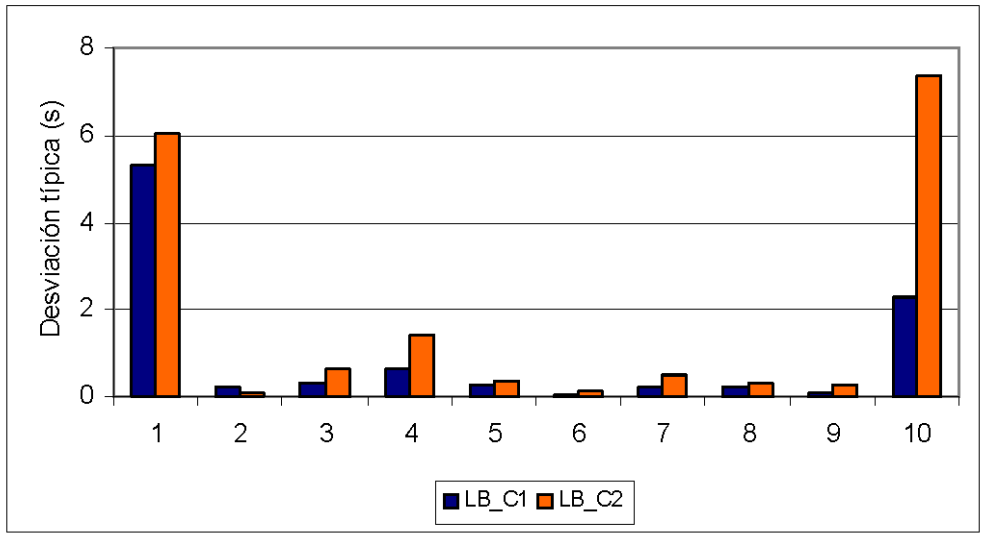


Figura 5.30 Desviación típica por iteración: a) balanceado y no balanceado, b) con vinculación y no disponibilidad de CP en tiempo de ejecución

controlando los CP, y además, hay que tener en cuenta que en un caso real la sobrecarga es inferior debido a que es poco probable que la hebra permanezca en ejecución durante toda la aplicación.

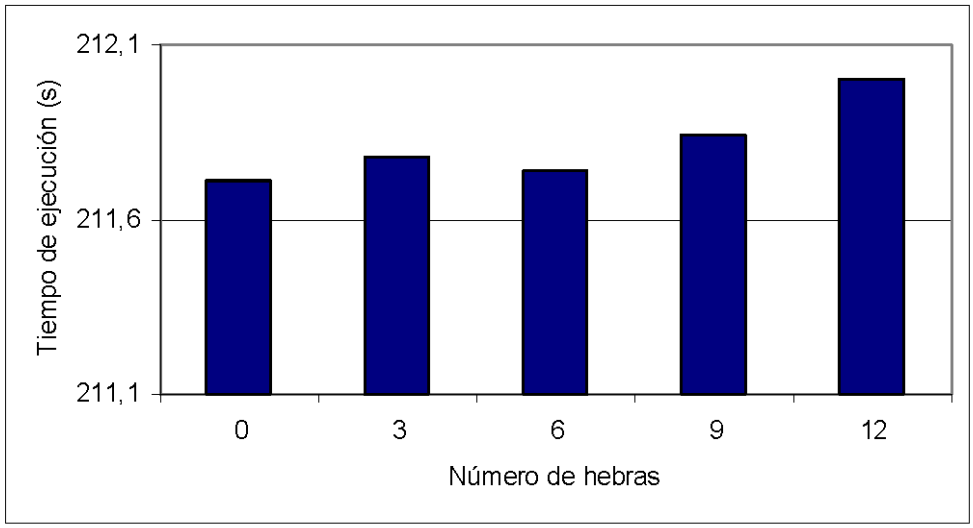


Figura 5.31 Sobrecarga de la hebra de control sobre los CP

5.6 Simulación de un canal difuso de infrarrojos

Las comunicaciones por infrarrojos son adecuadas por su compatibilidad internacional y una reducción en la complejidad del receptor que procesa las señales recibidas [KaB97], permaneciendo las comunicaciones confinadas dentro de la habitación donde se originan. Cuando no existe visión directa entre el emisor y el receptor es necesario disponer de configuraciones alternativas como el *enlace difuso* y el *enlace Q-difuso*. El enlace difuso hace uso de las superficies del entorno para que reflejen la potencia óptica emitida dentro de la habitación.

La simulación de sistemas infrarrojos para aplicaciones en interiores es un problema de ingeniería importante, y consiste en calcular la distribución temporal de potencia que alcanza un receptor óptico debido a la emisión de un impulso de luz generado en un emisor. Es decir, el objetivo de la simulación es caracterizar el canal de infrarrojos mediante el cálculo de la respuesta al impulso del sistema. Para simular estos problemas se considera que tanto el emisor como el receptor están ubicados dentro de una habitación, y las paredes de la misma actúan como superficies reflectoras. Estas paredes se dividen en áreas cuadradas denominadas celdas. Una celda que recibe un haz de

potencia óptica en un instante t_i, por efecto de la reflexión, ilumina a las celdas que están en las otras paredes en los posteriores instantes de tiempo. La distribución de potencia en el receptor es la suma de la potencia que llega de forma directa e indirecta. La primera se debe a la visión directa entre emisor y receptor, y la segunda se debe al conjunto de reflexiones en las celdas que alcanzan al receptor. Por este motivo, la distribución de potencia será un conjunto de impulsos con diferentes valores y retardos de propagación, donde el valor más alto corresponde a la potencia óptica recibida directamente desde el emisor, y el resto se corresponde con la potencia debido al conjunto de reflexiones que alcanzan el receptor.

Una especificación más completa de este algoritmo, donde se incluye las características del canal en enlaces infrarrojos no guiados, el modelo del emisor óptico, del receptor y del reflector, y el modelo de propagación para un sistema difuso puede ser consultado en [Mac01].

5.6.1 Implantación

Para explicar con mayor claridad la obtención de la distribución de potencia que alcanza al receptor nos vamos a basar en la explicación de los cálculos realizados en la implantación secuencial del algoritmo, figura 5.32.

Para cada instante de simulación, bucle de la variable *d*, se recorren todas las paredes de la habitación y para cada una de ellas se analiza cada una de sus celdas buscando las celdas iluminadas. Por cada celda iluminada, se calcula el efecto que produce ésta sobre las demás celdas que no pertenecen a su pared y el retardo que tarda en llegar la potencia a dichas celdas desde la celda iluminada. Con estos datos se actualiza la matriz de potencias *m_pot* que es donde se almacena la potencia que llega a cada una de las celdas de la habitación durante el intervalo de simulación.

A continuación, se calcula la potencia que llega a los receptores desde las celdas iluminadas. Para ello, se recorren todos los receptores, comprobando si la celda iluminada ilumina también a algún receptor. En caso afirmativo, se calcula la potencia

```
for (d= 0; d< long_canal; d++) {
    for (pared_o= 0; pared_o< max_num_paredes; pared_o++)
      for (celda_o= 0; celda_o< tam_pared[pared_o]; celda_o++)
       if (celda_o_iluminada) {
          for (pared_d= pared_o+1; pared_d< max_num_paredes; pared_d++) {
                // ... Calcular efecto y retardo entre celda origen (celda_o) y destino (celda_d)
                // ... Calcular potencia y actualizar m_pot
          }
          for (pared_d= pared_o-1; pared_d>=0; pared_d--) {
                // ... Calcular efecto y retardo entre celda origen (celda_o) y destino (celda_d)
                // ... Calcular potencia y actualizar m_pot
          }
          for (receptor= 0; receptor< max_num_receptores; receptor++)
                if (receptor_iluminado_por_celda_o)
                 // ... Calcular potencia entre celda origen y receptor y actualizar pot
       }
}
```

Figura 5.32 Algoritmo secuencial para la simulación de un sistema IR difuso

que llega al receptor y en qué instante de tiempo, y se actualiza la matriz *pot* que es donde se almacena la distribución temporal de potencia que llega al receptor.

Dependiendo del número de celdas, el cálculo de la distribución de potencia que recibe el receptor puede tomar un tiempo excesivo. Teniendo en cuenta que en un instante de simulación, la potencia con la que contribuye la reflexión en una celda depende de los instantes anteriores, la potencia con la que contribuye cada celda se puede calcular en paralelo dentro de la misma iteración.

En el programa paralelo de la simulación del canal de infrarrojos se asigna un conjunto de celdas de una pared entre los procesos en ejecución, repitiendo esta técnica para el resto de paredes de la habitación. Esta asignación se realiza en cada iteración y la dimensión de cada conjunto viene determinado por la llamada a la función *LAMGAC_Balance()*. Una vez que los procesos esclavos calculan la potencia reflejada por el conjunto de celdas en una iteración (instante de simulación), éstos envían la

distribución de potencia al proceso maestro. El proceso maestro realiza la suma de las contribuciones de potencia de cada celda y distribuye la matriz de potencia a todos los procesos para realizar los cálculos en la siguiente iteración. A continuación se describen las principales acciones de los tres esqueletos del programa paralelo.

Esqueleto para el proceso maestro

Las acciones del proceso maestro son las siguientes:

- *Acción 1*: *lectura de datos, inicialización de estructuras y distribución de datos iniciales*. El proceso maestro lee los datos de entrada de la aplicación, inicializa las estructuras de datos correspondientes y envía los datos al resto de procesos. Los datos de entrada a la aplicación son, básicamente, datos relativos a los emisores, receptores, información de la habitación, tamaño de la celda, el tiempo de simulación, el tiempo mínimo y máximo de radiación entre dos celdas y el nombre del archivo donde se almacena el resultado.
- *Acción 2*: *actualización del número de procesos*. En cada iteración del bucle de la variable *d*, se invoca la función *LAMGAC_Update()* para actualizar el número de procesos en los CP. A continuación, y si un nuevo proceso se expande, hay que comunicarle los datos iniciales necesarios para que pueda realizar los cálculos. Entre otros, se le comunica los datos referentes al emisor, receptor, dimensión de la habitación, la fila de la matriz de potencia con la que el proceso inicia los cálculos, y el valor inicial para el bucle de variable *d*. Si existe algún proceso en un CP que se quiere desvincular, se le envía un mensaje para que acabe su ejecución, y se recibe de éste la distribución temporal de potencia en los receptores que ha calculado.
- *Acción 3*: *envío de la matriz de potencias*. Se envía a todos los procesos los valores de potencia almacenados en la estructura *m_pot*. Se utilizan comunicaciones punto a punto para los procesos en los CP y comunicaciones colectivas para los procesos en los CF.
- *Acción 4*: *estimación del conjunto de celdas que debe procesar cada esclavo.*

Después de conocer los cambios producidos en la VM_LAM, se invoca la función *LAMGAC_Balance()* para conocer el conjunto de celdas que debe procesar cada proceso y la secuencia de envío a éstos. El tamaño del envío de datos hacia cada proceso es siempre de 12 números enteros (límite inferior y superior del conjunto de celdas asignadas en cada una de las seis paredes de la habitación), y por tanto, es independiente del número de celdas procesadas. Respecto, a los datos recibidos, coincide con el tamaño de la matriz de potencias, es decir, el número de celdas totales y un número entero que indica el número de celdas procesadas. Este último dato es necesario debido a que los cálculos en los procesos esclavos se realizan sólo cuando las celdas están iluminadas, y por tanto, se debe utilizar en la función *LAMGAC_Store_info()* para estimar correctamente el rendimiento del proceso (celdas computadas frente a tiempo de cómputo). El envío de este dato es despreciable frente a la matriz de potencias. Por tanto, el valor de los parámetros *sizedata* y *sizeresults* de esta función se fija a *1* (mínimo valor) y a *sizeof(float)*, respectivamente. Además, después de conocer el número de celdas a procesar por cada esclavo, éstas se reparten equitativamente entre todas las paredes, asignando a cada proceso un conjunto de celdas por cada una de las seis paredes. Para ello, se utilizan las matrices *imin* e *imax* (cada fila almacena los límites de cada pared de un proceso del programa paralelo-distribuido).

- *Acción 5*: *distribución de los límites del conjunto de celdas.* Después de conocer la cantidad de celdas a procesar por cada proceso, se envían los datos almacenados en *imin e imax* a cada proceso, distribuyéndose según la secuencia de envío obtenida en la acción anterior. El tamaño de esta comunicación de datos es la suma del número de *bytes* correspondientes a 12 números enteros. Por otro lado, si en una iteración dada no se distribuyen filas a uno o varios procesos, a éstos se les envía una señal para que no realicen operaciones de cálculo y pasen a la espera en la siguiente iteración.
- *Acción 6*: *recepción controlada de resultados.* Una vez realizada la

distribución de datos, se implementa el mecanismo de recepción controlada explicado en la sección 2 del capítulo anterior para recibir los datos calculados por los procesos esclavos. Durante este mecanismo se recibe la matriz de potencias y el número de celdas computadas por cada proceso. Al finalizar esta recepción, la matriz de potencias contendrá la suma de las contribuciones obtenidas por todos los procesos.

- *Acción 7*: *contabilización de conjuntos no evaluados.* En el caso de existir conjuntos de celdas no procesadas debido a CP no disponibles, el cálculo sobre las celdas correspondientes se reparte entre el resto de procesos disponibles. En este caso, se vuelven a repetir las acciones desde la acción 2 con las celdas no procesadas. Para ello, se le envía la señal correspondiente a cada uno de los procesos esclavos. Cuando todas las celdas sean evaluadas se pasa a la siguiente iteración.
- *Acción 8*: *recepción de resultados.* El proceso maestro recibe, al finalizar el bucle para la variable de iteración *d*, la distribución temporal de potencia en cada uno de los receptores, calculada en paralelo entre todos los procesos.
- *Acción 9*: *generación del fichero de salida.* El proceso maestro escribe en un fichero la distribución temporal de potencia recibida en el receptor.

En la figura 5.33 se muestra el esqueleto para el proceso maestro.

Esqueleto para los procesos en los CF

Básicamente, las acciones realizadas por los procesos que se ejecutan en los CF son las siguientes:

- *Acción 1*: *recepción inicial de datos.* Los procesos reciben del proceso maestro los datos necesarios para inicializar las estructuras y comenzar la aplicación.
- *Acción 2*: *recepción de la matriz de potencias.* En cada iteración del bucle *d* se recibe la matriz de potencias.

```
// .. Sentencias del bloque INITIALIZE
LAMGAC_Init_an (argv);
// ... Acción 1: lectura de datos, inicialización de estructuras y distribución de datos iniciales
for (d= 0; d< long_canal; d++) {
      do {
          // ... Acción 2: actualización del número de procesos
                    LAMGAC_Update (..., &NI, &ND, ...);
                    // Comunicar iteración actual a los procesos recién vinculados
                    // Comunicar a los procesos que se van a desvincular que finalicen el programa
          // ... Acción 3: envío de la matriz de potencias.
          // ... Acción 4: estimación del conjunto de celdas que debe procesar cada esclavo
          // ... Acción 5: distribución de los límites del conjunto de celdas
                    LAMGAC_Balance (..., micalc, 1, sizeof(float), total_celdas, order, ndata);
          // ... Acción 6: recepción controlada de resultados
          // ... Acción 7: contabilización de conjuntos no evaluados
      }while (repetir_it);
}
// ... Acción 8: recepción de resultados
// ... Acción 9: generación del fichero de salida
LAMGAC_Finalize ();
MPI_Finalize();
```

Figura 5.33. Esqueleto del proceso maestro en la simulación del canal de infrarrojos

- *Acción 3*: *recepción de los límites del conjunto de celda de cada pared.* Se reciben doce números enteros correspondientes al límite inferior (*imin*) y al límite superior (*imax*) del conjunto de celdas de cada una de las seis paredes sobre las que se van a realizar los cálculos.

- *Acción 4*: *calcular.* Para cada instante de la simulación, bucle de la variable *d*, el proceso recorre cada una de las paredes de la habitación, bucle de variable *pared_o* en la figura 5.32. Dentro de cada pared, opera con el conjunto de celdas que le han sido asignadas, (desde *imin[pared_o]* hasta *imax[pared_o]*). Por cada celda iluminada, se calcula la potencia que ésta envía a cada una de las celdas que no pertenecen a la pared *pared_o*. Para calcular la potencia que llega a la celda destino debido a la iluminación de la celda origen es necesario

obtener el efecto y el retardo entre ambas celdas. Una vez calculada la potencia, se actualiza la estructura de datos *m_pot*.

A continuación se recorren los receptores, bucle de la variable *receptor*, comprobando si la celda origen iluminada puede iluminar a algún receptor. En caso afirmativo, se calcula la potencia que llega al receptor y se actualiza *pot*. Una vez que el proceso ha recorrido todas las celdas asignadas dentro de la habitación, inicializa a 0.0 la fila de *m_pot* que ha analizado.

- *Acción 5*: *comunicación de potencias*. Una vez finalizados los cálculos, cada proceso comunica al proceso maestro los valores de potencia en *m_pot*, correspondiente al instante de simulación, así como el número de celdas iluminadas procesadas.
- *Acción 6: recepción de señal sobre repetición de iteración.* El proceso recibe una señal del proceso maestro que le indica si hay que repetir la iteración con un nuevo conjunto de celdas o se pasa a la siguiente iteración.
- *Acción 7*: *enviar resultados*. Al finalizar el bucle de variable *d*, se envía al proceso maestro la distribución temporal de potencia en cada uno de los receptores, calculada en paralelo entre todos los procesos.

En la figura 5.34 se muestran las acciones realizadas por los procesos de los CF.

Esqueleto para los procesos en los CP

El esqueleto de los procesos que se desarrollan en los CP es el mismo que de aquellos que se ejecutan en los CF, con las siguientes salvedades: al comienzo de la ejecución se invoca la función *LAMGAC_Init_pn()* en lugar de *LAMGAC_Init_fn()*; antes de comenzar el bucle de la variable *d* se recibe un número entero que indica la iteración de comienzo de dicho bucle, y por último, también hay que considerar la finalización del algoritmo si el proceso ha solicitado desvincularse.

En la figura 5.35 se presenta el algoritmo paralelo para estos procesos.

```
// .. Sentencias del bloque INITIALIZE
LAMGAC_Init_fn (argv);
// ... Acción 1: recepción inicial de datos
for (d= 0; d< long_canal; d++) {
      do {
          // ... Acción 2: recepción de la matriz de potencias
          // ... Acción 3: recepción del los límites del conjunto de celda de cada pared
          for (pared_o= 0; pared_o< max_num_paredes; pared_o++) {
                    for (celda_o= imin[pared_o]; celda_o<imax[pared_o]; celda_o++)
                              if (celda_o_iluminada) {
                                        // ... Acción 4: calcular
                              }
          }
          // ... Acción 5: comunicación de potencias
          // ... Acción 6: recepción de señal sobre repetición de iteración
      }while (repetir_it);
}
// ... Acción 7: enviar resultados
LAMGAC_Finalize_slave (my_id);
MPI_Finalize();
```

Figura 5.34. Esqueleto del proceso que se desarrolla en los CF

5.6.2 Análisis de los resultados

Para la simulación del canal de comunicación infrarrojo difuso se utilizó la configuración A propuesta en [BKK93] que consiste en una habitación de dimensiones 5x3x3 m^3. El emisor está ubicado en el centro de la habitación junto al techo y apuntando hacia el suelo. Este emisor emite un impulso de señal de 1 vatio. El receptor está situado en el suelo y apunta hacia el techo. El intervalo de simulación es de 60 nanosegundos. Cada pared está dividida en celdas de 15 centímetros, y tanto el emisor como el receptor están visión directa. Con estos datos, la aplicación realiza 170 iteraciones.

Para esta aplicación, se realizaron simulaciones en las dos agrupaciones, C1 y C2, mostradas en la tabla 5.3.

```
// .. Sentencias del bloque INITIALIZE_EP
LAMGAC_Init_pn (0, &my_id,...);
// ... Acción 1: recepción inicial de datos
// ... Se recibe iteración actual del bucle de variable d
for (d= iter; d< long_canal o DESVINCULAR; d++) {
      do {
          // ... Acción 2: recepción de la matriz de potencias
          if (señal_c!=DESVINCULAR) {
                    // ... Acción 3: recepción del los límites del conjunto de celda de cada pared
                    for (pared_o= 0; pared_o< max_num_paredes; pared_o++) {
                              for (celda_o= imin[pared_o]; celda_o<imax[pared_o]; celda_o++)
                                        if (celda_o_iluminada) {
                                                  // ... Acción 4: Calcular
                                        }
                    }
                    // ... Acción 5: comunicación de potencias
                    // ... Acción 6: recepción de señal sobre repetición de iteración
          }
      }while (repetir_it);
}
// ... Acción 7: enviar resultados
LAMGAC_Finalize_slave (my_id);
MPI_Finalize();
```

Figura 5.35. Esqueleto del proceso que se desarrolla en los CP

En la figura 5.36 se muestran el tiempo medio de ejecución del algoritmo secuencial, de la ejecución paralela utilizando una planificación equilibrada por bloques (NLB_Cx) y de la ejecución paralela utilizando la técnica de equilibrio de carga (LB_Cx). En las implantaciones paralelas, los procesos de los CP se expandieron en la primera iteración. Se observa que la ejecución secuencial es más lenta, tanto en el recurso más rápido como en el más lento, que la ejecución de las aplicaciones paralelas que utilizan el mecanismo de equilibrio de carga. Sin embargo, la ejecución en la máquina más rápida tarda menos que la aplicación paralela no balanceada. Incluso, el tiempo de ejecución secuencial en la máquina más lenta tarda menos que el paralelo no balanceado utilizando la configuración C2. Esto demuestra que una desintonización en los tiempos de ejecución

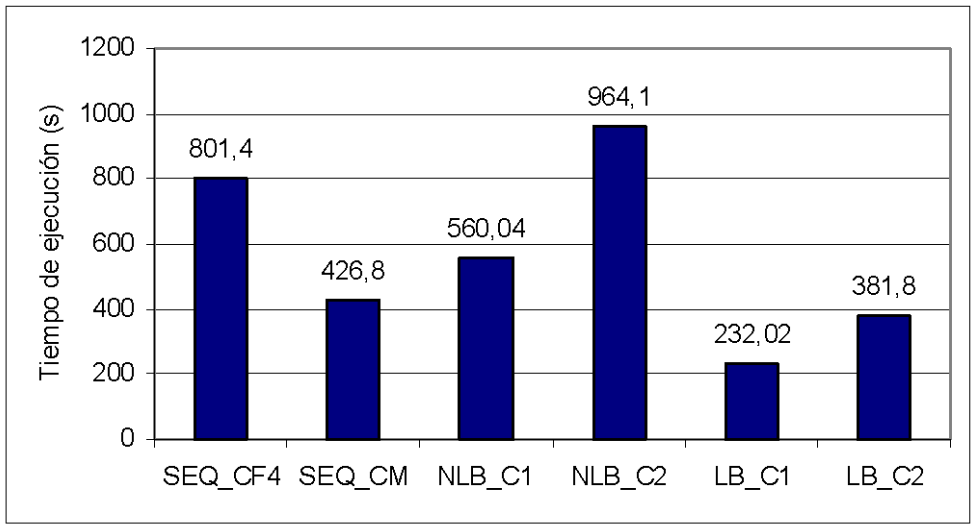


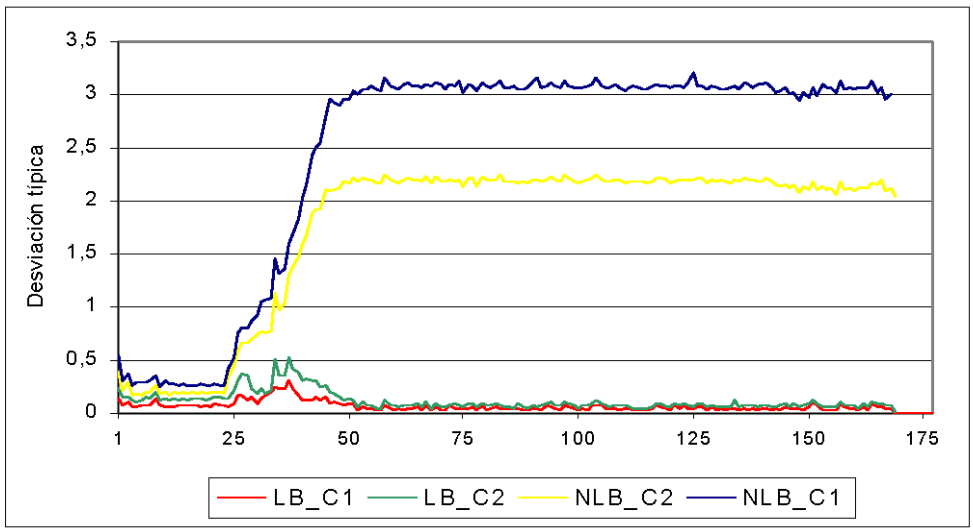
Figura 5.36 Eficiencia del protocolo de equilibrio de carga

de los procesos puede provocar que la aplicación paralela sea más lenta que el algoritmo secuencial.

En la figura 5.37 a) se muestra la desviación típica del tiempo medio de ejecución de los procesos en cada iteración para las soluciones paralelas usando ambas configuraciones. Como se aprecia, la desviación típica durante las primeras treinta iteraciones es similar con o sin mecanismo de equilibrio de carga debido a que durante las iteraciones iniciales se realizan pocos cálculos puesto que la mayoría de las celdas no están iluminadas. A partir de dicha iteración, los procesos empiezan a realizar más cálculos y va aumentando la diferencia entre las desviaciones presentadas.

Por otro lado, en la figura 5.37 b) se muestran la desviación típica y el tiempo de ejecución cuando se producen vinculaciones y existen CP no disponibles utilizando la configuración C1. En concreto, al comienzo de la iteración 75 se expande un nuevo proceso en CP3. En este punto, aumenta la desviación típica y el tiempo de ejecución, pero rápidamente el mecanismo de equilibrio de carga se adapta a la nueva situación. A partir de esta iteración, el tiempo medio de ejecución disminuye porque existe un proceso más. Durante la iteración 100, el CP1 sale de la cobertura del punto de acceso, y

a)



b)

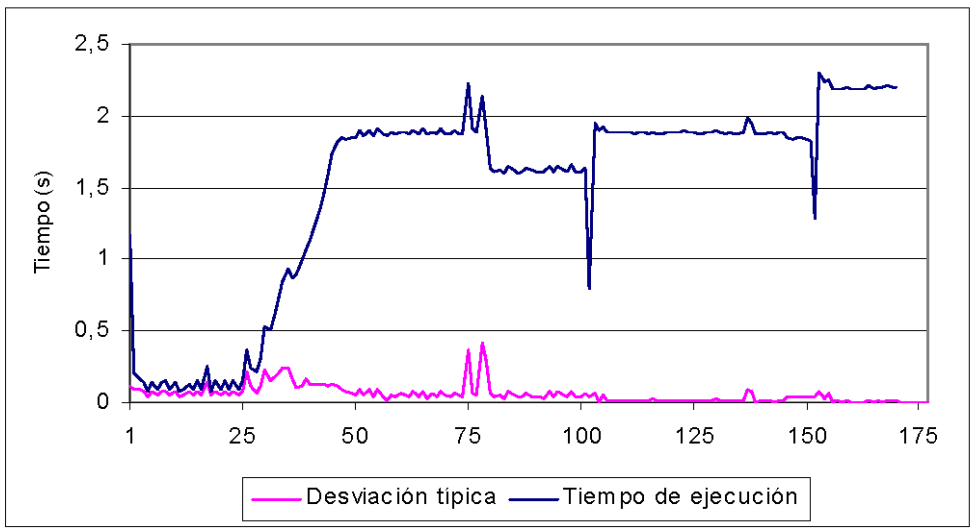


Figura 5.37 a) Desviación típica , b) Desviación típica y tiempo de ejecución con vinculaciones y computadores no disponibles

por tanto, el proceso maestro no se puede comunicar con el proceso que se desarrolla en ese computador. Lo mismo ocurre en la iteración 151 con CP3. Como se aprecia en la figura, en la iteración 101 y 152 el tiempo de ejecución disminuye debido a que se ralizan cálculos con las celdas no procesadas en la iteración previa. A partir de estas iteraciones, el tiempo medio de ejecución aumenta porque hay un proceso menos, comparando con la iteración previa, colaborando en los cálculos.

Por último, en la figura 5.38 se muestra la sobrecarga que introduce la ejecución de la

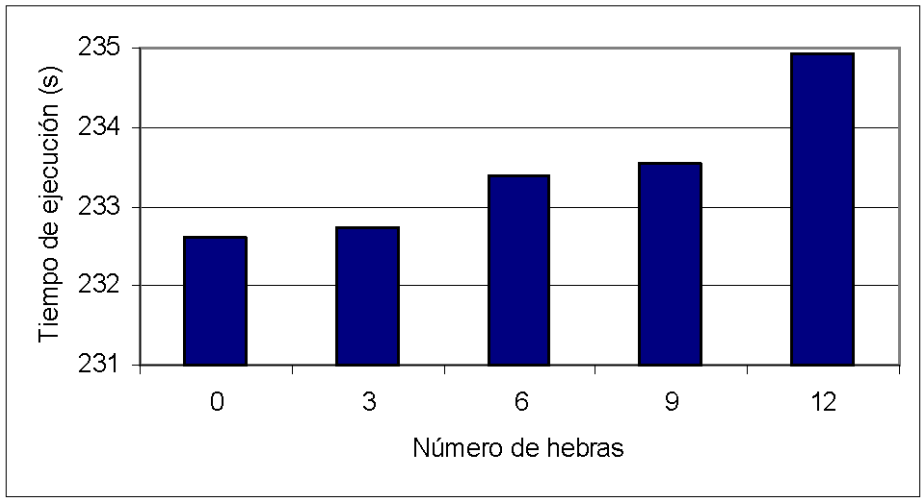


Figura 5.38 Sobrecarga de la hebra de control sobre los CP

hebra *Snmp_Ping* en el tiempo de ejecución de la aplicación. Para llevar a término esta evaluación, al igual que las aplicaciones anteriores se forzó la ejecución de las hebras desde el comienzo del programa paralelo hasta el final. Dado que en la configuración utilizada (C1) sólo se disponían de tres CP, para simular los experimentos con un número de hebras superior a tres, se replicó el número de éstas realizando las consultas sobre los CP de forma equitativa. Como se observa, el tiempo de ejecución de la aplicación se incrementa ligeramente con la ejecución de hebras de control. En el peor caso (12 hebras), se incrementa aproximadamente 2,5 segundos, lo que supone un incremento alrededor del 1,1%. En cualquier caso, este incremento produce una mínima sobrecarga sobre el tiempo de ejecución. Además, en una situación real es poco probable que las hebras permanezcan en ejecución durante toda la aplicación, ya que sólo se crean cuando algún CP está en un área de cobertura limitada.

6. Conclusiones y líneas futuras

En este capítulo se presentan las principales conclusiones de este trabajo de investigación y las líneas futuras de trabajo a seguir.

6.1 Conclusiones

El mercado tradicional de los computadores de escritorio y la utilización de redes cableadas está siendo complementado con la aparición de las redes de comunicación inalámbricas y la proliferación, cada día más, de recursos de computación portátiles. En concreto, la combinación de segmentos de red que utilizan la familia de estándares IEEE 802.3 e IEEE 802.11 puede ser utilizada para implementar un entorno de computación LAN-WLAN.

En el grupo de investigación en el que se ha desarrollado este trabajo se ha desmostrado que es posible, en este tipo de entornos de computación, la ejecución eficiente de aplicaciones paralelas restrictivas como las que siguen el paradigma Maestro/Esclavo, y en las que los cálculos paralelos son realizados en cada iteración y poseen dependencias de datos entre iteraciones. Debido a la heterogeneidad del entorno, al comportamiento dinámico y variación del número de computadores portátiles en tiempo de ejecución y al rendimiento aleatorio del canal de las redes inalámbricas, es necesario desarrollar un mecanismo de equilibrio de carga para ejecutar eficientemente este tipo de aplicaciones. En este sentido, en este trabajo de investigación se ha abordado este problema implantando un protocolo de equilibrio de carga que es capaz de adaptarse eficazmente a las condiciones cambiantes de este tipo de entornos de computación.

El protocolo de equilibrio de carga diseñado en este trabajo consta básicamente de dos partes. Por un lado, se ha diseñado una estrategia eficiente para estimar la distribución de carga que tiene en cuenta tanto el rendimiento de cada computador y de la red como la variación del número de CP en tiempo de ejecución. Por otro lado, se ha implantado una arquitectura, basada en SNMP, para obtener la información de rendimiento y estado de cada recurso del entorno en tiempo real. Esta información se suministra a la estrategia de equilibrio de carga para realizar de forma efectiva la estimación de la carga a distribuir a cada proceso del programa paralelo-distribuido. Además, debido al comportamiento dinámico y aleatorio de los CP, esta arquitectura controla la disponibilidad de dichos computadores en situaciones críticas de cobertura o energía de la batería. Si este control no se realiza, se producen desequilibrios en los

tiempos de ejecución cuando un CP desaparece de forma inesperada. Nótese la importancia de esta arquitectura, ya que existe un solapamiento entre la realización de los cálculos paralelos y la monitorización y control del estado de cada uno de los computadores del entorno de computación.

La implantación del protocolo de equilibrio de carga se ha realizado en el *middleware LAMGAC* para permitir al programador controlar de forma transparente, sin necesidad de conocer la estrategia de carga implementada ni la arquitectura SNMP, tanto el volumen de la distribución de datos entre los procesos del programa paralelo como la gestión de la variación dinámica de computadores en la VM_LAM en tiempo de ejecución.

Por último, el protocolo desarrollado en este trabajo de investigación se ha implantado en aplicaciones paralelas que resuelven problemas del álgebra lineal y de la Ingeniería de Telecomunicación. El resultado obtenido demuestra que es posible reducir el tiempo de ejecución de las aplicaciones paralelas equilibrando los tiempos de ejecución de los procesos en cada iteración. Además, se demuestra que la sobrecarga introducida por la ejecución del protocolo es despreciable frente al tiempo de ejecución de la aplicación, como se muestra en el capítulo de los resultados experimentales.

6.2 Líneas futuras

A continuación se indican algunas líneas futuras de trabajo que pueden ser desarrolladas como continuación al trabajo de investigación realizado en esta tesis doctoral para mejorar su rendimiento y capacidades.

En primer lugar, se podría dotar al *middleware LAMGAC* de un servicio de localización geográfica de CP para decidir a qué computadores, en función de su posición y cobertura, distribuir la carga. Incluso, con dicho servicio se podría caracterizar el movimiento habitual de los computadores y predecir con cierta probabilidad cuando éstos estarían disponibles. Teniendo esto en cuenta, se podría mejorar el tiempo de ejecución global al conocer con antelación los recursos disponibles en futuras iteraciones.

Por otro lado, el mecanismo de equilibrio de carga desarrollado en este trabajo utiliza una estrategia centralizada global, es decir, existe un único proceso encargado de estimar la carga a distribuir al resto de procesos. En este tipo de esquemas la escabilidad del sistema está limitada debido al cuello de botella que se puede generar en el computador donde se ubica el balanceador de carga. En este sentido, otra línea de trabajo futura sería el desarrallo de una estrategia distribuida donde cada proceso del programa paralelo pueda realizar la estimación de la carga que tiene que procesar, y a la vez conozca el volumen de trabajo que realizan el resto de procesos y el rendimiento del resto de computadores. Esto último es muy interesante porque se puede establecer un protocolo para el intercambio de volumen de carga entre un conjunto de procesos cualesquiera cuando éstos detectan desequilibrios entre ellos. Por ejemplo, un proceso puede estar descargado y utilizar dicho protocolo para conocer la carga a solicitar a un conjunto de procesos sobrecargados. Claro está que este nuevo esquema tendría el inconveniente de que todos los procesos tienen que saber el rendimiento y estado del resto, y por tanto, implica una comunicación de todos los procesos con todos, con la consecuente sobrecarga.

Por último, se podría mejorar la implantación de la función *LAMGAC_Balance()* para indicarle sobre qué procesos queremos estimar la distribución de carga. De esta forma, esta función se podría invocar varias veces en una misma iteración, pero obteniendo el volumen de distribución de datos para diferentes procesos. Esto sería muy interesante, debido a que con esta modificación podemos realizar la distribución de datos a determinados grupos de procesos, y por tanto, se implanta una estrategia centralizada local, donde se tiene un balanceador que distribuye la carga a los grupos de procesos, y en cada grupo existe un balanceador local que se encarga de mantener el equilibrio dentro del mismo.

Bibliografía

[AAM01] M. G. Arranz, R. Aguero, L. Muñoz, P. Mahonen, *"Behavior of UDP-based Applications over IEEE 802.11 Wireless Networks"*, 12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, San Diego, USA pp. 72-77, 2001.

[AbM88] H. Z. Abdalla, S. F. Midkiff, *"Performance of Analysis of Interconnection Networks for Massively Parallel Multicomputers"*, 2nd IEEE Symposium on the Frontiers of Massively Parallel Computation, pp. 639-642, 1988.

[AMP01] I. F. Akyldiz, G. Morabito, S. Palazzo, *"TCP-Peach: A New Congestion Control Scheme for Satellite IP Networks"*, IEEE/ACM Transactions on Networking, vol. 9, nº 3, pp. 307-321, junio 2001.

[AtH04] G. Attiya, Y. Hamam, *"Two Phase Algorithm for Load Balancing in Heterogeneous Distributed Systems"*, 12th IEEE Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'04), A Coruña, España, pp. 434-439, 2004.

[BaB95] A. Bakre, B. R. Badrinath, *"I-TCP: Indirect TCP for Mobile Hosts"*, 15th IEEE International Conference on Distributed Computing Systems (ICDCS'95), Vancouver, Canada, 136-143, 1995.

[BBR01] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, Yves Robert, *"Matrix Multiplication on Heterogeneous Platforms"*, IEEE Transactions on Parallel and Distributed Systems, vol. 12, nº 10, pp. 1033-1051, 2001.

[BiB92] R. Bianchini, K. Buskens, *"Implementation of On-Line Distributed System-Level Diagnosis Theory"*, IEEE Transactions on Computers, vol. 41, nº 5, pp. 616-626, 1992.

[Bin99] B. Bing, *"Measured Performance of the IEEE 802.11 Wireless LAN"*, 24th IEEE Conference on Local Computer Network (LCN'99), pp. 34-42, 1999.

[BKK93] John R. Barry, Joseph M. Kahn, Willian J. Krause, Edward A. Lee, David G. Messerschmitt, "*Simulation of Multipath Impulse Response for Indoor Wireless Optical Channels*", IEEE Journal on Selected Areas in Communications, vol. 11, nº. 3, abril 1993.

[BLR03] O. Beaumont, A. Legrand, Y. Robert, *"The Master-Slave Paradigm with Heterogeneous Processors"*, IEEE Transactions on Parallel and Distributed Systems, vol. 14 , issue: 9, ISSN: 1045-9219, pp. 897-908, 2003.

[BNA00] Robert Busby, Mitchel Neilsen, Daniel Andersen, *"Enhancing NWS for Use in an SNMP Managed Internetwork"*, 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS'00), Cancún, México, pp. 506-511, 2000.

[BRP99] P. Bhat, C. S. Raghavendra, V. Prasanna, *"Efficient Collective Communication in Distributed Heterogeneous Systems"*, 19th IEEE International Conference on Distributed Computing Systems, Texas, USA, pp. 15-24, 1999.

[BrP95] L. Brakmo, L. Peterson, *"TCP Vegas: End to End Congestion Avoidance on a Globus Internet"*, IEEE Journal on Selected Areas in Communications, vol. 13, nº 8, pp. 1465-1480, octubre 1995.

[BSM01] Tracy D. Braun, Howard Jay Siegel, Anthony A. Maciejewski, "*Heterogeneous Computing: Goals, Methods, and Open Problems*", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01), Las Vegas, USA, vol. I, pp. 7-18, 2001.

[CBP02] Rangini Chowdhury, Pravin Bhandakar, Manish Parashar, *"Adaptive QoS Management for Collaboration in Heterogeneous Environments"*, 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS'02), Florida, USA, pp. 90-100, 2002.

[CSC98] Juan P. Castellano, David Sánchez, Onassis Cazorla, Juan Bordón, Álvaro Suárez, *"GACSYS: a VHDL-based Hw/Sw Codesign Tool"*, 2nd International

Workshop on Design and Diagnostics of Electronic Circuit and Systems (DDECS'98), Szczyrk, Poland, pp. 293-299, 1998.

[DaP97] S. Dandamudi, A. Piotrowski, *"A Comparative Study of Load Sharing on Network of Workstations"*, International Conference on Parallel and Distributed Computing Systems (PDCS'97), New Orleans, USA, 1997.

[DeH02] Sasa Desic, Darko Huljenic, *"Agents Based Load Balancing with Component Distribution Capability"*, 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02), Berlín, Alemania, pp. 327-331, 2002.

[DHE05] S. Dhakal, M. H. Hayat, M. Elyas, J. Ghanem, C. T. Abdallah, *"Load Balancing in Distributed Computing Over Wireless LAN: Effects of Network Delay"*, IEEE Wireless Communications and Networking Conference (WCNC'05), New Orleans, USA, vol. 3, pp. 1755-1760, 2005.

[DuN96] Elías P. Duarte, Takashi Nanya, *"An SNMP-based Implementation of the Adaptive Distributed System-Level Diagnosis Algorithm for LAN Fault Management"*, IEEE Network Operations and Management Symposium, pp. 530-539, 1996.

[EgE02] M. Eggen, R. Eggen, *"Load Balancing on a Non-dedicated Heterogeneous Network of Workstations"*, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02), Las Vegas, USA, vol. II, pp. 856-862, 2002.

[ELZ86] D. L. Eager, E. D. Lazowska, J. Zahorjan, *"Adaptive Load Sharing in Homogeneous Systems"*, IEEE Transactions on Software Engineering, vol. 12, nº 2, pp. 141-154, 1986.

[FaA00] D. Famolari, P. Agrawal, *"Architecture and Performance of an Embedded IP Bluetooth Personal Area Network"*, IEEE International Conference on Personal Wireless Communications (PWC'00), Hyderabad, India, pp. 75-79, 2000.

[FlH99] S. Floyd, T. Henderson, *"The New Reno Modification to TCP's Fast Recovery Algorithm"*, RFC 2582, abril 1999.

[FLL03] Cheng P. Fu, William Lu, Bu Sung Lee, *"TCP Veno Revisited"*, IEEE Global Telecommunications Conference (GLOBECOM'03), San Francisco, USA, vol. 6, pp. 3253 - 3258, 2003.

[FMT02] S. Fujita, M. Masukawa, S. Tagshira, *"A Fast Branch-and-Bound Algorithm with an Improved Lower Bound for Solving the Multiprocessor Scheduling Problem"*, 9th IEEE International Conference on Parallel and Distributed

Systems (ICPADS'02), Taiwan, pp. 611-616, 2002.

[FoK97] Ian Foster, Carl Kesselman, *"Globus: A Metacomputing Infrastructure Toolkit"*, Journal on Supercomputer Applications, vol. 11, nº 2, pp. 115-128, 1997.

[FoK99] Ian Foster, Carl Kesselman, *"The GRID Blueprint for a New Computing Infraestructure"*, Morgan Kaufmann Publishers, Inc., San Francisco, California, ISBN: 1-55860-475-8, 1999.

[FuL04] C. P. Fu, S. C. Liew, *"TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks"*, IEEE Journal on Selected Areas in Communications, vol. 21, nº 2, pp. 216-228, febrero 2004.

[GDW92] D. Gasjki, N. Dutt, A. Wu, S.Lin, *"High-Level Synthesis: Introduction to Chip and System Design"*, Kluwer Academic Publishers, 1992.

[GLT99] William Gropp, Ewing Lusk, Rajeeev Thakur, *"Using MPI-2: Advanced Features of the Message-Passing Interface"*, MIT Press, Cambridge, Massachusets, 1999.

[GMP00] T. Goff, J. Moronski, D. S. Phatak, V. Gupta, *"Freeze-TCP: a True End-to-End TCP Enhancement Mechanism for Mobile Environments"* , 9th IEEE Conference on Computer Communications (INFOCOM'00), Tel-Avic, Israel, vol. 3, pp. 1537-1545, 2000.

[HaH00] Y. Hamam, K. S. Hindi, *"Assignment of Program Modules to Processors: A Simulated Annealing Aproach"*, European Journal of Operational Research, 122(2), pp. 509-513, 2000.

[HeP96] J. L. Hennessy, D. A. Patterson, *"Computer Architecture: A Quantitative Approach"*, Morgan Kaufmann Publishers, 1996.

[Hol91] Gerard J. Holzmann, *"Design and Validation of Computer Protocols"*, Prentice Hall, ISBN: 0-13-539834-7, 1991.

[KaA00] Ad. Kamerman, Guido Aben, *"Net Throughput with IEEE 802.11 Wireless LANs"*, IEEE Wireless Communication and Networking Conference (WCNC'00), Chicago, USA, pp. 747-752, 2000.

[KaB97] Joseph M. Kahn, John R. Barry, "*Wireless Infrared Communications*", Proceedings of the IEEE, vol. 85, nº. 2, febrero 1997.

[KaK04] K. Kaemarungsi, P. Krishnamurthy, *"Properties of Indoor Received Signal Strength for WLAN Location Fingerprinting"*, 1st IEEE International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous), Boston, USA, pp. 14-23, 2004.

[KaL01] A. Kalinov, A. Lastovetsky, *"Heterogeneous Distribution of Computations*

Solving Linear Algebra Problems on Networks of Heterogeneous Computers", International Journal of Parallel and Distributed Computing, vol. 61, pp. 520-535, 2001.

[KuS02] Devdatta Kulkarni, Masha Sosonkina, *"Minimizing Communication Waiting Time in Sparse Linear System Solution using Dynamic Network Information"*, International Conference on Communications in Computing (CIC'02), Las Vegas, USA, 2002.

[LaT01] Zhiling Lan, Valerie Taylor, *"Dynamic Load Balancing of SAMR Applications on Distributed Systems"*, International Conference on SuperComputing (SC2001), Denver, USA, 2001.

[LeV00] F. Lefevre, G. Vivier, *"Understanding TCP's Behavior over Wireless Links"*, IEEE Symposium on Communications and Vehicular Technology (SCVT'00), Leuven, Bélgica, pp. 123 - 130, 2000.

[LiS01] J. Liu, S. Singh, *"ATCP: TCP for Mobile Ad Hoc Networks"*, IEEE Journal on Selected Areas in Communications, vol. 19, nº 7, pp. 1300-1315, 2001.

[LLM88] M. J. Litzkow, M. Livny, M.W. Mutka, *"Condor: A Hunter of Idle Workstations"*, 8th IEEE International Conference of Distributed Computing Systems, California, USA, pp. 104-111, 1988.

[Mac01] Elsa Mª. Macías López, *"Computación Paralela en un Cluster LAN-WLAN Controlando la Variación Dinámica de Procesos en Tiempo de Ejecución"*, Tesis Doctoral, Universidad de Las Palmas de Gran Canaria, noviembre 2001.

[Mah96] Piyush Maheshwari, *"A Dynamic Load Balancing Algorithm for a Heterogeneous Computing Environment"*, International Conference on System Sciences, Hawaii, USA, pp. 338-346, 1996.

[Mal00] Shahzad Malik, *"Dynamic Load Balancing in a Network of Workstations"*, Research Report: 95.515F, noviembre 2000.

[Mar97] Markus Fischer, *"Dynamic Load Balancing for Heterogeneous Parallel Environments"*, Master Thesis, University of Paderborn, abril 1997.

[Mar02] Evangelos P. Markatos, *"Speeding up TCP/IP: Faster Processors are not Enough"*, 21st IEEE International Performance, Computing and Communication Conference (IPCCC'02), Phoenix, USA, pp. 341-345, 2002.

[MaS01] Douglas R. Mauro, Kevin J. Schmidt, *"Essential SNMP"*, O'Reilly & Associates Inc., ISBN: 0-596-00020-0, USA, 2001.

[MCD03] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, N. Venkatasubramanian, *"Integrated Power Management for Video Streaming to Mobile Handheld*

Devices", 11th ACM International Conference on Multimedia, Berkley, USA, pp. 582-591, 2003.

[MCG01] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, R. Wang, *"TCP Westwood: End-to-End Bandwidth Estimation for Efficient Transport over Wired and Wireless Networks"*, 7th ACM Mobile Comupting and Networking, Roma, Italia, pp. 287-297, 2001.

[MoV03] S. Mohapatra, N. Venkatasubramanian, *"PARM: Power Aware Reconfigurable Middleware"*, 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), Rhode Island, USA, pp. 312-319, 2003.

[MSO01] Elsa Mª Macías, Álvaro Suárez, Carmen Ojeda, Ernesto Robayna, "*Programming Parallel Applications with LAMGAC in a LAN-WLAN Environment*", 8th European PVM/MPI, Springer-Verlag, LNCS 2131, Santorini, Grecia, pp.158-165, 2001.

[NaS98] Paolo Narváez, Kai-Yeung Siu, *"New Techniques for Regulating TCP Flow over Heterogeneous Networks"*, 23rd IEEE Conference on Local Computer Network (LCN'98), Massachusetts, USA, pp. 42-51, 1998.

[NiS93] Hiroshi Nishikawa, Peter Steenkiste, *"A General Architecture for Load Balancing in a Distributed-Memory Environment"*, 13th IEEE International Conference on Distributed Computing Systems (ICDCS'93), Pittsburgh, USA, pp. 47-54, 1993.

[PPP01] Keuntae Park, Sangho Park, Daeyeon Park, "*Enhancing TCP Performance Over Wireless Network with Variable Segment Size*", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01), Las Vegas, USA, vol. IV, pp. 398-404, 2001.

[PoL03] Edmond Poon, Baochun Li, *"SmartNode: Achieving 802.11 MAC Interoperability in Power-efficient Ad Hoc Networks with Dynamic Range Adjustments"*, 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), Rhode Island, USA, pp. 650-657, 2003.

[Raj99] B. Rajkumar, "*High Performance Cluster Computing: Architectures and Systems*", vol. 1, Prentice Hall, ISBN: 0-13-013784-7, 1999.

[SaC99] David Sánchez (autor), Juan Castellano (director), "*Herramienta de Particionado Hardware/Software para Codiseño de Sistemas*", Proyecto Final de Carrera ETSIT-ULPGC, enero 1999.

[SAM06] David Sánchez, Sergio Afonso, Elsa Mª Macías, Álvaro Suárez, *"Devices Location in 802.11 Infrastructure Networks using Triangulation"*, The 2006

IAENG International Workshop on Wireless Networks (IWWN'06), Hong Kong, pp. 938-942, 2006.

[SCS00] David Sánchez, Juan P. Castellano, Álvaro Suárez, *"Hardware/Software Partitioning based on Simulated Annealing"*, International Conference on Modelling and Simulation (MS 2000), Gran Canaria, España, pp. 115-122, 2000.

[SCS03] S. Saha, K. Chaudhuri, D. Sanghi, P. Bhagwat, *"Location Determination of a Mobile Device Using IEEE 802.11b Access Point Signals"*, IEEE Wireless Communications and Networking Conference (WCNC'03), New Orleans, USA, pp. 1987-1992, 2003.

[SKV02] A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, *"Designing Energy-Efficient Software"*, 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS'02), Florida, USA, pp. 176-183, 2002.

[SMS02] David Sánchez, Elsa Mª. Macías, Álvaro Suárez, *"An Application Level Load Balancing Mechanism for Heterogeneous Cluster Programming"*, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02), Las Vegas, USA, vol. II, pp. 872-878, 2002.

[SMS03a] David Sánchez, Elsa Mª. Macías, Álvaro Suárez, *"Load Balancing Detecting Battery Energy Level and Wireless Beacon Strength"*, 15th IASTED Internacional Conference on Parallel and Distributed Computing and Systems (PDCS'03), Marina del Rey, USA, pp. 268-273, 2003.

[SMS03b] David Sánchez, Elsa Mª. Macías, Álvaro Suárez, *"Anticipating Performance Information of Newly Portable Computers on the WLAN for Load Balancing"*, 5th International Conference on Parallel Processing and Applied Mathematics (PPAM'03), Springer-Verlag, LNCS 3019, Czestochowa, Polonia, pp. 946-953, 2003.

[SMS03c] David Sánchez, Elsa Mª. Macías, Álvaro Suárez, *"Effective Load Balancing on a LAN-WLAN Cluster"*, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03), Las Vegas, USA, vol. I, pp. 473-479, 2003.

[SMS04] David Sánchez, Elsa Mª. Macías, Álvaro Suárez, *"A Library for Load Balancing in Master/Slave Applications on a LAN-WLAN Environment"*, 12th IEEE Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'04), A Coruña, España, pp. 168-175, 2004.

[SMS05a] David Sánchez, Elsa Mª. Macías, Álvaro Suárez, *"An Improved Mechanism for Controlling Portable Computers in Limited Coverage Areas"*, 12th

European PVM/MPI, Spinger-Verlag, LNCS 3666, Sorrento, Italia, pp. 467-474, 2005.

[SMS05b] David Sánchez, Elsa Mª. Macías, Álvaro Suárez, *"Una Mejora del Framework SNMP de Equilibrio de Carga para Controlar los Computadores de la WLAN en Zonas de Cobertura Reducida"*, V Jornadas de Ingeniería Telemática (JITEL 2005), Vigo, España, pp. 415-422, 2005.

[SoC01] M. Sosonkina, G. Chen, *"Design of a Tool for Providing Network Information to Distributed Applications"*, 6th International Conference on Parallel Computing Technologies (PaCT'01), Springer-Verlag, LNCS 2127, Rusia, 2001.

[Sta97] William Stallings, *"SNMP, SNMPv2 and RMON. Practical Network Management"*, 2ª edición, Addison-Wesley, ISBN: 0-201-63479-1, USA, 1997.

[StS04] Sam Storie, Masha Sosonkina, *"Packet Probing as Network Load Detection for Scientific Applications at Run-time"*, 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe, USA, pp. 62-71, 2004.

[Sub00] Mani Subramanian, *"Network Management: Principles and Practice"*, Addison-Wesley, ISBN: 0-201-35742-9, USA, 2000.

[Sun90] V. Sunderam, *"PVM: A Framework for Parallel Distributed Computing"*, Journal of Concurrency: Practice and Experience, 2(4), pp. 315-339, 1990.

[SuG99] V. Sunderam, A. Geist, *"Heterogeneous Parallel and Distributed Computing"*, Parallel Computing, vol. 25, nº. 13-14, pp. 1699-1721, 1999.

[TaT85] A. N. Tantawi, D. Towsley, *"Optimal Static Load Balancing in Distributed Computer Systems"*, Journal of Association of Computing Machinary, vol. 32, nº 2, pp. 445-465, 1985.

[TOC03] Dave Turner, Adam Oline, Xuehua Chen, Troy Benjegerdes, *"Integrating New Capabilities into NetPIPE"*, 10th European PVM/MPI Conference, Spinger-Verlag, LNCS 2840, Venecia, Italia, pp. 37-44, 2003.

[TQD01] F. Tinetti, A. Quijano, A. De Giusti, E. Luque, *"Heterogeneous Networks of Workstations and the Parallel Matrix Multiplication"*, 8th European PVM/MPI Conference, Springer-Verlag, LNCS 2131, Santorini, Grecia, pp. 296-303, 2001.

[TXA05] Ye Tian, Kai Xu, Nirwan Ansari, *"TCP in Wireless Environments: Problems and Solutions"*, IEEE Radio Communications, pp. S27-S32, marzo 2005.

[WaS01] R. Want, B. Schilit, "*Expanding the Horizons of Location-Aware*

Computing", IEEE Computer, pp. 31-34, agosto 2001.

[WFL04] C. Wu, L. Fu, F. Lian, *"WLAN Location Determination in e-Home via Support Vector Classification"*, IEEE Conference on Networking, Sensing & Control, Taiwan, pp. 1026-1031, 2004.

[WiR93] M. H. Willebeek-LeMair, A. P. Reeves, *"Strategies for Dinamic Load Balancing on Highly Parallel Computers"*, IEEE Transactions on Parallel and Distributed Systems, vol. 4, nº 9, pp. 979-993, 1993.

[WPL02] Haitao Wu, Yong Peng, Keping Long, Shiduan Cheng, Jian Ma, *"Performance of Reliable Transport Protocol over IEEE 802.11 Wireless LAN: Analysis and Enhancement"*, 11th IEEE Conference on Computer Communications (INFOCOM'02), New York, USA, pp. 599-607, 2002.

[WSP97] R. Wolski, N. Spring, C. Peterson, *"Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service"*, UCSD Technical Report No. TR-CS97-540, 1997.

[XLW03] R. Xu, Z. Li, C. Wang, P. Ni, *"Impact of Data Compression on Energy Consumption of Wireless-Networked Handheld Devices"*, 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), Rhode Island, USA, pp. 302-311, 2003.

[XTA04] Kai Xu, Ye Tian, Nirwan Ansari, *"TCP-Jersey for Wireless IP Communications"*, IEEE Journal on Selected Areas in Communications, vol. 22, nº 4, pp. 747-756, mayo 2004.

[YAU03] M. Youssef, A. Agrawala, A. Udaya, *"WLAN Location Determination via Clustering and Probability Distributions"*, IEEE International Conference on Pervasive Computing and Communications (PerCom'03), Dallas, USA, pp. 143-150, 2003.

[YuS03] Wing Ho Yuen, Chi Wan Sung, *"On Energy Efficiency and Network Connectivity of Mobile Ad Hoc Networks"*, 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), Rhode Island, USA, pp. 38-45, 2003.

[ZGH03] Lynn Y. Zhang, Ye Ge, Jennifer Hou, *"Energy-Efficient Real-Time Scheduling in IEEE 802.11 Wireless LANs"*, 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), Rhode Island, USA, pp. 658-667, 2003.

[ZLP97] Mohammed J. Zaki, Wei Li, Srinivasan Parthasarathy, *"Customized Dynamic Load Balancing for a Network of Workstations"*, Journal of Parallel and Distributed Computing (JPDC'97), Special Issue on Workstation Clusters

and Network-based Computing, vol. 43, nº. 2, pp. 156-162, junio1997.

[ZLC95] Mohammed J. Zaki, Wei Li, Michal Cierniak, *"Performance Impact of Processor and Memory Heterogeneity in a Network of Machines"*, 4th Heterogeneous Computing Workshop (HCW'95), California, USA, pp. 101-108, 1995.

[Zom02] Albert Y. Zomaya, *"Mobile Computing: Opportunities for Parallel Algorithms Research"*, 16th IEEE Interantional Parallel and Distributed Processing Symposium (IPDPS'02), Florida, USA, pp. 144-147, 2002.

Direcciones de páginas WEB visitadas

(Estas páginas se encuentran disponibles a fecha 10 de julio de 2006)

[Web-1] http://www.bluetooth.com/
(*Autor*: Bluetooth SIG, *Tema*: Portal oficial de Bluetooth).

[Web-2] http://portal.etsi.org/bran/kta/Hiperlan/hiperlan2.asp
(*Autor*: ETSI, *Tema*: HiperLAN/2 Standard).

[Web-3] http://www.johnsoncontrols.com/Metasys/articles/article7.htm
(*Autor*: Jay S. Bayne, *Tema*: Unleashing the POWER of Networks).

[Web-4] http://www.lam-mpi.org/
(*Autor*: LAM/MPI group, *Tema*: Portal oficial de la distribución LAM/MPI).

[Web-5] http://www.net-snmp.org
(*Autor*: Net-SNMP group, *Tema*: Portal oficial de la distribución Net-SNMP).

[Web-6] http://www.iana.org
(*Autor*: The IANA organization, *Tema*: Portal oficial de Internet Authority Numbers Assigned).

[Web-7] http://www.iab.org/
(*Autor*: The IAB organization, *Tema*: Portal oficial de Internet Architecture Board).

[Web-8] http://www.linux.com/
(*Autor*: The LINUX organization, *Tema*: Portal oficial del sistema operativo LINUX).