UNIVERSIDAD DE GRANADA
Departamento de Arquitectura y Tecnología de Computadores

COMPUTACIÓN EVOLUTIVA Y PROCEDIMIENTOS DE DISTRIBUCIÓN DE CARGA EN CLUSTERS DE COMPUTADORES

TESIS DOCTORAL

Realizada por:

Mohammed M. M. Aldasht

Dirigida Por:
Dr. D. Julio Ortega
Dr. D. Carlos G. Puntonet

Granada, Julio 2004

UNIVERSIDAD DE GRANADA
Departamento de Arquitectura y Tecnología de Computadores

COMPUTACIÓN EVOLUTIVA Y PROCEDIMIENTOS DE DISTRIBUCIÓN DE CARGA EN CLUSTERS DE COMPUTADORES

Memoria presentada por

Mohammed M. M. Aldasht

Para optar al grado de
DOCTOR EN INGENIERÍA INFORMÁTICA

Fdo. Mohammed M. M. Aldasht

D. Julio Ortega Lopera, Catedrático de Universidad, y **D. Carlos García Puntonet**, Profesor Titular de Universidad, ambos del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada,

CERTIFICAN:

Que la memoria titulada **"Computación Evolutiva y Procedimientos de Distribución de Carga en Clusters de Computadores"**, ha sido realizada por **D. Mohammed M. M. Aldasht** bajo nuestra dirección en el departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

Granada, a 23 de Julio de 2004

Fdo. Dr. Julio Ortega Lopera
Director de la Tesis

Fdo. Dr. Carlos García Puntonet
Director de la Tesis

بسم الله الرحمن الرحيم

"وقل إعملوا فسيرى الله عملكم ورسوله والمؤمنون"

صدق الله العظيم.

شكر و تقدير

(*Agradecimientos*)

في البداية أود أن أتوجّه بجزيل الشكر و العرفان لأمي وأبي للتضحيات والعطف والحنان المتواصل تجاهي و تجاه أخوتي و التي بدونها لكان من المستحيل إتمام هذا العمل. كما أتوجه بالشكر إلى إخواني و أخواتي على عطفهم ومساندتهم الدائمة.

شكر خاص إلى خطيبتي آيات التي تزيدني سعادة في كل يوم وتُشعرني أنها دائماً إلى جانبي.

كما يتوجّب التوجه بالشكر الجزيل إلى مشرفيْ هذا البحث الأستاذ الدكتور خوليو أورتيجا والدكتور كارلوس غ. بونتونيت لجهودهما و تعاونهما العظيمين لأتمام هذا العمل.

إلى جميع الأساتذه في قسم عمارة وتكنولوجيا الحاسوب بجامعة غرناطه أتوجه إليهم بالشكر لما أبدوه من تعاون و دعم مستمرّين. أخص بالذكر الأستاذ الدكتور ألبيرتو بريبيتو,د. أنطونيو دياث,د. خافيير فرناندث,د. أنطونيو كانياس,د. خيسوس غونثالث و د. هكتور بومارس.

كما أتوجه بالشكر و العرفان إلى كل معلم و مدرّس تتلمذت على يديه. لأنهم أعطوني الكثير من أسباب حب العلم و السعي في طلبه.

أتوجه بالشكر الجزيل إلى جميع أصدقائي على مساعدتهم ودعمهم المتواصلين. وأخص بالذكر الأخ و الصديق فايز أبو عمرية و الأصدقاء فرانثيسكو تشيكا و بدرو دانييل بونيًا.

كما أود توجيه جزيل الشكر إلى الوكالة الإسبانية للتعاون الدولي لمنحي فرصة الإلتحاق ببرنامج الدكتوراه عبر المساعدة المادية طوال ثلاث سنوات. كما و أشكر جامعة بوليتكنك فلسطين و جامعة غرناطة على تقديم المساعدة لي في السنة الرابعة من دراستي في برنامج الدكتوراه.

لكل هؤلاء ..... جزيل الشكر و الإمتنان

Agradecimientos

Ante todo quiero agradecer a mis padres su esfuerzo y cariño, sin ellos hubiese sido imposible realizar este proyecto. A mis hermanos por su continuo apoyo y afecto.

Con especial cariño a mi novia Ayat, que me ayuda a ser más feliz cada día y me hace sentir que está en todo momento junto a mí. Espero verla pronto.

A mis directores de tesis, Prof. Dr. D. Julio Ortega y Prof. Dr. D. Carlos G. Puntonet, por su enorme esfuerzo y colaboración en este trabajo, y por su ayuda en todo lo que he necesitado.

A todos los profesores del departamento, por su continuo apoyo y colaboración. Cabe destacar más directamente a los profesores D. Alberto Prieto, D. Antonio Díaz, D. Javier Fernández, D. Antonio Cañas, D. Jesús González y D. Héctor Pomares.

A todos los profesores que he tenido, porque ellos me han transmitido el interés por la ciencia y han dedicado parte de su tiempo en mi educación.

A todos mis amigos, por su apoyo y ayuda en todo momento y en todo lo que he necesitado. En especial, a mi amigo Faeyz Abu-Amria, a Francisco Chica Navas y a Pedro Daniel Díez Bonilla.

A la Agencia Española de Cooperación Internacional AECI, por darme la oportunidad de iniciar los estudios de doctorado a través su ayuda económica durante tres años.

A la Universidad Politécnica de Palestina, al departamento de arquitectura y tecnología de computadores y a la Universidad de Granada por su ayuda económica en el cuarto año de estancia.

A todos ellos… Muchas Gracias

A mis padres

A mi novia

ÍNDICE

Prólogo

La convergencia entre la tecnología de las redes de área local y los sistemas de interconexión utilizados en los procesadores masivamente paralelos, ha ocasionado una cierta evolución hacia arquitecturas de redes como Gigabit Ethernet, ATM, Myrinet, Infinband, etc. [FER99], con anchos de banda de los gigabits o gigabytes por segundo, y prestaciones cada vez más próximas a las de las redes de comunicación en multicomputadores. Por otra parte, las mejoras tecnológicas permiten disponer de microprocesadores, buses, memorias, etc. para configurar nodos de cómputo con una elevada relación prestaciones/coste. Así, en los últimos años están siendo objeto de un enorme interés los *clusters de computadores* [AND95,BUY99,PFI98], arquitecturas paralelas basadas en computadores personales o estaciones de trabajo basadas en hardware disponible comercialmente, conectados mediante redes de comunicación como Gigabit Ethernet, ATM, ServerNet, SCI, Autonet, Memory Channel, Synfinity, HiPPI, Myrinet, Infiniband, etc. [FER99,WEI00]. Estos sistemas ofrecen la posibilidad de aprovechar el *procesamiento paralelo* para conseguir una potencia de cómputo elevada a un costo mucho menor que otras arquitecturas multiprocesador al utilizar elementos hardware *estándar*, para los que existe un amplio mercado, con volúmenes de ventas significativos, y donde es más fácil repercutir los costes de investigación y desarrollo necesarios para producir elementos con mejores prestaciones. Es evidente que, para que el

paralelismo constituya una herramienta ampliamente utilizada en la práctica, es preciso que su utilidad quede patente en este tipo de sistemas, que están al alcance de prácticamente todas las empresas, centros de investigación y desarrollo, etc. Además, si es posible resolver una aplicación computacionalmente costosa en un sistema del que ya se dispone o que resulta relativamente barato de adquirir o configurar, el desarrollo de software paralelo es todavía más importante.

Por tanto, el uso de clusters y el desarrollo de procedimientos paralelos eficaces para aprovechar su capacidad tiene un enorme interés, puesto que permitiría ampliar el conjunto de aplicaciones abordables, no sólo técnicamente, sino también de forma económicamente rentable. En este mismo sentido, una dirección de desarrollo de la línea de investigación que aquí se describe la constituye la metacomputación (*metacomputing* o *grid computing* en inglés). Precisamente el desarrollo de Internet y la disponibilidad de computadores y redes de altas prestaciones y bajo coste han contribuido a generar este nuevo paradigma. Si el ancho de banda de comunicación crece, la ubicación de la potencia de cálculo pierde importancia dado que es posible configurar plataformas de altas prestaciones a partir de redes de computadores distribuidos geográficamente. La imagen que se impone es la de una red de cómputo con características de ubicuidad, disponibilidad, y transparencia, similares a las de la red eléctrica, que permitiría alcanzar plenamente la sociedad de la información. No sólo las aplicaciones que requieren altas prestaciones de cómputo, típicas en ciencia, ingeniería, y negocios, aprovecharían las posibilidades de esta tecnología sino también la exploración de datos, el trabajo cooperativo, y todas las aplicaciones que requieren alto rendimiento y fiabilidad. Además, desde el punto de vista de la investigación en el área de arquitectura de computadores, la implementación de una estructura de metacomputación implica abordar problemas de gran interés relacionados con la heterogeneidad, la escalabilidad, y la adaptabilidad y fiabilidad de las plataformas, y el desarrollo de una capa intermedia (*middleware*) que permita el acceso transparente a los recursos utilizados.

En la décimocuarta edición (año 2000) del IPDPS (*International Parallel and Distributed Processing Symposium*) se celebró una mesa redonda con el siguiente título: ¿Cuáles son los diez conceptos más influyentes del último milenio en el ámbito del procesamiento paralelo y distribuido?. Las conclusiones de esa mesa redonda se recogieron posteriormente en el

artículo [THE01]. Uno de esos diez conceptos, entre los que están la ley de Amdahl, la segmentación de cauce, el procesamiento multihebra, y la sincronización, es, precisamente, el procesamiento en clusters de computadores. Otro concepto es la distribución equilibrada de la carga (*load balancing*), en cuyo ámbito se sitúan las cuestiones que se abordan en esta tesis.

La distribución equilibrada de la carga tiene como objetivo repartir el trabajo a realizar (la carga) entre los distintos nodos de procesamiento de la plataforma paralela y/o distribuida que se utiliza, de forma que se maximice la utilización de los recursos (distribución equilibrada) para optimizar el rendimiento de la plataforma. Los procedimientos de distribución de carga deben encontrar un compromiso (*tradeoff*) entre la utilización de los procesadores que constituyen la plataforma y los costes asociados a la comunicación y sincronización entre esos procesadores, de manera que, al final, se minimice el tiempo de ejecución de la aplicación considerada.

La distribución de carga puede ser *estática* o *dinámica*. La distribución de carga estática contempla la aplicación a procesar como un conjunto de tareas de las que se conocen sus tamaños y sus dependencias. De esta forma, se puede prefijar una asignación de tareas a procesadores en la que, una vez que una tarea se asigna a un procesador, se ejecuta en dicho procesador hasta que termina. La determinación de la asignación óptima de tareas que verifique las características deseadas de mínimo tiempo de respuesta, mínimo coste de comunicación, y rendimiento máximo es un problema NP-completo para el que se han propuesto diversos procedimientos como los basados en la teoría de grafos, o en la teoría de colas, o en metaheurísticas como los algoritmos genéticos, el enfriamiento simulado, etc.

La distribución de carga dinámica aprovecha los recursos de comunicación de la plataforma paralela para intercambiar información de estado y tareas entre los procesadores y mejorar así las prestaciones del sistema. Por tanto, la distribución de carga dinámica puede considerarse un proceso en el que, en general, los procesadores utilizan la información local de que disponen acerca del estado global del sistema para tomar las decisiones correspondientes de cara a alcanzar los objetivos de tiempo mínimo de respuesta y rendimiento máximo. La eficiencia del procedimiento de distribución de carga depende, por tanto, del coste de comunicación entre procesadores, de la complejidad asociada al procedimiento de toma de decisiones en cada procesador, y del coste de mantener

información relativa al estado global del sistema en cada uno de los nodos. Las estrategias de distribución de carga dinámica pueden implementarse según un modelo *centralizado* o *distribuido*. En la distribución de carga dinámica centralizada, un procesador se encarga de mantener información del estado global del sistema y, a partir de dicha información, lleva a cabo la asignación de tareas a procesadores. En el caso de sistemas con un número elevado de procesadores distribuidos geográficamente, esta estrategia resulta inviable en la práctica, por lo que las estrategias distribuidas son las más adecuadas. En estas estrategias distribuidas, los nodos del sistema disponen de información local (más o menos actualizada) acerca del estado global del sistema y toman las decisiones de forma autónoma a partir de ese estado local y de la información que intercambian con otros procesadores. Dado que los procesadores utilizan una información parcial del estado del sistema, las decisiones que se toman no suelen corresponder a la decisión óptima.

El trabajo que se describe en esta memoria se ha centrado en los procedimientos de distribución o equilibrado de carga dinámicos. En la literatura se describen numerosos ejemplos de este tipo de procedimientos cuya eficacia depende tanto del tipo de plataforma sobre la que se aplican, como de las características del problema que se aborda mediante el correspondiente programa paralelo. Todos esos procedimientos conforman un espacio de diseño considerablemente complejo. Consiguientemente, la toma de decisiones acerca del tipo de procedimiento a elegir, según el problema a abordar y la plataforma, resulta bastante complicada.

En esta memoria estudiamos las posibilidades de la computación evolutiva, y más concretamente los algoritmos genéticos, en el ámbito de la distribución dinámica de carga. Por un lado se considera su uso a la hora de encontrar distribuciones de carga óptimas, y por otro se plantea la posibilidad de optimizar los parámetros de los procedimientos de distribución dinámica de carga mediante el uso de algoritmos genéticos, gracias a las posibilidades que ofrecen para explorar el espacio de diseño de los procedimientos de distribución de carga. Para poder llevar a cabo dicha exploración es necesario disponer de una taxonomía capaz, tanto de incorporar los procedimientos que se han propuesto hasta el momento, como de sugerir posibilidades no consideradas hasta el momento en la definición de nuevos procedimientos de distribución de carga. A partir de esa taxonomía es posible

realizar una codificación de los procedimientos de distribución de carga que permite la exploración del espacio de diseño definido. Así, mediante el uso de los algoritmos genéticos se podrá dar respuesta a cuestiones relevantes en el ámbito de los problemas de distribución de carga. Por ejemplo, se podrá comprobar si existe algún procedimiento que sea mejor que los restantes independientemente del tipo de plataforma o de la aplicación paralela, o bien las relaciones entre las características de las aplicaciones, o los datos sobre los que actúan, y los parámetros óptimos de las procedimientos de distribución de carga más adecuados.

Aunque aquí se considerarán determinados ejemplos de aplicaciones que nos han servido para definir nuestro conjunto de benchmarks, y se utiliza un cluster de computadores como plataforma para realizar nuestros experimentos, la metodología que se describe en esta memoria puede aplicarse a conjuntos de benchmarks más extensos y a más tipos de plataformas. En cualquier caso, a través de esta memoria se pone de manifiesto la utilidad de la computación evolutiva para explorar espacios complejos como los involucrados en las tareas de diseño propias de la Arquitectura de Computadores. Teniendo esto en cuenta, la memoria se ha organizado a través de los siguientes capítulos:

- En el Capítulo 1, titulado "*Procesamiento Paralelo y Distribución de Carga*" se describe el proceso de generación de un programa paralelo y se introduce el problema de la distribución de carga en ese ámbito. También se presentan las métricas (el tiempo de ejecución, la eficiencia, la ganancia de velocidad "*speedup*" y el coste) que permiten evaluar las prestaciones de un programa paralelo y sus expresiones, para ilustrar cuantitativamente la incidencia de la distribución de carga. Para concluir el capítulo se describen las características y los tipos principales de procedimientos de distribución dinámica de carga, y se detalla un modelo de evaluación de prestaciones que ayude en el diseño óptimo de los procedimientos de distribución de carga. Precisamente, la complejidad y las limitaciones de los modelos de evaluación de prestaciones que se han propuesto contituyen argumentos importantes en favor de explorar las posibilidades que, en este contexto de la distribución dinámica de la carga, ofrece la computación evolutiva.
- En el Capítulo 2, titulado "*Equilibrado de carga dinámico y centralizado*" se consideran las posibilidades de los algoritmos genéticos en los procedimientos de

distribución dinámica y centralizada de carga. Así se describe un procedimiento centralizado basado en un algoritmo genético que se ha propuesto en [ZOM01].En este capítulo se evalúan las prestaciones de este procedimiento y se comparan con las de un nuevo procedimiento que proponemos. Se trata de un procedimiento adaptativo y centralizado basado en la estrategia de ofertas, que es una de las dos estrategias más comúmente utilizadas en este ámbito.

- En el Capítulo 3, titulado "*Una taxonomía para las estrategias distribuidas de equilibrado dinámico de carga*" se proporciona una taxonomía que contiene varios procedimientos de distribución de carga. Esta taxonomía facilita la parametrización de los diferentes procedimientos de distribución de carga para luego poder aplicar la búsqueda genética en el espacio definido por la taxonomía. De esta manera se pueden optimizar los parámetros que determinan el comportamiento de los diferentes procedimientos de distribución de carga. También se ha diseñado un procedimiento general de equilibrado dinámico y distribuido de carga que puede incorporarse en las aplicaciones paralelas y queda especificado a partir de los parámetros que, en la taxonomía que proponemos, caracterizan a cada procedimiento de equilibrado de carga.
- En el Capítulo 4, titulado "*Exploración de las estrategias distribuidas de equilibrado de carga*", se desarrolla un algoritmo genético para la exploración del espacio de diselo definido por la taxonomía presentada en el Capítulo 3. y se analizan los resultados obtenidos por el algoritmo genético.
- Las conclusiones y principales aportaciones del trabajo que se describe en esta memoria, así como las perspectivas que se plantean para continuar nuestra investigación en este campo, se recogen en el capítulo 5.

Capítulo 1

Procesamiento Paralelo y Distribución de Carga

En este capítulo se describen las etapas del proceso de elaboración de un programa paralelo. A partir de esa descripción se introducirá el problema de la distribución de carga en cuyo ámbito se ha desarrollado el trabajo de investigación descrito en esta memoria. La Sección 1.1 se completa con la presentación de los diferentes paradigmas de programación paralela. En la Sección 1.2 se aborda el problema de la evaluación de prestaciones en las aplicaciones paralelas. Así, se describen las métricas comúnmente utilizadas para medir el rendimiento de los programas paralelos, es decir la ganancia de velocidad (*speedup*) y la eficiencia, y se analiza el concepto de escalabilidad, presentando el concepto de función de isoeficiencia como métrica para cuantificar dicha escalabilidad. La Sección 1.3 se dedica a precisar el problema de la distribución de carga en plataformas paralelas. Se describen las distintas alternativas para abordar este problema, poniendo de manifiesto las dificultades que plantea el diseño de un procedimiento eficiente de distribución de carga y el modelado del mismo, y haciendo referencia a los trabajos más significativos que se han propuesto en la literatura. Para terminar el capítulo, se describen los objetivos del trabajo de investigación que se ha realizado en el contexto de la problemática descrita para la distribución de carga, y se ubican estos objetivos en los distintos capítulos que constituyen esta memoria.

1.1 Diseño de una Aplicación Paralela

En [CUL99] se describen las etapas a llevar a cabo para paralelizar un algoritmo secuencial. Asimismo, se precisa la relación entre tareas y procesos que, por otra parte, se va a utilizar a lo largo de esta memoria. El conocimiento de las funciones que se realizan en cada una de las etapas del proceso de paralelización y la problemática que se plantea en cada una de ellas es fundamental para realizar una paralelización eficiente del algoritmo secuencial del que se parte y, además, es esencial de cara a evaluar el comportamiento de las distintas arquitecturas paralelas frente al programa paralelo resultante. Por otra parte, en el contexto del trabajo de investigación que aquí se describe, la descripción de las etapas del proceso de paralelización nos permitirá precisar el problema que hemos abordado al situarlo en el ámbito en el que se plantea. A fin de cuentas, la paralelización de un problema, para cuya resolución se dispone de un algoritmo secuencial codificado a través del correspondiente programa (programa secuencial), implica identificar el trabajo que puede realizarse en paralelo, y la forma de distribuir ese trabajo (y los datos sobre los que se trabaja) entre los procesadores, gestionando convenientemente el acceso a los datos, la comunicación, y la sincronización entre los procesadores. Por lo tanto, se trata de organizar la computación, el acceso a los datos y la actividad de entrada/salida en los distintos elementos de la plataforma de cómputo paralela, de forma que se obtenga una ganancia de velocidad suficientemente buena con respecto al mejor programa secuencial que resuelva el mismo problema. Para conseguir este objetivo es necesario garantizar una distribución equilibrada del trabajo entre los procesadores, reducir las necesidades de comunicación entre procesadores, y minimizar la sobrecarga (*overhead*) asociada a la gestión del paralelismo, la sincronización y la comunicación entre procesos.

En la Figura 1.1 se muestran las etapas a seguir para crear un programa paralelo. Estas etapas pueden ser realizadas bien por el programador, bien por algunas de las capas de software que existen en el sistema entre el programador y la arquitectura: el compilador, el sistema de ejecución (*run-time system*), y el sistema operativo. Lo ideal sería que el programador pudiera escribir el programa en la forma que estime más conveniente y que automáticamente se produjeran las transformaciones pertinentes para que el programa se ejecutase de forma eficiente en la plataforma paralela correspondiente. A pesar de la actividad investigadora que se está desarrollando en este ámbito de los compiladores que paralelizan

automáticamente y en los lenguajes de programación paralelos, la mayor parte del proceso de paralelización es responsabilidad del programador, con cierta ayuda del compilador y del sistema de ejecución.

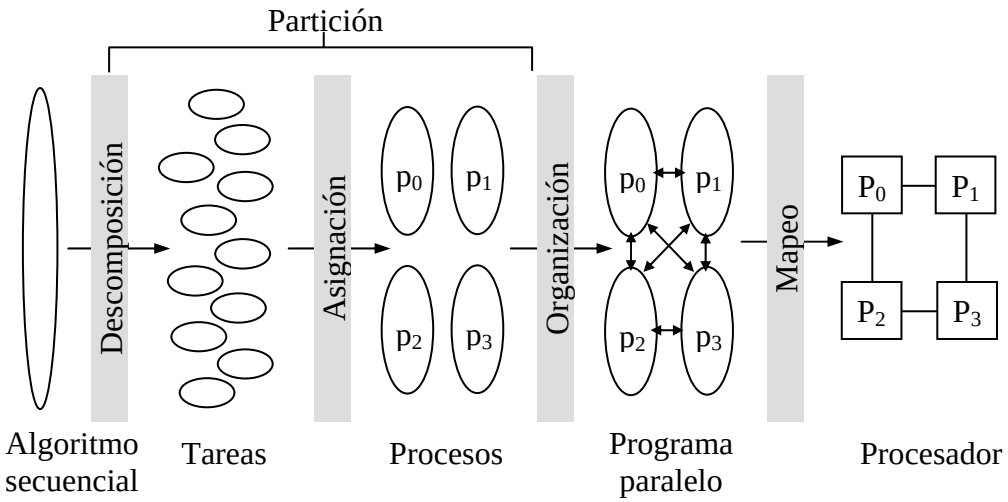


Figura 1.1. Etapas del proceso de paralelización de un programa [CUL99]

A continuación se describen las funciones a realizar en cada una de las etapas de la paralelización de un programa: *descomposición*, *asignación*, *organización*, y *mapeo*.

1. ***Descomposición en tareas.*** El trabajo a realizar se divide en una serie de *tareas* (en inglés *tasks*). Una *tarea* es una parte, definida arbitrariamente, del trabajo que implementa el programa. Se trata de la unidad más pequeña que puede ejecutarse concurrentemente con otras en un programa paralelo. Por tanto, una tarea se ejecuta en un procesador, y los procesadores de la plataforma ejecutan tareas (una tarea en cada procesador) concurrentemente. El tamaño o el contenido de las tareas no está predeterminado por el programa secuencial que se paraleliza, sino que es una decisión de quien realiza la paralelización. Si una tarea realiza una cantidad pequeña de trabajo e habla de una tarea de *granularidad fina*, y si en cambio incluye un mayor volumen de trabajo se dice que es una tarea de *granularidad gruesa*. Obviamente, estos términos cualitativos referentes a la granularidad, fina o gruesa, de una tarea son términos relativos. Lo *procesos*, o también *hebras*, son entidades abstractas que realizan tareas, de

forma que un programa paralelo estará constituido por varios procesos que cooperan, de forma que cada uno de ellos completa un subconjunto de las tareas correspondientes al trabajo a realizar. Si el volumen de trabajo que realiza el programa secuencial se puede determinar con precisión a partir del propio programa es posible distribuir ese trabajo entre las distintas tareas, de forma más o menos precisa, en el proceso de paralelización. No obstante, en muchas aplicaciones, el trabajo a completar puede variar en el momento en que el programa paralelo se está ejecutando. En este caso, el número de tareas disponibles y/o su volumen de trabajo pueden cambiar dinámicamente cuando el programa se ejecuta. El número máximo de tareas disponibles para la ejecución a la vez proporciona un límite superior para el número de procesos, y por consiguiente, para el número de procesadores que pueden usarse eficazmente. Por lo tanto, el objetivo principal en la descomposición es extraer un grado de concurrencia suficiente como para mantener ocupados a los procesadores durante todo el tiempo, garantizando que la sobrecarga asociada a la gestión de las tareas sea suficientemente baja en comparación con el trabajo útil a realizar. Hay que tener en cuenta que si la concurrencia de que se dispone es limitada, la ganancia de velocidad que puede alcanzarse se verá también limitada, como veremos más adelante utilizando la ley de Amdahl. Precisamente, la concurrencia de que se dispondrá es la que se haya extraído en esta etapa de descomposición.

2. ***Asignación de tareas a procesos.*** En esta etapa se especifica la forma de distribuir las tareas entre los procesos. Los objetivos principales de esta etapa son la distribución equilibrada de la carga de trabajo entre los procesos, reducir la comunicación entre dichos procesos, y minimizar la sobrecarga (*overhead*) asociada al propio proceso de asignación. La carga de trabajo a distribuir no sólo incluye las operaciones o cálculos a realizar, sino también la comunicación y el acceso a los datos, y las operaciones de entrada/salida. La comunicación entre procesos conlleva un coste importante desde el punto de vista del tiempo de ejecución, sobre todo si los procesos que deben comunicarse se asignan luego a procesadores diferentes. La asignación de tareas a procesos puede ser *estática* o *dinámica*. Es estática si puede determinarse completamente en el momento en que se lleva a cabo el proceso de paralelización, en la compilación, o justo al principio de la ejecución del programa tras leer y analizar las

entradas del programa, y no cambia durante la ejecución del programa. Si, en cambio, la asignación se va realizando a medida que el programa se ejecuta, la asignación es dinámica.

3. ***Organización del acceso a los datos, la comunicación, y la sincronización entre procesos.*** Para poder ejecutar las tareas que se les han asignado en la etapa anterior, los procesos necesitan mecanismos para identificar y acceder a los datos, y para intercambiar datos con otros procesos, y sincronizarse con ellos. En esta fase se determinan aspectos como la forma de organizar las estructuras de datos, la manera de planificar las tareas en el tiempo en cada procesador para aprovechar la localidad en el acceso a los datos, si la comunicación se realiza implícita o explícitamente, y a través de mensajes largos o cortos, y el modo de expresar la comunicación y la sincronización necesaria para la asignación de tareas que se ha llevado a cabo. Por tanto, en esta etapa intervienen de forma esencial la arquitectura, el modelo de programación (más adelante nos referiremos a los paradigmas de programación paralela), y por supuesto, el propio lenguaje de programación, ya que es esta etapa en la que se escribe el programa haciendo uso de los mecanismos que proporciona el lenguaje para expresar la comunicación, y la sincronización. La disponibilidad de determinados mecanismos en el lenguaje de programación, y el soporte que la arquitectura proporciona para dichos mecanismos, determinan en gran medida los compromisos a realizar en esta etapa. Precisamente, los principales objetivos a alcanzar en esta etapa se refieren a la reducción del coste de comunicación y sincronización entre procesadores, y para ello se busca mantener la localidad en el acceso a los datos, planificar las tareas de forma que todas aquellas tareas de las que dependa una tarea dada se completen antes de que le toque el turno a ésta, y reduciendo el coste asociado a la gestión del paralelismo (creación de tareas, etc.).

4. ***Asignación de procesos a procesadores*** (*mapeo*). Como resultado de las etapas previas de descomposición, asignación y organización se obtiene un conjunto de procesos que cooperan entre sí y constituyen el programa paralelo. La siguiente etapa consiste en asignar los procesos del programa a los procesadores de la plataforma. Esta función puede implementarse en el propio programa, pero también puede ser el sistema operativo el que puede controlar la ejecución de los procesos en los procesadores

disponibles. Una alternativa para llevar a cabo la asignación de procesos a procesadores, quizá la situación más sencilla, consiste en dividir los procesadores de la plataforma en subconjuntos, de forma que cada programa se ejecuta utilizando los procesadores de uno de los subconjuntos definidos. En este caso se habla de distribución del espacio de la plataforma paralela (*space-sharing*). Los programas pueden asignar procesos a procesadores específicos asegurándose que dichos procesos no se moverán (migrarán) de un procesador a otro, incluso se puede tener en cuenta qué proceso se ejecuta en cada procesador para mantener la localidad de acceso a los datos o minimizar el coste de comunicación según la topología de la red de interconexión entre procesadores. La otra alternativa, deja que sea el sistema operativo el que determine dinámicamente, y sin la intervención del usuario, cuándo se ejecuta un proceso y en qué procesador, para conseguir una mejor utilización de los recursos disponibles en el computador. En la mayoría de los sistemas actuales se produce una situación intermedia en la que el programa tiene un cierto control de la asignación de procesadores al permitir que el usuario pueda seleccionar o preservar ciertas propiedades, pero se deja que sea el sistema operativo pueda cambiar la asignación dinámicamente de manera que sea posible una gestión eficiente de los recursos.

Las dos primeras etapas, correspondientes a la *descomposición* en tareas del trabajo y a la *asignación* de tareas a procesos, se pueden contemplar conjuntamente como una etapa de *partición*. Como se ha visto, esas dos etapas pueden considerarse independientes de la arquitectura y del modelo de programación, aunque en algunos casos el coste y la complejidad de la utilización de determinadas primitivas pueden afectar a las decisiones que deben tomarse en esta fase de partición y, en esos casos deben revisarse la partición realizada.

La secuencia de etapas para el proceso de paralelización que se ha descrito, se puede encontrar en [CUL99]. No obstante existen otras aproximaciones diferentes a la hora de distribuir las funciones que deben realizarse. Así, otra forma de contemplar el proceso de paralelización de un programa se describe en [FOS95], donde también se hace una división en cuatro etapas del diseño de un programa paralelo, aunque las funciones y la denominación de las etapas difiere de la que se presenta en [CUL99]. A continuación describimos brevemente las etapas indicadas en [FOS95]:

1. ***Partición.*** Descomposición de las operaciones de cómputo y de los datos que se utilizan en dichas operaciones. La descomposición de los datos del problema se denomina *descomposición de dominio*, mientras la descomposición de las funciones en tareas separadas es la *descomposición funcional*.
2. ***Comunicación.*** Se considera el intercambio de información y la coordinación entre las tareas creadas en la etapa de partición anterior. La comunicación que se debe llevar a cabo en un programa paralelo dependerá de las características del problema a resolver y de la forma en la que se haya realizado la descomposición.
3. ***Agrupación.*** En esta etapa se evalúan las estructuras de tareas y comunicaciones que se han definido en las dos etapas previas, considerando las restricciones existentes en cuanto al funcionamiento y al coste de la implementación. Mediante la agrupación de tareas en procesos se puede reducir el coste de comunicación, se procura que el programa sea más escalable, y se dispone de una cierta flexibilidad de cara a la planificación.
4. ***Planificación.*** En esta última etapa se asignan las tareas a los procesadores buscando maximizar la utilización de los recursos del sistema y minimizar el coste de comunicación.

Comparando las etapas descritas en [CUL99] y las de [FOS95] se tiene que, las dos primeras etapas que se consideran en [CUL99] incluyen las funciones que se distribuyen entre las tres primeras etapas de [FOS95]. Es decir, la fase de *partición* en [CUL99] se podría asimilar a las etapas de *partición*, *comunicación* y *agrupación* de [FOS95]. Igualmente, la etapa de *asignación de procesos a procesadores* de [CUL99] puede identificarse con la etapa de *planificación* de [FOS95]. Así, en la distribución de etapas de [CUL99] se incluye una etapa (la etapa de *organización*) que no se considera en [FOS95], o simplemente no se identifica como una función distinta de las que se llevan a cabo en la etapa denominada etapa de comunicación en [FOS95]. A través de la etapa de organización que se describe en [CUL99] se hace hincapié en la necesidad de tener en cuenta los recursos que proporciona la arquitectura en cuanto a comunicación, sincronización y ubicación de los recursos de memoria, así como las primitivas de que dispone el lenguaje de programación, y las

características del modelo de programación. Estos aspectos no se tienen en cuenta tan explícitamente en la descripción de etapas presentada en [FOS95].

Como se ha indicado, el modelo o el *paradigma de programación* intervienen en la etapa de *organización del acceso a los datos, la comunicación, y la sincronización entre procesos* que se lleva a cabo tras la fase de partición en la descripción de [CUL99]. A continuación se describen los tres paradigmas o modelos de programación paralela existentes.

1.1.1 Modelos de Programación Paralela

El modelo de programación es la imagen que tiene el programador de la plataforma paralela para la que está elaborando el programa. Como se ha indicado más arriba, un programa paralelo está constituido por una serie de procesos o hebras que actúan sobre unos datos. Ese programa se escribe utilizando un modelo de programación que, como todo modelo de programación, debe especificar los datos a los que las hebras pueden hacer *referencia*, las *operaciones* que pueden hacerse con esos datos, y qué *orden* debe mantener se entre esas operaciones. De esta forma, un modelo de programación paralelo incluirá la forma según la cual las distintas hebras comunican entre sí la información (dado que se debe especificar los datos la los que las hebras pueden hacer referencia), y las operaciones de sincronización de que se dispone para coordinar su trabajo (dado que se debe especificar el orden que debe mantenerse entre las operaciones).

Básicamente, existen dos modelos de programación paralela: modelo de *espacio de memoria compartido*, y modelo de *paso de mensajes*. No obstante, en algunos textos se añade además el modelo de *paralelismo de datos*. Antes de pasar a describir las características de estos modelos vamos a definir los términos SPMD (*Single-Program Multiple Data*) y MPMD (*Multiple-Program Multiple-Data*), que hacen referencia a la similitud entre los procesos que constituyen un programa paralelo. Así, en un programa SPMD todos los procesos ejecutan el mismo código aunque utilizan conjuntos de datos diferentes. En cambio, en un programa MPMD los procesos pueden ejecutar códigos diferentes y, usualmente, utilizarán conjuntos de datos diferentes.

En un programa SPMD, los datos deben distribuirse de forma que manera que cada proceso se ocupe de un área de datos diferente, y los procesos se comuniquen entre ellos

cuando necesiten sincronizarse o comunicar datos entre sí. El coste de comunicación depende del número de procesos que pueden considerarse vecinos en el sentido de que utilizan datos o resultados generados por uno de esos procesos a partir del área de datos que le corresponde. Mientras, el coste de computación está relacionado, fundamentalmente, con el tamaño del área de datos que le corresponda. Este tipo de programas puede resultar muy eficaz en un sistema homogéneo en el que sea relativamente sencillo y poco costoso realizar una distribución adecuada de los datos. En algunos problemas es difícil realizar una partición estática de los datos que permita una distribución equilibrada de la carga. Es el caso de problemas en los que las estructuras de datos sean irregulares o los costes computacionales varíen de forma poco previsible según los datos que se procesan es necesario utilizar procedimientos dinámicos para la distribución de los datos entre los distintos procesos. Un problema que plantean los programas SPMD reside en su sensibilidad ante posibles fallos en la ejecución de un proceso. En ese caso, el programa paralelo no podría terminar.

En cuanto a los programas MPMD, existen algunas formas características de organizar los distintos procesos que constituyen el programa. Entre estas alternativas está la organización maestro-trabajador, la organización de *segmentación de cauce* y la organización *divide-y-vencerás*.

La *organización maestro/trabajador* utiliza dos tipos de procesos: el proceso maestro (master) y el proceso trabajador (worker). En un programa paralelo habrá un proceso maestro y varios procesos trabajadores. El proceso maestro lleva a cabo tres operaciones. En primer lugar descompone el problema en tareas, después distribuye las tareas entre los trabajadores y, por último, recoge los resultados, a partir de los cuales compone (si fuera necesario) el resultado final del programa. Por otra parte, cada uno de los procesos trabajadores realiza otras tres operaciones. En primer lugar recibirá un mensaje del maestro a partir del cual puede determinar la tarea o tareas que debe completar, después se encargará de procesar esas tareas y, finalmente, enviará el resultado al maestro. Un programa maestro/trabajador en el que todos los procesos trabajadores fueran iguales y cada uno procesase un conjunto de datos diferente prácticamente podría considerarse un programa SPMD (salvo el proceso maestro, todos los procesos son iguales). El funcionamiento de estos programas se esquematiza en la Figura 1.2. En un programa maestro/trabajador se puede utilizar una distribución estática o dinámica de las tareas. En la distribución estática el maestro establece la distribución de tareas

antes de comenzar el procesamiento de éstas y, por lo tanto, también puede participar en la computación como un trabajador más. Cuando el número de tareas es muy grande o no se conoce de antemano, o en el caso de que las tareas tengan un tiempo de ejecución desconocido a priori, no es eficiente realizar una distribución estática del trabajo a realizar. En ese caso el trabajo debe distribuirse dinámicamente, según vaya avanzando el procesamiento que están realizando los trabajadores. El maestro debe encargarse de ir asignando las tareas que queden o las nuevas tareas que vayan surgiendo a los trabajadores ociosos. En los programas maestro/trabajador, a medida que aumenta el número de trabajadores, el maestro se va convirtiendo en un cuello de botella cada vez mayor, por lo que la escalabilidad de este paradigma es limitada.

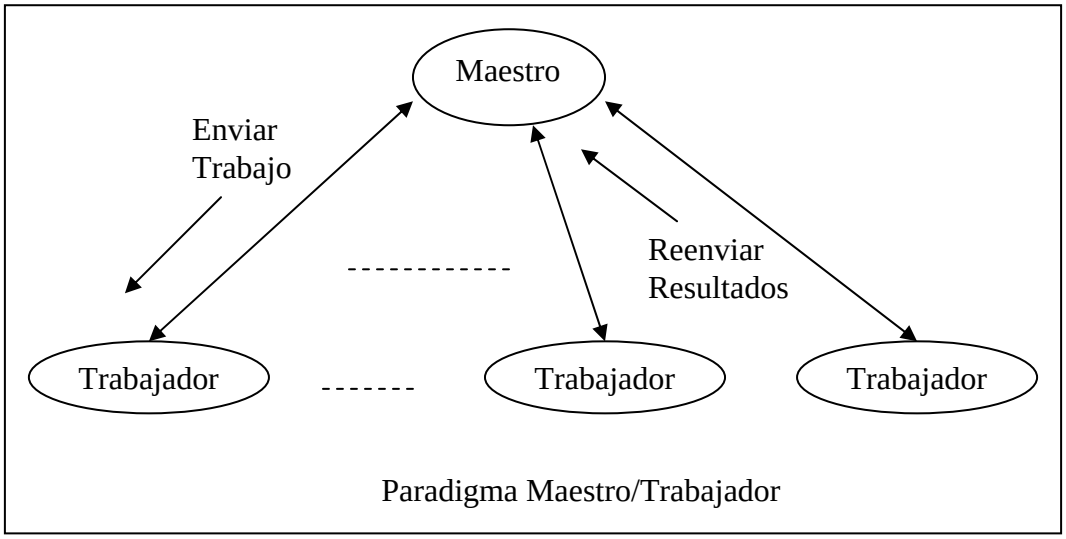


Figura. 1.2. Esquema de funcionamiento de un programa maestro/trabajador

La *organización de segmentación* de cauce se suele utilizar cuando se pretende aprovechar una granularidad más fina, en el caso de una descomposición del trabajo a realizar en una serie de funciones o tareas a través de las que van pasando los datos sucesivamente. En este caso, cada proceso se encarga de una función o varias funciones de las que constituyen el algoritmo. Como se pone de manifiesto en la Figura 1.3, estos programas llevan a cabo una descomposición funcional de la carga de trabajo, en la cual cada proceso ejecuta una o varias etapas que se pueden distinguir en la misma. Los datos pasan de etapa a etapa, por lo que la

comunicación es simple y se puede realizar de forma asíncrona. No obstante, hay que tener en cuenta que para conseguir un buen equilibrio de carga hay que llevar a cabo una buena distribución de las tareas entre las distintas funciones que constituyen el cauce. Este modelo de paralelización es eficaz en aplicaciones relacionadas con el procesamiento de señales.

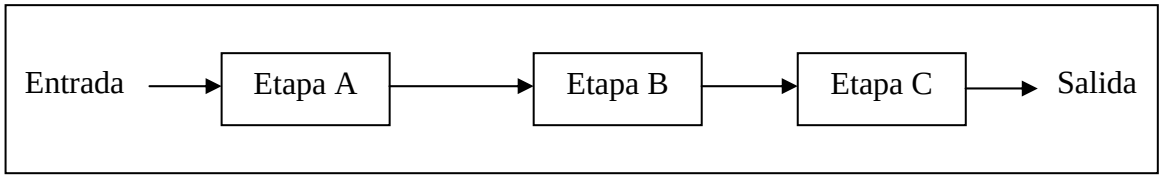


Figura. 1.3. Esquema del paradigma de segmentación de cauce.

La *organización divide-y-vencerás* se basa en una partición del problema a resolver en una serie de sub-problemas que se procesan separadamente, y cuyos resultados se componen para obtener la solución final del problema. Por tanto, existen costes adicionales asociados a la división previa del problema a resolver y a la composición final de las soluciones obtenidas para cada uno de los distintos sub-problemas. Normalmente no se realiza ninguna comunicación entre los procesos mientras éstos trabajan en cada uno de los sub-problemas generados. Las distintas etapas por las que se pasa en este paradigma, se pueden representar usualmente mediante un árbol, cuya raíz corresponde al problema principal y cada uno de los nodos finales (hojas) del árbol representa la computación a realizar en cada uno de los subproblemas.

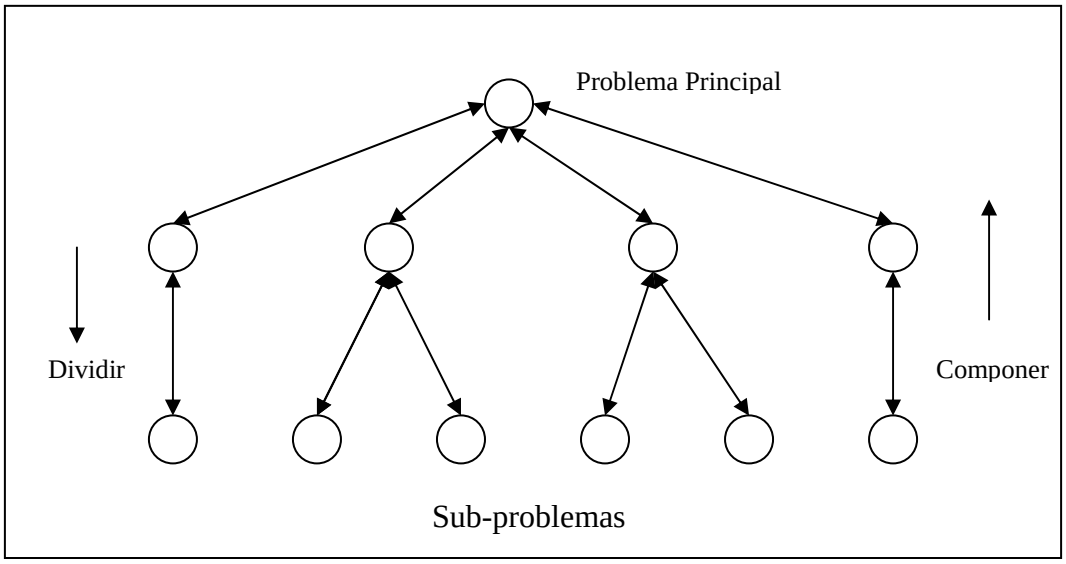


Figura. 1.4. Esquema de la organización Divide-y-Vencerás

Una vez definidos los términos SPMD y MPMD y descritas algunas organizaciones características de los programas en cada una de estas dos alternativas, pasamos a describir los modelos de programación paralela que pueden utilizarse para codificar esos programas.

1. ***Modelo de programación de espacio de memoria compartido.*** Cada uno de los procesos o hebras de control del programa utilizan un espacio de direcciones que incluye una región compartida por varios procesos, además de poder disponer también de una zona de memoria privada para cada proceso. Normalmente, la región compartida está compartida por todos los procesos. Las operaciones de acceso a memoria definidas para las zonas de memoria privadas para cada proceso también se pueden utilizar para acceder a la memoria compartida. Así, los accesos y modificaciones de las variables compartidas se realizan simplemente utilizándolas en expresiones y mediante instrucciones de asignación para lectura o escritura.

2. ***Modelo de programación de paso de mensajes.*** En este modelo se considera que cada proceso dispone de una zona de memoria local o privada a la que se puede acceder mediante las instrucciones de asignación para lectura o escritura o utilizando las correspondientes expresiones en las que participan las variables utilizadas por cada proceso. Un proceso no puede acceder utilizando estas sentencias a la memoria privada de otro proceso sino que tiene que utilizar operaciones de envío (*send*) o recepción (*receive*). A través de los *send* se transfieren datos de la memoria local del proceso que ejecuta la

primitiva a la memoria local de otro proceso. Los *receive* permiten que el proceso que los ejecuta acepte datos, que se almacenarán en su memoria local, provenientes de otro proceso. Cada par de operaciones send/receive definen una operación de sincronización punto a punto. Incluso existen lenguajes de paso de menajes que proporcionan operaciones de memoria globales o colectivas que definen transferencias entre varios procesos (de un proceso a varios o todos los procesos, recepción de varios o de todos los procesos, etc.).

3. ***Modelo de programación de paralelismo de datos.*** Este modelo de programación se recoge en algunos textos [CUL99] y corresponde a una forma de cooperación en la que los procesos actúan simultáneamente sobre elementos distintos del conjunto de datos del programa e intercambian información globalmente para después continuar su actividad hasta que se llega a un nuevo punto de sincronización/comunicación. La reorganización y el intercambio de los datos que se realiza en los puntos de sincronización/comunicación podría realizarse mediante paso de mensajes o utilizando direcciones de memoria compartidas puesto que este modelo únicamente define la forma a través de la cual se va sucediendo la computación y la comunicación. Los programas que siguen este modelo de programación suelen corresponder a una organización de tipo SPMD.

En esta memoria nos circunscribimos al modelo de programación de paso de mensajes que permite la biblioteca de paso de mensajes MPI que se ha utilizado para elaborar todos los programas paralelos que hemos usado para desarrollar el trabajo de investigación que se describe en esta memoria que, como se ha indicado, aborda el problema de la distribución de carga en el marco de los algoritmos evolutivos. Se puede decir que el objetivo que se plantea con la distribución equilibrada de la carga de trabajo es la asignación de tareas a procesadores de tal forma que cada uno de ellos realice una cantidad de trabajo similar. De esta forma se conseguiría aumentar la eficiencia del procesamiento paralelo y con ello, usualmente se alcanza una reducción del tiempo de ejecución. En la sección siguiente se presentarán unas definiciones más precisas de las métricas que determinan las prestaciones de una aplicación paralela que nos ayudarán a cuantificar el problema de la distribución de carga. Por el momento podemos indicar que, si contemplamos las etapas del diseño de una aplicación paralela que se han descrito al comienzo de esta sesión [CUL99], se puede considerar que la

distribución de carga incide sobre todas esas etapas, y si, además, esa distribución se realiza dinámicamente, se extendería a la misma ejecución del programa.

1.2 Evaluación de prestaciones en las implementaciones paralelas

Esta tesis realiza aportaciones en el contexto de los procedimientos de distribución de carga, para cuya comparación es necesario disponer de métodos de evaluación de prestaciones. Así, en esta sección se aborda el problema de la evaluación del rendimiento de las aplicaciones paralelas. No se pretende hacer un estudio exhaustivo de las métricas de evaluación del rendimiento de un sistema paralelo (para eso existen textos como [KUM94, KUC96]) sino exponer los problemas más relevantes que se plantean en este contexto y presentar los conceptos de este ámbito, y las expresiones que se van a utilizar a lo largo de esta memoria. Esta introducción nos permitirá realizar una definición precisa del problema de la distribución de carga que abordamos en esta memoria.

Así, en primer lugar nos centraremos en la definición de tiempo de ejecución de un programa paralelo y en los conceptos de ganancia de velocidad, eficiencia y coste. Terminaremos la sección analizando el concepto de escalabilidad, su importancia en el desarrollo de las aplicaciones paralelas, y las formas de contemplar y evaluar la escalabilidad.

1.2.1. Tiempo de Ejecución y Ganancia de Velocidad

El *tiempo de ejecución* es una medida clave para evaluar el rendimiento de un programa paralelo. Al comparar el tiempo de ejecución del programa en un procesador con el tiempo que tarda en ejecutarse en paralelo utilizando P procesadores se obtiene la *ganancia de velocidad* que proporciona el paralelismo. Así, si un programa que implementa un algoritmo o procedimiento de tamaño N, tarda un tiempo $T(N,1)$ en ejecutarse en un procesador, cuando se disponen de P procesadores entre los que se distribuye el trabajo que implementa el programa, el tiempo, $T(N,P)$, vendrá dado por

$$T(N,P) = t_1 + \frac{t_2}{2} + \frac{t_3}{3} + \ldots + \frac{t_P}{P}$$

si se supone que las tareas que se distribuyen entre *i* procesadores consumen un intervalo de tiempo t_i del tiempo del programa secuencial, *T(N,1)*. Por tanto, t_1 es el tiempo secuencial de las tareas que no pueden paralelizarse y se ejecutan en un sólo procesador, y t_P es el tiempo secuencial de las tareas que se distribuyen entre los *P* procesadores considerados. En la expresión anterior se está suponiendo que el trabajo que se distribuye entre *i* procesadores se distribuye por igual entre todos ellos que, por otra parte, tienen la misma capacidad. Según esto tendríamos que

$$T(N,1) = t_1 + t_2 + t_3 + \ldots. + t_P$$

En el caso de que todo el trabajo del programa secuencial se pueda distribuir entre todos los procesadores (que de esta forma se encontrarían trabajando durante todo el tiempo de ejecución del programa paralelo) se tendría que $t_i=0$ para $i \neq P$, con lo que $T(N,1)=t_P$ y, por lo tanto, $T(N,P)=t_P/P=T(N,1)/P$.

Por otro lado, la ganancia de velocidad, *S(N,P)*, se define como el cociente entre tiempo de ejecución del programa en un solo procesador y el tiempo de ejecución del algoritmo paralelo en *P* procesadores:

$$S(N,P) = \frac{T(N,1)}{T(N,P)}$$

Por tanto, cuando $t_i=0$ para $i \neq P$, dado que $T(N,P)=t_P/P=T(N,1)/P$, se tendría que:

$$S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{t_P}{t_P / P} = P$$

Es decir, se alcanzaría una ganancia igual al número de procesadores que intervienen en la ejecución del programa paralelo. En el caso en que el trabajo a realizar no se pueda distribuir siempre entre todos los procesadores, tendremos la ganancia de velocidad será menor que *P*, dado que no se podrá aprovechar la capacidad de todos los procesadores en todo momento:

$$S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{t_1 + t_2 + \ldots + t_P}{t_1 + \frac{t_2}{2} + \ldots + \frac{t_P}{P}} \leq \frac{t_1 + t_2 + .. + t_P}{\frac{t_1}{P} + \frac{t_2}{P} + .. + \frac{t_P}{P}} = \frac{T(N,1)}{T(N,1)/P} = P$$

De ahí la importancia de conseguir una distribución equilibrada del trabajo en la que todos los procesadores se encuentren siempre trabajando. Precisamente, las objeciones al uso del paralelismo parten de considerar que en cualquier programa hay una parte del trabajo que no puede paralelizarse (por ejemplo, cuando debe actualizarse un único índice). Así, si se tiene una fracción, *f*, del tiempo de ejecución del programa secuencial, *T(N,1)*, que sólo puede ejecutarse en un procesador, en el mejor de los casos, que corresponde a que en el resto del tiempo el trabajo a realizar se distribuye entre los *P* procesadores tendríamos que la ganancia de velocidad sería igual a:

$$S(N,P) = \frac{T(N,1)}{f \times T(N,1) + (1 - f) \times \frac{T(N,1)}{P}} = \frac{P}{1 + f \times (P - 1)} \qquad (1.1)$$

Esta es la ganancia máxima que se puede obtener al paralelizar un programa con una fracción, *f*, de tiempo de ejecución no paralelizable: el resto del trabajo es paralelizable pero si no todos los procesadores están trabajando en todo momento, la ganancia de velocidad será menor. Esto es lo que se conoce como la ley de Amdahl [AMD67], que establece como límite superior para la ganancia de velocidad de un programa paralelo la ganancia de velocidad dada en la expresión (1.1). A partir de la ley de Amdahl se puede ver que existe una ganancia de velocidad máxima para un programa paralelo por muchos procesadores que se utilicen. Esta ganancia máxima es igual a *1/f*, el límite de la cota que establece la ley de Amdahl cuando *P* tiende a infinito. Por otra parte, a medida que *f* se aproxima a 1, es decir, no hay ninguna parte de trabajo paralelizable, la ganancia de velocidad tiende a 1, con lo que no se obtendría ganancia alguna. Cuando *f* se hace igual a 0, la cota máxima que establece la ley de Amdahl es igual a *P*, indicando que, en ese caso se podría alcanzar una ganancia igual al número de procesadores que intervienen en la ejecución.

Hasta ahora, en el tiempo de ejecución del programa paralelo en *P* procesadores sólo se ha tenido en cuenta el tiempo asociado al procesamiento de la carga de trabajo del programa secuencial. De hecho, para contabilizar el tiempo de ejecución del programa paralelo se ha considerado únicamente la reducción en el tiempo de ejecución secuencial debido a la posibilidad de que varios procesadores trabajen en paralelo en ciertas partes del programa. Sin embargo, hay que tener en cuenta que para que el trabajo distribuido entre los procesadores se realice correctamente y se complete el programa suele ser necesario que los

procesadores intercambien datos y sincronicen su actividad en mayor o menor grado. Por lo tanto, también habría que tener en cuenta el tiempo que los procesadores consumen al comunicarse y que no se solapa con el tiempo que pasan los procesadores completando el trabajo que se les ha asignado. Además de este costo de comunicación necesario para la cooperación entre los procesadores también habría que tener en cuenta un *tiempo de sobrecarga* u *overhead* con respecto al tiempo del programa secuencial que se encuentra asociado a tareas relacionadas con la propia ejecución del programa paralelo. Entre éstas puede estar la creación de procesos, y, por supuesto, la distribución dinámica de la carga. Así, el tiempo de ejecución del programa paralelo, se puede expresar en términos de los tiempos de computación, de comunicación y de sobrecarga correspondientes al procesador *k*-ésimo (uno de los *P* procesadores que intervienen en la ejecución del programa paralelo) de la forma:

$$T(N,P) = T^k_{comp} + T^k_{com} + T^k_{sobrec} + T^k_{ocio} \qquad (1.2)$$

Además de los tiempos que consume el procesador k en el procesamiento de la carga de trabajo que le corresponde (T^k_{comp}), en la comunicación con otros procesadores (T^k_{com}), y en tareas relacionadas con la propia gestión del programa paralelo que no están incluidas en el trabajo del programa secuencial ni corresponden a tiempo de comunicación (T^k_{sobrec}), también se ha incluido el *tiempo de ocio* T^k_{ocio}, en el que el procesador no lleva a cabo ninguna actividad porque está esperando datos para continuar la ejecución de una tarea, o simplemente porque no tiene tareas que completar. Para evitar los tiempos de espera debido a la falta de datos se puede intentar que el procesador pueda realizar otro trabajo mientras le llegan los datos, solapando así la comunicación y la computación. La reducción del tiempo de espera debido a falta de trabajo es, precisamente, el objetivo del equilibrado de la carga entre los procesadores. Dado que tanto la computación como la comunicación aparecen más o menos explícitamente indicadas en los programas paralelos, los tiempos de computación y comunicación pueden estimarse de una manera relativamente fácil. Sin embargo, el tiempo de ocio puede ser más difícil de predecir, ya que en muchos casos depende del orden en el que se realizan las operaciones.

Teniendo en cuenta que la suma $T_{comp}^k + T_{com}^k + T_{sobrec}^k + T_{ocio}^k$ es igual para todos los procesadores (aunque cada uno tendrá distintos tiempos de computación, comunicación, sobrecarga, y ocio) se puede expresar el tiempo de ejecución de un programa paralelo como:

$$T(N,P) = \frac{\sum_{k=1}^{P}(T_{comp}^k + T_{com}^k + T_{sobrec}^k + T_{ocio}^k)}{P} = \frac{\sum_{k=1}^{P} T_{comp}^k}{P} + \frac{\sum_{k=1}^{P} T_{com}^k}{P} + \frac{\sum_{k=1}^{P} T_{sobrec}^k}{P} + \frac{\sum_{k=1}^{P} T_{ocio}^k}{P}$$

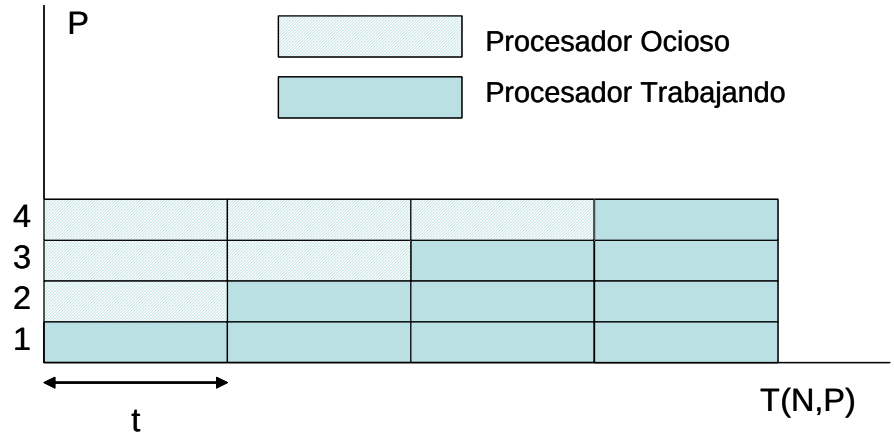


Figura 1.5. Posible ejecución paralela utilizando 4 procesadores de un programa con un tiempo de ejecución secuencial igual a 10t

Es decir, como la suma de los tiempos de computación, de comunicación, de sobrecarga, y de ocio de todos los procesadores, dividido por el número de procesadores:

$$T(N,P) = \frac{1}{P}(T_{comp} + T_{com} + T_{sobrec} + T_{ocio})$$

donde $T_{com} = \sum_{k=1}^{P} T_{com}^k$, $T_{comp} = \sum_{k=1}^{P} T_{comp}^k$, $T_{sobrec} = \sum_{k=1}^{P} T_{sobrec}^k$, y $T_{ocio} = \sum_{k=1}^{P} T_{ocio}^k$. Para ilustrar el significado de los tiempos que se han definido y las expresiones que se han considerado se puede considerar el ejemplo que se muestra en la Figura 1.5. En ella se muestra la distribución de la carga de trabajo de un programa secuencial entre cuatro procesadores. Durante un primer intervalo de tiempo, *t*, sólo está trabajando un procesador, en el segundo intervalo de tiempo están trabajando dos, en el tercer intervalo trabajan tres y, finalmente, en el último

intervalo trabajan los cuatro. Aquí no se están considerando los tiempos de comunicación ni los de sobrecarga de cada procesador. Por tanto, el tiempo de ejecución paralelo es $T(N,4)= t + t + t + t = 4t$, y el tiempo de ejecución secuencial es $T(N,1)= t + 2t + 3t + 4t =10t$. Los tiempos de computación en cada uno de los procesadores son $T^1_{comp}=4t$, $T^2_{comp}=3t$, $T^3_{comp}=2t$ $T^4_{comp}=t$ y los tiempos de ocio $T^1_{ocio}=0$, $T^2_{ocio}=t$, $T^3_{ocio}=2t$ $T^4_{ocio}=3t$. Por lo tanto $T_{comp}=10t$ (el tiempo de computación del programa secuencial $T(N,1)$) y $T_{ocio}=6t$ y $T(N,4)=(T_{comp}+T_{ocio})/4= 16t/4=4t$, como habíamos obtenido a partir de la figura. Queda claro que, en la expresión de la ganancia que nos llevó hasta la ley de Amdahl, que se muestra en la expresión (1.1), no se tienen en cuenta los tiempos de comunicación y sobrecarga. Por tanto, aunque en un programa paralelo todos los procesadores estén trabajando durante todo el tiempo en la parte paralelizable del programa secuencial (condición en la que se obtendría la máxima ganancia según la ley de Amdahl), se alcanzaría una ganancia de velocidad menor que la que establece la ley de Amdahl.

1.2.1.1. El tiempo de comunicación

Como se ha indicado más arriba, el tiempo de comunicación es uno de los componentes del tiempo de ejecución de un programa paralelo. El tiempo de comunicación es el tiempo que consumen las tareas que se están ejecutando en el procesador enviando y recibiendo mensajes. Pueden distinguirse dos tipos distintos de comunicación, la *comunicación interprocesador* y la *comunicación intraprocesador*. En la comunicación interprocesador, las tareas que se comunican están ubicadas en procesadores diferentes. En el caso de que te tuviera una tarea por procesador, toda la comunicación que se generaría sería comunicación interprocesador. En la comunicación intraprocesador, las tareas que se comunican se han asignado al mismo procesador.

En una plataforma donde la comunicación se realice a través de mensajes, el coste del envío de un mensaje entre dos tareas situadas en procesadores diferentes se puede expresar en términos de dos parámetros: el tiempo de inicio del mensaje, t_s, que es el tiempo necesario para iniciar la comunicación, y el tiempo de transferencia por palabra, t_w, que viene determinado por el ancho de banda físico del canal de comunicación que conecta los

procesadores origen y destino. Así, el tiempo de comunicación para un mensaje de L palabras vendrá dado por:

$$T_{msg}(L) = t_s + t_w L$$

La Figura 1.6 ilustra el significado de los distintos parámetros de este modelo sencillo del tiempo de comunicación.

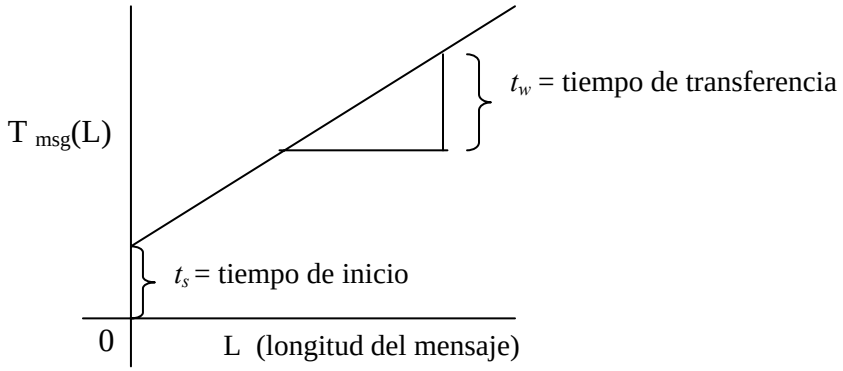


Figura 1.6: Modelo sencillo para evaluar el tiempo de comunicación: $T_{msg} = t_s + t_w L$

1.2.1.2. Eficiencia y Coste

La *eficiencia* de un programa paralelo se define como el cociente entre la ganancia de velocidad y el número de procesadores que intervienen en la ejecución del programa. Así, la eficiencia *E(N, P)* se expresa como:

$$E(N,P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{P \times T(N,P)}$$

A través de la eficiencia se expresa hasta que punto se están aprovechando adecuadamente los recursos adicionales que se están empleando en ejecutar el programa paralelo. Es decir, si se utilizan *P* procesadores, dado que se han incrementado en un factor igual a *P* los recursos de procesamiento, sería de esperar que el beneficio que se obtiene de este incremento, es decir, la ganancia de velocidad, también se incrementase en un factor igual a *P*. Mediante la eficiencia se evalúa hasta qué punto se alcanza esta situación. A partir de la ley de Amdahl se ha visto

que la ganancia de velocidad de un programa estará por debajo de *P*, y por lo tanto, la eficiencia tomará valores entre 0 y 1.

Hay que tener en cuenta que las expresiones que se han proporcionado tanto para la ganancia de velocidad como para la eficiencia no dan ninguna información acerca de la mejora real que proporciona el paralelismo frente al uso de un sólo procesador. Realmente, sólo indican hasta qué punto el algoritmo secuencial que implementaba el programa secuencial de partida está eficientemente paralelizado. Para evaluar hasta qué punto el programa paralelo que se considera constituye una mejora respecto al uso de un programa secuencial se utiliza la *eficiencia numérica* del programa paralelo, definida como el mejor tiempo de ejecución secuencial en un solo procesador, $T_{mejor}(N)$. Es decir, el tiempo empleado por un procesador del computador paralelo ejecutando el algoritmo secuencial más rápido conocido para problema. Utilizando $T_{mejor}(N)$ como tiempo secuencial en las expresiones de ganancia de velocidad y eficiencia se definen la *ganancia de velocidad total* y la *eficiencia total* mediante las expresiones:

$$\overline{S}(N,P) = \frac{T_{mejor}(N)}{T(N,P)}$$

$$\overline{E} = \frac{\overline{S}(N,p)}{P} = \frac{T_{mejor}(N)}{P \times T(N,P)}$$

No obstante, hay que tener en cuenta que estas expresiones tienen una utilidad limitada. En primer lugar, suele ser difícil determinar cual es el mejor algoritmo secuencial, ya que puede depender del tamaño del problema, *N*, de las características del hardware particular utilizado, de cuestiones relacionadas con la propia implementación del algoritmo secuencial, etc. Además, con el tiempo pueden ir apareciendo nuevos algoritmos que mejoran los existentes. Por otra parte, no siempre se dispone de una buena implementación del mejor algoritmo para un problema. Así, en la práctica, se suelen medir la *ganancia total* y la *eficiencia total* a partir del tiempo de ejecución de un algoritmo secuencial suficientemente bueno.

Otra medida que se utiliza con frecuencia y que está relacionada con la eficiencia de un programa paralelo es el *coste*, que se define como el producto del tiempo de ejecución del programa paralelo y el número de procesadores que intervienen en el mismo: $T(N,P) \times P$. Es decir, el coste se puede contemplar como la suma de los tiempos empleados por cada uno de

los procesadores al ejecutar el programa paralelo. A partir de esta definición de coste, la eficiencia puede contemplarse como la relación entre el del programa secuencial ($T(N,1)\times 1$) y el coste del algoritmo paralelo. Se puede decir que un programa paralelo es óptimo en coste si su coste es linealmente proporcional al coste secuencial cuando se utiliza $T(N,1) = T_{mejor}(N)$.

1.2.2. Escalabilidad de los Sistemas Paralelos

Como se ha visto a partir de la descripción que se ha hecho de la ley de Amdahl, la ganancia de velocidad es menor que el número de procesadores, aún sin tener en cuenta los tiempos de comunicación y de la sobrecarga asociada a la gestión del paralelismo del programa. La escalabilidad de un programa paralelo es una medida de su capacidad para aumentar su ganancia de velocidad a medida que se incrementa el número de procesadores. Así, se puede decir que un programa paralelo es escalable si es capaz de mantener constante su eficiencia con el aumento simultáneo del número de procesadores y del tamaño del problema.

En muchos programas paralelos podemos observar dos fenómenos acerca de la escalabilidad:

- Para un tamaño de problema dado, la ganancia de velocidad no aumenta linealmente con el número de procesadores, sino que aumenta más despacio hasta un punto de saturación, tal que la eficiencia disminuye con el aumento del número de procesadores.
- Para un número de procesadores dado, a medida que el tamaño del problema aumenta, la ganancia de velocidad y la eficiencia crecen.

Ambas observaciones se pueden explicar si se tiene en cuenta la ley de Amdahl a la que corresponde la expresión (1.1) que introdujimos en la Sección 1.2.1. En primer lugar, para un tamaño de problema fijo, es decir, para una fracción de trabajo secuencial, *f*, fija la variación de la ganancia máxima que se puede obtener al aumentar el número de procesadores, *P*, se puede determinar a partir de la derivada de la cota que establece la ley de Amdahl con respecto al número de procesadores:

$$\frac{\partial}{\partial P}\left[\frac{P}{1+f\times(P-1)} = \frac{1-f}{(1+f\times(P-1))^2}\right] > 0$$

como se puede observar, esta derivada es positiva pero tiende a cero a medida que crece P. Por lo tanto, cada vez se obtendría un incremento de ganancia menor a medida que aumenta el número de procesadores. Como ya se indicó en la Sección 1.2.1, la ganancia máxima no podrá ser mayor que *1/f*, por muchos procesadores que se utilicen. En cuanto a la variación de la ganancia con el tamaño del problema, si consideramos que al aumentar dicho tamaño se reduce la fracción secuencial, *f*, se tiene que la cota que establece la ley de Amdahl aumenta al disminuir *f* manteniendo *P* constante. Si se incrementa P para ese tamaño de programa mayor, la ganancia máxima que se podría obtener sería mayor ya que *1/f* crece al disminuir *f*.

La ley de Amdahl ha sido el argumento central utilizado por aquellos que dudan de la utilidad del procesamiento paralelo, sobre todo del procesamiento masivamente paralelo. Esta crítica está justificada cuando se trata de solucionar un problema particular de un tamaño fijo. Aunque la conclusión que puede extraerse de la ley de Amdahl es que, para un tamaño de problema dado habrá un número de procesadores a partir del cual no se obtienen mejoras de velocidad apreciables que justifiquen el coste asociado al uso de más procesadores. Un criterio para establecer ese número máximo de procesadores puede ser el de alcanzar un tiempo mínimo de ejecución, en el caso de que existan restricciones de este tipo para las prestaciones del programa.

En muchos casos prácticos, sin embargo, el tamaño del problema aumenta al aumentar el número de procesadores y con el poder de cómputo disponible. Un ejemplo de esta situación la constituyen los algoritmos de optimización en espacios de búsqueda de gran tamaño, y todos aquellos problemas en los que la mayor disponibilidad de recursos de cómputo y de memoria permite abordar el problema con un algoritmo más costoso que proporciona mejores prestaciones para las restricciones de tiempo existentes. Así, los sistemas paralelos grandes permiten abordar problemas más grandes que los que se solucionan sobre sistemas paralelos pequeños. Además, como se ha indicado más arriba, para muchos problemas, la fracción secuencial, *f*, converge rápidamente a cero con el aumento del tamaño del problema. Una alternativa a la ley de Amdahl es la denominada *ganancia de velocidad escalada* en la que interviene $\bar{f}$, definida como la fracción secuencial del tiempo correspondiente al programa paralelo que se ejecuta con *P* procesadores. Entonces, la ganancia de velocidad máxima alcanzable verificará que:

$$S'(N,P) \leq (1 - \overline{f}) + \bar{f}$$

dado que cuando no se pueda utilizar más que un procesador no se ganará nada, y cuando se puedan utilizar varios procesadores, en el mejor de los casos, se alcanzaría una ganancia P si todos estuvieran trabajando durante todo el tiempo. La ganancia $S'(n,P)$ se denomina *ganancia escalada*. A continuación se describen dos leyes que permiten incorporar los conceptos de escalado en la estimación de las mejoras que caben esperar del aprovechamiento del paralelismo, ofreciendo una imagen más optimista que la que se deduce de la ley de Amdahl.

Antes de presentar y analizar el significado de estas leyes, es conveniente hacer referencia a la posibilidad de alcanzar ganancias de velocidad superiores al número de procesadores utilizado. En efecto, se han observado ganancias que superan al número de procesadores, es decir *ganancias superlineales*, en determinados problemas. Una de las razones de este comportamiento se encuentra en el uso de caches en las plataformas paralelas. Al distribuir el problema entre los nodos del procesador los datos que utilizan cada uno de los procesadores tienen un tamaño más reducido y disminuye la necesidad de acceder a la memoria principal que se tendría si sólo se utiliza un procesador que debe procesar todos los datos. Otra razón puede encontrarse en que, en determinadas aplicaciones, el algoritmo paralelo lleva a cabo un procesamiento de una naturaleza diferente al algoritmo secuencial de que se partía. Un ejemplo de esta situación se ha mostrado los algoritmos genéticos [ALB02].

1.2.2.1. Ley de Gustafson

El planteamiento de John Gustafson [GUS88] para obtener un modelo de ganancia de velocidad escalada se basa en un concepto de tiempo fijo. Es decir, existen aplicaciones en las que es más importante la precisión o la calidad de las soluciones que se obtienen que el tiempo que tardan en encontrarse. De esta forma, el incremento del tamaño de la máquina permite incrementar el tamaño del problema y con ello obtener resultados mejores el mismo tiempo de ejecución que se necesitaba con un sistema con menos procesadores. De esta forma, el tamaño del problema crecería con el número de procesadores para alcanzar mejores prestaciones gracias a las posibilidades que ofrece ese incremento de prestaciones.

Así, se considera un problema de tamaño W, en el que una fracción α se ejecuta en un sólo procesador y la restante fracción, $1-\alpha$, en paralelo. En un sistema paralelo con P procesadores, el tamaño del problema se puede escalar según la expresión $W' = \alpha W + (1 - \alpha) PW$ de forma que el tiempo de ejecución en un procesador y en P procesadores se mantenga constante. De esta forma, la ganancia de tiempo fijo que establece la ley de Gustafson para un tamaño de problema escalado viene dada como:

$$S'_P = \frac{\mathrm{W}'}{\mathrm{W}} = \frac{\alpha W + (1 - \alpha) PW}{W} = \alpha + (1 - \alpha) P$$

Por tanto, la ganancia de tiempo fijo es una función lineal de P si el tamaño del problema se escala para mantener un tiempo de ejecución fijo. Es evidente que para mantener la ley de Gustafson, hay mantener la parte secuencial igual a αW. También es importante resaltar que la ganancia que proporciona la ley de Amdahl no es una ganancia de velocidad sino que realmente mide la mejora en la capacidad que proporciona el aumento de procesadores para resolver problemas más complejos.

Se puede obtener una expresión para la ganancia escalada de la ley de Gustafson si se incorporan en la misma los tiempos asociados a la sobrecarga, la comunicación, etc. Se obtendría entonces que:

$$S'_P = \frac{\alpha W + (1 - \alpha) PW}{W + T_0} = \frac{\alpha + (1 - \alpha) P}{1 + \frac{T_0}{W}}$$

donde T_0 es el tiempo asociado a todos los tiempos de sobrecarga, comunicación, etc. (el subíndice *0* hace referencia al término inglés *overhead*), y dependerá del número de procesadores *P*.

1.2.2.2. Ley de Sun y Ni

El modelo de tiempo fijo que se considera en la Ley de Gustafson no tiene en cuenta que la capacidad de memoria principal es decisiva a la hora de evaluar las prestaciones de una plataforma paralela. La ley de Sun y Ni [SUN93] generaliza las leyes de Amdahl y Gustafson considerando tanto el uso de los procesadores como la memoria. Por tanto, de lo que se trata

es de resolver el problema más grande posible teniendo en cuenta las limitaciones que establece el espacio de memoria disponible. Así, si M es la memoria necesaria para un problema y W es la carga de trabajo, estos factores se relacionan de distintas maneras dependiendo de la arquitectura del sistema paralelo. Por ejemplo, en un sistema multicomputador, la capacidad total de memoria se incrementaría linealmente con el número de nodos disponible. Si M es la capacidad de memoria de un solo nodo, entonces en un sistema paralelo de P nodos, la capacidad de memoria es $P \times M$. Se considera un problema que utiliza toda la capacidad de memoria de un nodo, con una carga de trabajo secuencial W. Cuando se dispone de P nodos, se podrá resolver un problema más grande debido al incremento de memoria, ya que ahora se dispone de un tamaño $P \times M$. Suponiendo que la porción paralela de la carga puede escalarse hasta un factor $G(P)$, la carga escalada sería $W' = \alpha W + (1 - \alpha)G(P)W$. El factor $G(P)$ refleja el incremento en la carga al aumentar la memoria P veces. La *ganancia limitada por la memoria* (incluyendo el coste asociado a las sobrecargas, comunicaciones, etc.) se define como:

$$S_P^* = \frac{\alpha W + (1 - \alpha)G(P)W}{\alpha W + (1 - \alpha)G(P)\frac{W}{P} + T_0} = \frac{\alpha + (1 - \alpha)G(P)}{\alpha + (1 - \alpha)\frac{G(P)}{P} + \frac{T_0}{W}}$$

La aplicación de esta expresión en distintas situaciones proporciona las expresiones de las distintas opciones de escalado y leyes que hemos descrito. Concretamente:

1. Si $G(P) = 1$, se tiene el caso en el que el tamaño del problema es fijo, siendo la expresión anterior de la ganancia limitada por la memoria equivalente a la ley de Amdahl.

2. Si $G(P) = P$ se tiene el caso en el que la carga se incrementa P veces cuando la memoria se incrementa P veces. Por tanto, la ecuación se corresponde con la ley de Gustafson con un tiempo de ejecución fijo.

3. Si $G(P) > P$ se tiene la situación en la que la carga se incrementa más rápidamente que la memoria. En este caso, el modelo de memoria fija proporcionaría una ganancia mayor que el de tiempo fijo.

1.2.2.3. Función de Isoeficiencia

Como se ha visto, cuando un algoritmo paralelo se usa para solucionar un problema de un tamaño fijo, entonces la eficiencia decrece con el aumento del número de procesadores *P*. Para muchos algoritmos paralelos, dado un *P* fijo, si el problema el tamaño del problema *W* se aumenta, entonces la eficiencia se crece (y se acerca de 1) porque los tiempos de sobrecarga, comunicación, etc. crecen más lentamente que *W*. Para estos algoritmos paralelos, la eficacia puede mantenerse a un valor entre 0 y 1, al aumentar el número de procesadores, a condición de que el tamaño de problema también aumente. Estos algoritmos paralelos se llaman algoritmos paralelos escalables [KUM94c]. La *función de isoeficiencia* establece el crecimiento que debe producirse en *W* con el número de procesadores *P* para mantener un valor fijo de eficiencia cuando crece *P*. Cuanto más rápidamente deba crecer *W* con *P* para mantener la isoeficiencia, menos escalable será el algoritmo.

Para algunos algoritmos, ejecutados en determinadas arquitecturas, es posible alcanzar un límite mínimo muy próximo a $O(P)$ para la isoeficiencia. Consideremos un problema cuyo tiempo de ejecución en una arquitectura paralela con *P* procesadores viene dado por *T(W,P)*. La ganancia de velocidad para este problema es T(W,1)/T(W,P) , y eficiencia por $\frac{T(W,1)}{P \times T(W,P)}$ y, por tanto, para que la eficiencia para ser una constante, $\frac{T(W,1)}{P \times T(W,P)}$ debe ser constante. Por tanto *T(W,1)* debería crecer como $\Theta(P \times T_P)$ y, considerando que *T(W,1)* es proporcional a *W*, *W* también debería crecer como $\Theta(P \times T_P)$. Además, si *T(W,P)* tiene un límite inferior de orden $\Omega(G(P))$ también tendremos un límite inferior de orden $\Omega(P \times G(P))$ para la iso-eficiencia. Si en una plataforma paralela, como es el caso de una red de computadores, es necesario que se intercambien al menos P mensajes para que los P procesadores puedan conseguir trabajo, y esos mensajes tienen que ser enviados secuencialmente a través de la red, el tiempo de ejecución tiene un límite inferior de orden $\Omega(P)$, y un límite inferior para la iso-eficiencia de orden $\Omega(P^2)$.

A lo largo de esta sección, en la que se han descrito los principales aspectos y métricas para evaluar las prestaciones del procesamiento paralelo, se ha mostrado el efecto pernicioso de una distribución no equilibrada del trabajo entre los procesadores que intervienen. En

determinadas aplicaciones la distribución de carga debe realizarse dinámicamente, al mismo tiempo que se procesa la carga de trabajo en paralelo con el consiguiente coste, que se pone de manifiesto tanto en el tiempo de sobrecarga como en el tiempo de comunicación. E la siguiente sección consideraremos estos aspectos con más detalle al con más detenimiento el problema de la distribución de carga en las plataformas paralelas.

1.3 Distribución de carga en plataformas paralelas

A través de las secciones 1.1 y 1.2 hemos proporcionado una primera aproximación al objetivo de la distribución de carga y su incidencia en las prestaciones que proporciona una plataforma paralela al ejecutar un programa paralelo. En esta sección describiremos las características generales de las alternativas que se han propuesto para resolver el problema de la distribución de carga. Para ello partiremos de una clasificación general de los procedimientos de distribución de carga, a partir de la cual se presenta la terminología utilizada en este ámbito. Después, y teniendo en cuenta la clasificación descrita, se describirán algunos de los procedimientos de distribución de carga más relevantes que se han propuesto en la bibliografía.

En lo que sigue se considerará que existen tantos procesos como procesadores se dedican a la ejecución del programa paralelo. Son las tareas las que se asignan a un proceso o a otro para hacer posible una distribución de carga equilibrada entre todos los procesadores. Por tanto, el término procesador y proceso se podrán utilizar indistintamente.

1.3.1 Clasificación de los procedimientos de distribución de carga

Como se ha indicado en las secciones anteriores, los procedimientos de distribución de carga tienen como objetivo repartir el trabajo correspondiente al programa paralelo que se ejecuta entre los procesadores de la plataforma paralela que se utiliza, para maximizar su utilización y conseguir, de esta forma, minimizar el tiempo de ejecución del programa [SIE94,ZOM01]. De esta forma, debe minimizarse la diferencia entre la carga asignada a los distintos procesadores que intervienen en la ejecución. A continuación se proporciona una categorización de los

distintos procedimientos de distribución de carga basada en una taxonomía propuesta en [CAS88] en el contexto de los procedimientos de planificación en sistemas distribuidos.

En primer lugar se puede distinguir entre procedimientos de distribución *estática* o *dinámica* de la carga. Ya hemos hecho referencia a la diferencia entre estos dos tipos de procedimientos a la hora de describir las etapas a seguir en el diseño de una aplicación paralela. Como se indicó, los procedimientos para la distribución estática parten de una descripción del trabajo que realiza el programa como un conjunto de tareas para las que se tiene un conocimiento a priori de sus interdependencias, que se supone que no se verá afectado por el estado del sistema en el momento de la ejecución del programa. De esta forma se puede realizar una asignación de tareas a procesadores previa a dicha ejecución, de forma que se alcance el objetivo de minimizar el tiempo de respuesta, minimizando para ello los tiempos asociados a la comunicación, el tiempo de espera u ocio de los procesadores, y el tiempo de sobrecarga. Por tanto, en la distribución estática, una tarea se asigna a un procesador determinado y se ejecuta en el mismo hasta que termina. La distribución estática de la carga presenta la ventaja de que no tiene ningún coste en el momento de la ejecución (no existe un tiempo de sobrecarga que se pueda asociar a la distribución de la carga). No obstante, hay que tener en cuenta que el problema de la asignación óptima de tareas a procesadores es un problema NP-completo. Esto quiere decir que no existe un programa general de complejidad polinómica que resuelva este problema. Es decir, que en problemas con dependencias entre tareas suficientemente complejos un procedimiento de carácter general puede necesitar un tiempo demasiado elevado para encontrar la solución óptima. Por lo tanto, en la práctica se deben utilizar procedimientos que, si bien no garantizan la determinación de la distribución de carga óptima sí que son capaces de alcanzar soluciones suficientemente buenas en tiempos razonables. Así, se puede distinguir entre procedimientos *aproximados*, *basados en heurísticas*, o *basados en metaheurísticas*.

Los procedimientos de *distribución de carga aproximados* son aquellos que realizan una búsqueda del espacio de soluciones hasta que encuentran una solución que satisface unas condiciones establecidas. Es decir, buscan hasta que encuentran una solución suficientemente buena. Los *procedimientos basados en heurísticas* utilizan en la búsqueda de la distribución óptima determinadas características del problema, como por ejemplo la topología asociada al grafo asociado a la interrelación entre tareas. Por lo tanto, estos procedimientos sólo son

aplicables a un dominio específico de problemas, fuera de ahí, no son capaces de proporcionar soluciones aceptables. En los últimos años se han aplicado en este ámbito procedimientos basados en metaheurísticas como el enfriamiento simulado (*simulated annealing*), las redes neuronales, la búsqueda tabú, la computación evolutiva [COR99, ZOM01], o híbridos de los anteriores [GIL98, GIL02]. Estas metaheurísticas son procedimientos inspirados en procesos naturales como la evolución natural, el enfriamiento lento de un sólido, o el funcionamiento de una red neuronal, que pueden modelar problemas de optimización sin establecer restricciones importantes respecto a sus características. Si bien no garantizan la solución óptima, sí son capaces de alcanzar soluciones suficientemente buenas que pueden mejorarse si se dispone de más recursos o se dispone de más tiempo para la evolución de la metaheurística.

Teniendo en cuenta que cada vez se dispone de mejores procedimientos para encontrar buenas soluciones para el problema NP-completo de la asignación de tareas a procesadores, y considerando que en la distribución de carga estática no lleva asociado ningún coste en el momento de la ejecución, parecería que esta alternativa constituye una buena forma para abordar el problema. No obstante, existen muchos casos en los que no es posible conocer de antemano el conjunto de tareas que constituyen la carga de trabajo de un programa y la interrelación entre las mismas. Dentro de este tipo de aplicaciones está la búsqueda exhaustiva implícita mediante algoritmos de ramificación y poda (*branch-and-bound*) o programas de simulación de fenómenos físicos basados en ecuaciones en derivadas parciales (PDE) en los que cambia la geometría del dominio en el que están definidas las PDE [BAR04]. Por otro lado, una distribución de carga estática puede no proporcionar buenas prestaciones si no se han modelado suficientemente bien las características de los recursos de la plataforma de cómputo y no se predicen bien los tiempos de ejecución de las tareas o los retardos asociados a la comunicación. Además, un procedimiento estático no puede adaptarse a circunstancias que sobrevengan en el momento de la ejecución y cambien las características que se han supuesto para la plataforma (fallos, cambios en el estado de utilización de los procesadores o de los enlaces de comunicación, etc.). En estas situaciones se necesitan procedimientos que permitan una distribución de carga *dinámica* o *adaptativa* [ANT04].

Los procedimientos de distribución dinámica de carga realizan la asignación de tareas a procesadores dinámicamente, a medida que esas tareas van apareciendo de forma que el trabajo correspondiente al programa paralelo se complete en el menor tiempo posible. De esta

forma, a lo largo de la ejecución del programa se producirán redistribuciones de las tareas para repartir la carga de trabajo. En este caso, el procedimiento de distribución de carga debe ser capaz de *encontrar* la distribución óptima de tareas concurrentemente con la ejecución del propio programa. Por tanto, la distribución dinámica de la carga conlleva un coste que proviene tanto del proceso de toma de decisión respecto a la asignación de tareas a procesadores, como del propio movimiento de tareas entre los procesadores. Para llevar a cabo el proceso de toma de decisión es necesario disponer de información global respecto de la situación de carga de los distintos procesadores. La cantidad de información que debe procesarse para repartir la carga y el coste asociado al mantenimiento de esa información relativa al estado de carga de la plataforma también debe tenerse en cuenta a la hora de evaluar las prestaciones del procedimiento de distribución dinámica de carga. Por tanto, la eficiencia de un procedimiento de distribución dinámica de carga depende del coste de la comunicación entre los procesadores, la complejidad del procedimiento de toma de decisiones, y los requisitos de almacenamiento del estado de información global en cada uno de los procesadores que participan en el procedimiento de distribución. Los procedimientos de distribución de carga deben, por tanto, alcanzar un punto de equilibrio entre las prestaciones que proporcionan y la sobrecarga que introducen. El objetivo es, por supuesto, conseguir los menores tiempos de ejecución, pero para ello, hay que destinar parte de los recursos al procedimiento de distribución. El problema es que a partir de un punto, el coste del procedimiento de distribución puede hacer que el tiempo de ejecución empiece a aumentar. Además, ese punto puede diferir según el tipo de aplicación, y por supuesto, del procedimiento de distribución de carga, que será tanto mejor cuanto menores sean los tiempos de ejecución que consiga, y para el mayor número de aplicaciones. Uno de los objetivos del trabajo de investigación que se describe en esta memoria consiste, precisamente, en utilizar los algoritmos evolutivos para extraer consecuencias acerca de las características de los procedimientos de distribución dinámica de carga. En la literatura se han propuesto multitud de procedimientos de distribución dinámica de la carga. En primer lugar podemos clasificarlos en dos grupos: los procedimientos *centralizados*, y los *distribuidos*.

En un procedimiento de distribución dinámica de carga *centralizado*, un procesador es responsable de gestionar la información global de carga del sistema. A partir de esa información, el mismo procesador se encargará de asignar las tareas a los procesadores. En el

caso de sistemas con un número de procesadores elevado las necesidades de memoria para almacenar el estado global de carga pueden ser demasiado elevadas, lo mismo que el coste asociado al mantenimiento de dicho estado. Por otra parte, la fiabilidad del sistema es reducida ya que el funcionamiento correcto depende de que el procesador encargado de la distribución de la carga no falle. En los procedimientos de distribución de carga *descentralizados* o *distribuidos*, cada procesador dispone de información local acerca del estado de carga del sistema, es decir, de su propia visión del estado global de carga del sistema. A partir de ese estado, y de la información que se intercambia con otros procesadores (que precisamente permite actualizar la información local del estado de carga), cada procesador toma autónomamente las decisiones asociadas al reparto de la carga. La frecuencia a la que se actualiza la información local de la carga del computador afecta a la sobrecarga del procedimiento y a la calidad del mismo. Cuanto menor sea la frecuencia de actualización menor será el coste de comunicación y la sobrecarga del procedimiento, pero se reducirá la calidad de las decisiones que cada procesador tome a partir de la visión parcial que tiene de la carga del sistema. Se han propuesto diversos esquemas que tienen en cuenta la topología de interconexión de los nodos de la plataforma paralela, o establecen una organización jerárquica entre los nodos para conseguir procedimientos distribuidos con mejores prestaciones. En algunos procedimientos se divide la plataforma en regiones, cada una de las cuales utiliza un procedimiento centralizado para equilibrar la carga, existiendo, además, un procedimiento distribuido que se encarga del movimiento de tareas de una región a otra. En algunos trabajos [ZNA94], estos procedimientos se denominan *semidistribuidos*, y sus prestaciones dependen bastante de la forma en que se agrupan los nodos para constituir las diferentes regiones.

Por otra parte, los procedimientos de distribución dinámica de carga distribuidos pueden ser *cooperativos* o *no cooperativos*. En los procedimientos cooperativos los mecanismos que se utilizan para repartir la carga implican la cooperación (con una mayor o menor necesidad de comunicación) entre los entre los procesadores en la toma de decisiones. En cambio, en los procedimientos no cooperativos cada procesador toma independientemente las decisiones relacionadas con la distribución de la carga. Según los procedimientos de distribución de carga cooperativos sean capaces de alcanzar la solución óptima para la distribución de la carga o no, se puede hablar de procedimientos óptimos o subóptimos. En el caso de los subóptimos también se puede distinguir entre procedimientos aproximados,

heurísticos, o metaheurísticos, de la misma forma que en el caso de los procedimientos de distribución estática de la carga. La Figura 1.7 proporciona un esquema con la relación entre los términos presentados para clasificar los procedimientos de distribución de carga [CAS88].

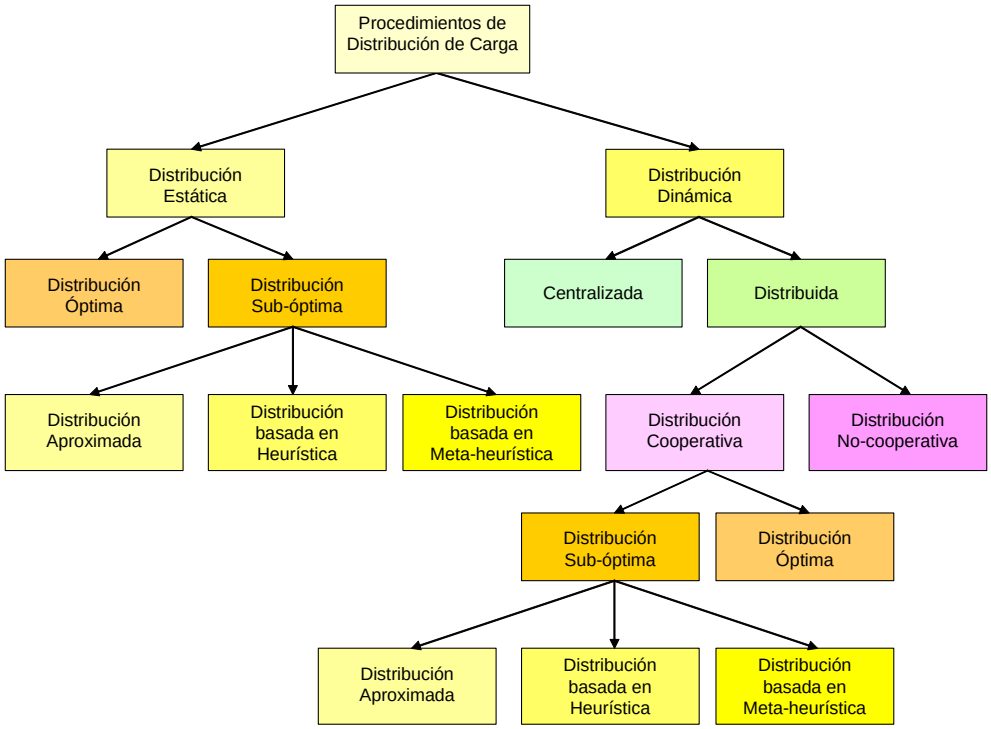


Figura 1.7. Clasificación de los procedimientos de distribución de carga (adaptada de [CAS88])

Dado que el trabajo de investigación de esta memoria se centra en los procedimientos de distribución dinámica de la carga, en la siguiente sección describiremos con más detalle los elementos característicos que definen un procedimiento de distribución de carga de este tipo.

1.3.2. Distribución dinámica de carga

Como se ha indicado los procedimientos de distribución dinámica de carga modifican el reparto de tareas entre los procesadores durante la ejecución del programa para conseguir una distribución equilibrada que permita minimizar el tiempo de ejecución. La investigación en este área es bastante activa, habiéndose propuesto una gran cantidad de procedimientos en los últimos años [ANT04, BAR04, DRI95, EVA93, HOR93, KOH95, LIN87, MUN95, SON94, WAL95, WAT98, WIL91, WIL93, XU97] que buscan mejoras en la escalabilidad y la

portabilidad, a través de nuevas propuestas para seleccionar y distribuir tareas entre procesadores que minimicen el volumen de trabajo a transferir entre procesadores para equilibrar la carga, y que mantengan o mejoren la localidad en las comunicaciones que presenta el programa paralelo. Todo ello minimizando la sobrecarga asociada a la actualización y mantenimiento de la información de carga, o bien asegurando que se encuentre dentro de límites razonables [DAN97].

En un procedimiento de distribución dinámica de la carga pueden distinguirse cinco etapas [WAT98]: *evaluación* de la carga, *iniciación* de la distribución, cálculo del *volumen de trabajo a transferir*, *selección* de tareas, y *migración* de tareas. A través de esta descomposición en etapas es posible realizar una clasificación detallada de los procedimientos de distribución dinámica de la carga analizando las posibilidades existentes en cada una de esas etapas. Además se pueden configurar diferentes procedimientos, más o menos ajustados a las características de una aplicación, combinando distintas alternativas de cada una de las fases. A continuación se indican algunos detalles de las etapas enumeradas más arriba (en el capítulo 3 se estudiarán con más detenimiento):

1. Evaluación de la carga. A partir de alguna medida que permite estimar la carga del computador se establece si la carga no se encuentra distribuida equilibradamente. Además debe disponerse de información acerca del volumen de trabajo que realiza cada una de las tareas para así poder seleccionar las tareas que deben transferirse para conseguir una redistribución adecuada de la carga. Por tanto, la utilidad de cualquier esquema de distribución dinámica de la carga depende de la calidad de las medidas de carga que se utilicen y de las predicciones que se puedan realizar a partir de ellas puesto que es necesario determinar con la suficiente precisión si se ha producido un desequilibrio en la carga. Dada la tarea j-ésima, si el volumen de trabajo asociado a dicha tarea se denomina l_j (se utiliza l por el inglés *load*), la carga del procesador i-ésimo, al que se han asignado las tareas contenidas en el conjunto T_i, será

$$L_i = \sum_{j \in T_i} l_j$$

La evaluación de la carga puede realizarse a través de la propia aplicación, a través de los recursos del sistema, o mediante una combinación de ambas alternativas. En algunas

aplicaciones resulta relativamente sencillo determinar la carga de una tarea a partir del propio algoritmo y de sus entradas. Por ejemplo, en un simulador de circuitos, la carga de trabajo puede ser función del número de puertas, y en un procedimiento de resolución de ecuaciones de elementos finitos la carga puede ser función del número de puntos de la malla que se consideren. Sin embargo existen características de la plataforma, como por ejemplo la influencia de la cache o del sistema de páginas de memoria virtual, o la propia carga del computador, que pueden afectar considerablemente al tiempo de ejecución de una tarea. Si se utilizan las llamadas al sistema para medición de tiempos es posible estimar la carga de trabajo de una tarea a partir de los tiempos de ejecución (tiempos entre operaciones de comunicación) y de comunicación (tiempo enviando o recibiendo datos). En este caso, obviamente, es difícil realizar predicciones acerca de la carga de trabajo de una tarea. La mejor opción puede se una combinación de los procedimientos de evaluación a través de la aplicación y a través de los recursos del sistema. En este caso, a través de las medidas obtenidas mediante llamadas a funciones de medida de tiempos se puede extraer información que permita ajustar modelos que relacionen las tareas de la aplicación con el tiempo de ejecución en la plataforma concreta: las medidas realizadas a través de llamadas al sistema proporcionan la realimentación necesaria para ajustar la predicción que se hace a partir de las características de la aplicación. En [WAT98] se indica que, en la práctica las técnicas sencillas como utilizar información de la historia de la carga para predecir ésta a partir de una función lineal o cuadrática son suficientemente buenas en la mayoría de los casos.

2. Iniciación de la distribución. A partir de las medidas de carga utilizadas se puede detectar si existen desequilibrios en la distribución de la carga. Se platea entonces el problema de decidir si se inician las operaciones de distribución de carga o no, y para ello es necesario comparar la sobrecarga asociada a esas operaciones con la reducción en prestaciones que se deriva del desequilibrio en la carga. Si el coste del equilibrado de la carga es menor que la mejora de prestaciones esperado, se iniciaría la distribución de la carga.

Así, para determinar el momento en que es necesario redistribuir la carga en un procedimiento de distribución dinámica de carga es preciso (1) detectar si existe una distribución desigual de la misma, y (2) determinar si el coste asociado a las propias operaciones de distribución es mayor o menor que el beneficio que se obtiene. El nivel de

equilibrio de carga (*eq*) se puede evaluar a partir del cociente entre la carga media del computador (L_{med}) entendida como la media de las cargas de los procesadores del mismo, y la carga del procesador más cargado (L_{max}). Es decir $eq=L_{med}/L_{max}$. Por lo tanto, cuanto mayor sea L_{max} con respecto al promedio de carga del computador *eq* sería más pequeño, y el procedimiento de distribución de carga debería iniciarse cuando el valor de *eq* se sitúe por debajo de umbral especificado eq_{min}. El problema de esta forma de evaluar el desequilibrio de carga es que tiene un cierto coste asociado a la necesidad de comunicación y sincronización entre los procesadores para determinar los valores medios y máximos de la carga. Una posibilidad para reducir este coste es que cada procesador determine utiliza un valor de L_{med} que calcula teniendo en cuenta sólo la carga de sus procesadores vecinos e inicia la distribución de carga si su carga supera a L_{med} en un determinado umbral. El problema que se plantea en este caso es que no se garantiza la distribución de carga equilibrada en todo el computador [WAT98]. Cuando la carga total del programa paralelo se mantenga prácticamente constante, es posible utilizar esquemas menos costosos en comunicación y sincronización. Así, se podría tomar L_{max}/eq_{min} como índice para realizar una distribución de la carga. En este caso, se iniciaría la distribución de la carga cuando la carga de algún procesador (L_i, para algún i) sea mayor que L_{max}/eq_{min}. En [LIN87, MUN95, WIL91] se utiliza una aproximación similar a ésta, aunque en estos casos se consideran dos umbrales, de forma que la carga se redistribuye si algún procesador tiene una carga mayor que un umbral superior o menor que un umbral inferior. De todas formas, si se utilizan hebras específicas para realizar la distribución de carga y se aprovechan los recursos para la comunicación asíncrona disponibles en muchas arquitecturas se puede reducir el efecto de la detección global de desequilibrios de carga. Por ejemplo, es posible calcular las cargas máximas y medias con operaciones de suma y máximo global que en la mayoría de las arquitecturas se completan en un número de pasos del orden de $log_2(P)$, siendo P el número de procesadores.

Para determinar si es beneficioso realizar la distribución de la carga es necesario estimar la reducción en el tiempo de ejecución que se conseguiría y el tiempo que consumiría la propia distribución. El beneficio se puede estimar utilizando eqmin o guardando información de lo que ocurrió como consecuencia de las distribuciones de carga pasadas. El tiempo necesario para realizar la distribución de carga se puede estimar también a partir de esa información de operaciones de distribución de carga anteriores.

3. Cálculo del volumen de trabajo a transferir. En esta fase se determina la cantidad de carga de trabajo que debe transferirse de un computador a otro para conseguir equilibrar la carga de los procesadores, una vez se ha decidido que es beneficioso realizar una redistribución de la carga. Para aprovechar la localidad de las comunicaciones esas transferencias de trabajo debería realizarse entre procesadores vecinos. Entre los algoritmos presentados en la literatura están el algoritmo de *equilibrado jerárquico* (*hierarchical balancing*, *HB*) [HOR93], el *intercambio dimensional* (*dimensional exchange*, *DE*) [XU97], y las *técnicas de difusión* (*diffusive techniques*, *DT*) [WIL93].

El *equilibrado jerárquico* constituye una aproximación recursiva de carácter global. Los computadores se dividen inicialmente en dos grupos y se determina la carga para cada partición. Si una partición está constituida por P_1 procesadores y tiene una carga L_1 y la otra partición incluye P_2 procesadores con una carga total L_2, la transferencia desde la primera a la segunda partición viene dada por la expresión

$$\Delta L_{1,2} = L_1 \quad \frac{P_1}{P_1 + P_2} \times (L_1 + L_2)$$

Una vez se ha calculado el trabajo a transferir, también denominado *vector de transferencia*, cada partición se divide y se reequilibra recursivamente, y por tanto se necesitan del orden de $log_2(P)$ pasos para calcular todos los vectores de transferencia entre procesadores.

El procedimiento de *intercambio dimensional* se planteó inicialmente para hipercubos. En este caso, los procesadores se emparejan con sus vecinos en cada dimensión y se intercambian la mitad de la diferencia de sus respectivas cargas. También se necesitan $log_2(P)$ pasos. Existe una generalización de esta técnica, denominada *intercambio dimensional generalizado* (*GDE*) para grafos arbitrariamente conectados. En este caso, para una red de grado máximo $|N_{max}|$ se colorean los enlaces de cada nodo con sus vecinos de forma que se utilice el mínimo número de colores para que ningún nodo tenga os enlaces del mismo color. Para cada color de enlace el procesador intercambia con su vecino a través de ese enlace μ

veces la diferencia de su carga. El proceso se repite hasta que se alcanza el estado de distribución equilibrada. Es decir:

$$\Delta L_{i,j}^{(k+1)} = \Delta L_{i,j}^{(k)} + \mu(L_i^{(k)} - L_j^{(k)}) \text{ con } \Delta L_{i,j}^{(0)} = 0$$

para cada nodo *j* de la vecindad del nodo *i*. En el caso de que μ=0.5 el algoritmo GDE se denomina GDE promedio (averaging GDE, AGDE) [HOR93]. Los procedimientos *HB* tienen una menor complejidad que los *GDE* por lo que se les suele considerar superiores.

Los métodos de difusión se basan en la resolución de la ecuación de difusión para la carga $\frac{\partial L}{\partial t} = \nabla^2 L$. Estos métodos, propuestos inicialmente en [CYB89] y posteriormente en [WIL93] son superiores a otros procedimientos en cuanto a robustez, prestaciones, y escalabilidad. En [HEI95] se propone un método de difusión general que utiliza un procedimiento de diferenciación implícita para resolver la ecuación de difusión. Este procedimiento permite reducir rápidamente grandes diferencias de carga pero converge más lentamente en el caso de diferencias de carga reducidas. Para evitar el efecto de este problema se puede incrementar el intervalo de tiempo de integración a medida que la diferencia de carga se reduce.

4. Selección de tareas. Después de calcular el trabajo que hay que transferir entre los procesadores se debe determinar qué tareas permiten alcanzar los volúmenes de trabajo obtenidos y deben migrar de un procesador a otro. Existen dos posibilidades para reunir el trabajo a transferir entre dos procesadores: (1) enviar tareas desde un procesador a otro, o (2) enviar tareas entre los dos procesadores con una transferencia final neta de trabajo en el sentido que se haya determinado. Esta opción es muy útil si el trabajo asociado a las tareas es grande en relación con los volúmenes de trabajo a transferir (las diferencias entre los trabajos asociados a las tareas sí podrían ser pequeñas). Este problema de selección de tareas se corresponde con el problema de *la mochila 0-1* que, como es conocido, es NP-completo. Existen, no obstante algoritmos aproximados con complejidad polinómica para resolver este problema con una determinada precisión [PAP94]. Usualmente, el algoritmo de selección no obtiene los volúmenes de carga a intercambiar que se desean por lo que se suelen realizar varios intentos. En el peor de los casos, en el que todas las tareas se encuentran en un

procesador, sólo los procesadores vecinos a dicho procesador pueden alcanzar los valores de trabajo calculados en el primer intento. Por tanto, se necesitarían un número de intentos del orden del diámetro de la red de interconexión entre los procesadores.

Migración de tareas. Las tareas que se han seleccionado en la fase anterior se transfieren de un procesador a otro. Debe asegurarse la integridad del estado de la tarea y las comunicaciones que puedan estar pendientes frente a las transferencias de tareas. La transferencia del estado de la tarea suele requerir cierta asistencia de la propia aplicación: el usuario suele tener que escribir rutinas que empaqueten y desempaqueten el estado de la tarea que migra.

Para poder desarrollar las etapas que se han indicado anteriormente es preciso definir tres políticas fundamentales:

- La *política de información*, que especifica la información de carga necesaria para tomar la decisión de transferir tareas de un procesador a otro. Sobre todo en el caso de los procedimientos distribuidos las transferencias se producen desde los procesadores más cargados a los menos cargados, y para ello es necesario un conocimiento mutuo del estado de los dos procesadores. Se necesita disponer de información local y de información remota (estado de otros procesadores del computador o información global de todo el sistema) para comprobar si se verifica la condición necesaria para empezar el proceso de distribución de la carga e identificar a los emisores y receptores de las tareas a transferir.
- La *política de transferencia*, que, precisamente, establece las condiciones para las que un trabajo debe transferirse teniendo en cuenta la carga de los procesadores y el tamaño del trabajo que debe transferirse. Esta política de transferencia, por tanto, interviene en las fases 2 (iniciación), 3 (cálculo del volumen de trabajo), y 4 (selección de tareas) que se han descrito más arriba.
- La *política de localización*, que establece los procesadores que deben intercambiar las tareas. Esta política se puede considerar que interviene en la etapa de selección de tareas y en la de migración.

- La *política de distribución*, que determina cómo redistribuir la carga entre los procesadores que han sido designados como emisores o receptores mediante la política de localización.
- La *política de selección*, que determina las tareas que deben transferirse entre diferentes procesadores. Esta política interviene, lógicamente, en la fase denominada también como fase de *selección* de tareas.

Se ha indicado que los procedimientos de distribución dinámica de carga pueden ser centralizados o distribuidos, según exista un procesador (que también puede participar en el trabajo de procesamiento) donde se implementa el procedimiento de distribución de la carga, o este procedimiento esté distribuido entre los procesadores, respectivamente. Además, se puede hablar de técnicas son *globales* o *locales*, en base al tipo de información que se tenga en cuenta para tomar la decisión de equilibrio de carga.

En los procedimientos globales, la decisión del equilibrio de carga se toma usando información global del sistema. Es decir, todos los procesadores envían su estado de funcionamiento al módulo de distribución de carga centralizado o distribuido, que, por tanto, puede ser [ZAK95]:

- **Global Centralizado (GCDLB):** En este esquema, el procedimiento de distribución de carga está localizado sobre un único procesador. Después de determinar la nueva distribución, y la rentabilidad de movimiento de trabajo, el procesador envía instrucciones a aquellos procesadores que tengan que transferir trabajo a otros. Estas instrucciones incluyen información del procesador (o procesadores) que debe recibir trabajo y la cantidad de trabajo que debe transferirse. Los procesadores que deben recibir trabajo solamente deben esperar hasta haber recibido la cantidad de trabajo que necesitan.
- **Global Distribuido (GDDLB):** En este caso, el procedimiento de distribución de carga está distribuido entre todos los procesadores. A diferencia de los procedimientos GCDLB, donde la información del estado de cada procesador se envía sólo a un procesador (donde se implementa el procedimiento de distribución de carga), en un procedimiento GDDLB, la información de estado de cada procesador se envía a todos lo procesadores. Esto permite evitar que se tengan que mandar instrucciones para el

equilibrado de la carga desde un procesador central puesto que cada procesador puede tomar las correspondientes decisiones ya que implementa parte del procedimiento de distribución de carga. Los procesadores que deben recibir trabajo esperan que éste llegue, y los procesadores que deben proporcionar ese trabajo realizan la correspondiente transmisión.

En los procedimientos locales, los procesadores se organizan en grupos. La partición del conjunto de procesadores en grupos se puede realizar, por ejemplo, teniendo en cuenta la proximidad física de los procesadores. En algunos casos los grupos se configuran seleccionando al azar los procesadores de cada grupo. Además, los grupos pueden permanecer fijos durante toda la ejecución, o cambiar dinámicamente. En este tipo de procedimientos las decisiones del equilibrio de carga se toman dentro de cada grupo. De hecho, los procedimientos globales podrían considerarse una extensión de los procedimientos locales respectivos cuando el tamaño de grupo es igual al número de procesadores. Además, en los procedimientos locales no hay intercambio de trabajo entre procesadores de grupos diferentes. Un procedimiento local puede ser:

- **Local Centralizado (LCDLB):** Esta alternativa es similar a la GCDLB. Un procesador de cada grupo se encarga de distribuir las instrucciones entre los procesadores se ese grupo. Además, hay un módulo de equilibrado de carga centralizado que asincrónicamente gestiona los diferentes grupos. Una vez que este procesador recibe las informaciones de estado de un grupo, envía instrucciones para la redistribución de la carga en ese grupo, para después pasar al grupo siguiente.
- **Local Distribuido (LDDLB):** En este esquema, el procedimiento de distribución de carga se implementa en todos los procesadores, pero las informaciones de estado se difunden sólo entre los miembros de un mismo grupo.

La ventaja de los esquemas globales consiste en que se puede alcanzar una distribución óptima de la carga. Sin embargo, son más costosos en cuanto a necesidades comunicación y sincronización. En cambio, en los esquemas locales el trabajo no suele alcanzar una distribución óptima, pero la cantidad de comunicación y sincronización que se necesitan son inferiores. Otro factor a considerar en las estrategias locales, es la diferencia entre los diferentes grupos diferentes. Así, si un grupo tiene procesadores muy rápidos, con

poca o ninguna carga, terminará pronto y permanecería ocioso, mientras otro grupo más cargado o con procesadores menos potentes permanecería ocupado sin poder ceder carga al otro grupo. Para evitar esta situación se deben implementar mecanismos que permitan la migración de tareas entre grupos.

1.3.3. Modelos de distribución dinámica de carga.

La generación de modelos que permitan anticipar las prestaciones de los distintos procedimientos para la distribución dinámica de carga es una tarea bastante ardua. Por la propia naturaleza del problema es difícil extraer expresiones analíticas precisas que permitan predecir las prestaciones de una aplicación paralela que utilice una determinada estrategia de distribución dinámica de la carga. Más bien habría generar modelos probabilísticos que permitieran estimar el comportamiento promedio de un procedimiento, para que de esta forma sea posible la selección de los parámetros óptimos para el mismo y la comparación con otros procedimientos alternativos. Para ilustrar con algunos ejemplos lo que se acaba de indicar presentaremos dos tipos de modelos basados en trabajos de Zaki [ZAK97] y de Kumar [KUM94c]. Se han propuesto bastantes modelos para la distribución de carga dinámica, en muchos casos se trata de modelos específicos para determinado tipo de aplicaciones paralelas. Dentro de éstos cabe citar la métrica descrita en [BER00] para algoritmos paralelos adaptativos de resolución de ecuaciones en derivadas parciales (PDE).

1.3.3.1 Modelo híbrido para la distribución dinámica de carga.

En [ZAK97] se propone un modelo de distribución de carga dinámica que considera tanto la distribución de carga en tiempo de compilación como en tiempo de ejecución. De ahí la denominación de modelo híbrido. El modelo que se describe en esta sección está basado en la descripción presentada en [ZAK97], donde también se ilustra cómo utilizar el modelo para seleccionar entre distintos procedimientos de distribución de carga.

Los parámetros que utiliza el modelo se clasifican en cuatro grupos: *parámetros de los procesadores*, *parámetros del programa*, *parámetros de la red*, y *parámetros de carga externa*.

Los *parámetros de los procesadores* incluyen la información relevante para el procedimiento de distribución de carga correspondiente a los procesadores de la plataforma paralela que van a utilizarse en la aplicación. Estos parámetros son los siguientes:

- ***Número de procesadores.*** Se considera un número fijo, *P*, de procesadores disponibles para la aplicación.
- ***Velocidad del procesador.*** Se especifica como una relación entre la potencia del procesador en cuestión y la de un procesador que se toma como referencia. Por tanto, en el modelo, el parámetro de velocidad es una cantidad adimensional que, realmente, se puede considerar como una especie de ganancia de velocidad. La estimación de este parámetro se puede realizar mediante ejecuciones previas o durante la compilación [ZAK95]. La velocidad del procesador i se notará como S_i.
- ***Número de vecinos.*** Este parámetro hace referencia al número, *K*, de computadores o procesadores que pueden considerarse vecinos, o próximos a un procesador dado. Este parámetro se utiliza en el caso de procedimientos locales de distribución de carga.

En un modelo de distribución dinámica de carga debe considerarse la interacción entre la aplicación paralela y la plataforma en la que se ejecuta. Los *parámetros del programa* son los que recogen la información relativa a la aplicación. La forma de parametrizar una aplicación pueden depender bastante de las características de la aplicación. En general, habría que indicar el volumen de cómputo asignado a cada una de las tareas que componen la aplicación, el tamaño de los datos que utilizan cada una de ellas, y el coste de comunicación que relaciones de precedencia.

- ***Tamaño de los datos*** *(N).* Este parámetro representa el volumen de datos que procesa cada una de las tareas de la aplicación paralela. En [ZAK97] se consideran aplicaciones constituidas por bucles que procesan matrices y, por lo tanto, este parámetro se refiere al tamaño de las matrices que se procesan. Incluso se tiene en cuenta que las matrices pueden tener tamaños diferentes en cada dimensión. Así, $N_{a,d}$ indica el número de datos en la dimensión *d* de la matriz *a*.
- ***Volumen de cómputo por tarea.*** Este parámetro se utiliza para representar el volumen de trabajo que debe completar cada una de las tareas de la aplicación. Cuanto más detalles se

conozcan de una aplicación, el modelado del trabajo de las tareas de la aplicación se podrá realizar de forma más precisa. Por ejemplo, como se ha dicho, en [ZAK97] se consideran aplicaciones constituidas por bucles cuyas iteraciones procesan datos de matrices. En este caso, el volumen de cómputo por tarea se describe a través de dos parámetros:

- ***Número de iteraciones de bucle.*** Usualmente este parámetro es función del tamaño de los datos que se procesan, por lo que se expresa como $I_i(N_{a,d})$, donde *i* hace referencia al bucle.
- ***Trabajo por iteración.*** En este caso el volumen de trabajo se mide en términos del número de operaciones básicas por iteración, por lo que es una función del tamaño de datos y se expresa como $W_{ij}(N_{a,d})$ donde *i* indica el bucle y *j* el número de iteración de dicho bucle.

- ***Comunicación intrínseca*** *(IC)***.** Especifica la cantidad de comunicación, por ejemplo debido a las dependencias de datos, que necesita cada una de las tareas que constituyen la aplicación. En [ZAK95] las tareas son iteraciones, por lo tanto este parámetro se refiere al número de bytes que deben comunicarse por iteración, y se denota como $IC_{ij}(P,N_{a,d})$, para la ejecución de la iteración *j* del bucle *i* que utiliza $N_{a,d}$ datos, correspondientes a la dimensión *d* de la matriz *a*.

- ***Comunicación de datos*** *(DC)***.** Este parámetro indica el coste de comunicación asociado al movimiento de datos debido al proceso de distribución de carga. En [ZAK97], dado que se consideran iteraciones de bucles que procesan matrices, usualmente se realizan distribución de filas o de columnas de dichas matrices. En ese caso, si se distribuye una fila o una columna, este coste de comunicación datos se hace corresponder con el número de columnas o de filas, respectivamente. Se indica como $DC_{a,ij}(N_{a,d})$, donde a es la matriz, *j* el bucle, e *i* la iteración que utiliza $N_{a,d}$ datos, correspondientes a la dimensión *d* de la matriz *a.*

- ***Tiempo por tarea***. Indica el tiempo que se necesita para ejecutar una iteración de un bucle sobre el procesador de referencia. En [ZAK97], las tareas se corresponden con las iteraciones de los bucles. Así, este parámetro se denomina *tiempo por iteración*, y se representa por $T_{ij}(W,IC)$, para la iteración *j* del bucle *i*, y depende del trabajo, *W*, y de la comunicación intrínseca, *IC*, de esa iteración. Este tiempo se podría obtener analizando el

perfil de ejecución de la aplicación, o mediante un análisis estático del programa. Puesto que el tiempo T_{ij} se establece para el procesador de referencia, el tiempo para ejecutar una iteración sobre el procesador k-ésimo será T_{ij}/S_k.

En una plataforma paralela, junto con los procesadores, el sistema los interconecta para hacer posible la comunicación y la sincronización entre ellos es una característica esencial del sistema que afecta de manera decisiva al tiempo de ejecución de la aplicación paralela. Los *parámetros de la red* se utilizan para especificar las propiedades de la red de interconexión:

- ***Latencia de Red.*** Es el tiempo necesario para enviar un mensaje sin datos entre dos procesadores. Aunque la latencia de comunicación pudiera ser diferente para diferentes parejas de procesadores, se asume que es uniforme y se expresa como L (en la sección 1.2.1.1 este parámetro se designaba como t_s, aquí se ha preferido utilizar la misma notación de [ZAK94] para no aumentar el número de subíndices y facilitar la consulta de ese trabajo).
- ***Ancho de Banda de la Red.*** Es el número de bytes que transferirse por la red por unidad de tiempo. Aquí nos referiremos al ancho de banda como B (verificándose la relación $B=1/t_w$ con el tiempo t_w que utilizamos en la sección 1.2.1.1). El ancho de banda puede ser diferente entre distintas parejas de procesadores. No obstante, aquí se considera un ancho de banda uniforme en toda la plataforma paralela.
- ***Topología de la Red.*** La topología de la red afecta a la latencia y al ancho de banda de comunicación entre dos procesadores. La topología determina el número de vecinos físicos que tiene un procesador y, en cuanto al coste de la estrategia de distribución de carga, afecta coste de comunicación asociado a la redistribución de tareas y a las características de las estrategias locales. En [ZAK97] se considera una red con conectividad completa entre todos los procesadores. En el trabajo experimental que se muestra en esta tesis se utilizan clusters de computadores interconectados por redes Ethernet con conmutadores que permiten la conexión directa de todos con todos, por lo que se ajustan a la simplificación establecida en [ZAK97].

Para completar la presentación de los parámetros que se utilizan en el modelo que se va a describir falta considerar los parámetros que definen la variación que puede producirse en la

carga del sistema y que, precisamente, determinan la utilidad de un procedimiento de distribución dinámica de la carga. En el modelo que describimos aquí, cada procesador, *i*, tiene una función de carga independiente, que se denota como ℓ_i, y que depende de dos parámetros a partir de los que se genera:

- ***Carga Máxima.*** Éste parámetro especifica la cantidad máxima de carga por procesador, se denota como *ml*, y se considerará igual para todos los procesadores.
- ***Duración de Persistencia.*** Este parámetro, *tl*, indica el intervalo de tiempo durante el que se mantiene un determinado valor de la carga. De esta forma, un valor pequeño para *tl* indica que la carga varía rápidamente, mientras que un valor elevado corresponde a una situación en la que la carga varía lentamente. Para indicar la carga el procesador *i*-ésimo durante el intervalo *k*-ésimo de la carga externa se utilizará $l_i(k)$.

Una vez presentados los parámetros que se van a utilizar en el modelo para la distribución dinámica de carga pasamos a describir dicho modelo. El modelo plantea que en un procedimiento de distribución de carga hay que tener en cuenta cuatro costes: el *coste de sincronización*, el *coste del cálculo la distribución* de la carga, el coste del movimiento de instrucciones, y el *coste del movimiento de datos*. A continuación se consideran cada uno de estos costes.

Coste de sincronización. Las estrategias de distribución dinámica de carga necesitan que se produzcan sincronizaciones entre parejas o grupos de procesadores para intercambiar información de sus estados de carga, etc., o para enviar o recibir tareas. Así, el coste de sincronización se puede especificar en función del coste asociado al tipo de comunicación (*uno-a-uno*, *uno-a-todos*, etc.) que se necesita para llevar a cabo la sincronización. A continuación se indican los costes de comunicación para las distintas estrategias (locales/globales y centralizadas/distribuidas) de distribución dinámica de carga:

GCDLB : ξ=*uno-a-todos(P)+todos-a-uno(P)*
LCDLB : ξ=*uno-a-todos(K)+todos-a-uno(K)*
GDDLB : ξ=*uno-a-todos(P)+todos-a-todos*(P^2)
LDDLB : ξ=*uno-a-todos(K)+todos-a-todos*(K^2)

Coste del cálculo de la distribución. Este coste, que notaremos como δ, suele ser bastante pequeño si se compara con el coste de sincronización y de movimiento de datos y tareas. En

cualquier caso, un procedimiento de distribución dinámica de carga debe necesitar un tiempo lo más reducido posible para calcular la nueva distribución una vez que se cumplen las condiciones de inicio de las operaciones para el equilibrado de la carga. En los procedimientos distribuidos todos los procesadores participan en el correspondiente cálculo de la nueva distribución por lo que debe considerarse estimarse el correspondiente coste δ para cada procesador. Previsiblemente, este coste será ligeramente menor en los procedimientos locales ya que en cada grupo sólo se tienen K procesadores, en lugar de tener que considerar todos los procesadores (P) del computador paralelo.

Coste de Movimiento de Datos. Para evaluar el coste asociado al movimiento de datos hay que tener en cuenta el volumen de datos que hay que transferir y el número de mensajes que se necesitan. Utilizaremos $X_i(j)$ para designar el trabajo asignado al procesador i tras la sincronización j-exima, y $\gamma_i(j)$ el número de iteraciones que el procesador i no hizo después de esa sincronización j-exima, porque fue otro procesador el que terminó primero su trabajo y originó la sincronización. Si T es el tiempo necesario para completar, en el procesador tomado como referencia, la unidad más pequeña de trabajo que puede asignarse (en [ZAK97] esta unidad de trabajo es una iteración), el tiempo que tarda el procesador más rápido (lo denominaremos procesador f) en completar la tarea que se le ha asignado antes de generar la sincronización j-ésima viene dado por:

$$t = t_j \quad t_{j\ 1} = \frac{\left(X_f(j\text{-}1) \times T\right)}{\sigma_f(j)}$$

donde t_j es el instante en que se produce la sincronización j-ésima y $\sigma_f(j)$ es la *velocidad efectiva media* del procesador, para cuya determinación se ha de tener en cuenta la carga externa que puede tener el sistema. Por lo tanto, la velocidad eficaz de un procesador en el instante k será inversamente proporcional a su carga, $S_i/(l_i(k)+1)$, donde $l_i(k) \in \{0,...,ml\}$. En el denominador se suma uno a $l_i(k)$ para evitar desbordamientos cuando $li(k)$ sea igual a 0. Puesto que el número de veces que la carga externa se ha podido modificar hasta que se produjo la sincronización $(j-1)$ viene dado por la expresión $a=\lceil t_{j-1}/tl \rceil$ y hasta que se produjo

la sincronización j-ésima por $b=\lceil t_j/tl \rceil$, la velocidad eficaz media del procesador i se puede determinar a partir de:

$$\sigma_i(j) = \frac{\sum_{k=a}^{b} S_i/(l_i(k)+1)}{b-a+1} = \frac{S_i}{\lambda_i(j)}$$

donde $\lambda_i(j)$ es la carga eficaz sobre el procesador i entre la sincronización j y la $(j-1)$, y es igual a:

$$\lambda_i(j) = \frac{b-a+1}{\sum_{k=a}^{b}(l_i(k)+1)}$$

Por tanto, la parte del trabajo (número de iteraciones en [ZAK97]) que se le asignó en la sincronización *(j-1)*-ésima que el procesador i no ha podido terminar hasta que se produjo la sincronización *j*-ésima, se puede calcular restando a $X_i(j-1)$ el trabajo realizado durante el intervalo de tiempo t, es decir:

$$\gamma_i(j) = X_i(j-1) - \left\lceil \frac{t}{T\lambda_i(j)/S_i} \right\rceil$$

y, usando la expresión para t que hemos mostrado más arriba se tiene que:

$$\gamma_i(j) = X_i(j-1) - X_f(j-1)\left(\frac{\lambda_f(j)}{S_f}\right)\left(\frac{S_i}{\lambda_i(j)}\right)$$

De esta forma, del trabajo que se distribuyó en la sincronización *(j-1)*-ésima (todo el trabajo que restaba por realizar), al producirse la sincronización *j*-ésima no se habrá concluido el trabajo:

$$\Gamma(j) = \sum_{i=1}^{P} \gamma_i(j)$$

Hasta ahora hemos considerado que todas las tareas (iteraciones en [ZAK97]) tienen una duración uniforme, como se puede observar en la expresión de *t* que hemos utilizado, donde

utilizamos para todas ellas un tiempo *T*. No obstante, se puede realizar un análisis similar al que se ha realizado considerando tareas de duración no uniforme (bucles no uniformes en [ZAK97]). Así, el tiempo consumido por el procesador f para terminar su parte del trabajo se podría expresar como

$$t = t_i - t_{i\ 1} = \sum_{k=1}^{X_f(j-1)} T_k \lambda_f(j) / S_f$$

donde k se hace variar en el conjunto de tareas (iteraciones en [ZAK97]) asignadas al procesador f. Por tanto, las iteraciones, χ, que ha completado el procesador i en el intervalo de tiempo *t* verificarán que, $\chi \leq X_i(j\text{-}1)$, y se pueden determinar a partir de la desigualdad:

$$\left(\frac{\lambda_f(j) S_i}{\lambda_i(j) S_f}\right) \sum_{k=1}^{X_f(j-1)} T_k \leq \sum_{k'=1}^{\chi} T_{k'} \leq \sum_{k'=1}^{\chi} T_{k'} \lambda_i(j) / S_i$$

Finalmente, en este caso de tareas no uniformes, el trabajo que no ha podido realizar el procesador *i* al producirse la sincronización *j*-ésima será igual a $\gamma_i(j)=X_i(j)-\chi$, y este será el que se introducirá en la expresión del trabajo que queda por hacer tras la sincronización *j*:

$$\Gamma(j) = \sum_{i=1}^{P} \gamma_i(j)$$

Este trabajo se distribuye entre los procesadores proporcionalmente a la velocidad media eficaz de cada uno de ellos. Es decir que tendremos que:

$$X_i(j) = \left(\frac{S_i / \lambda_i(j)}{\sum_{k=1}^{P} S_k / \lambda_k(j)}\right) \Gamma(j)$$

Dado que, al principio, se considera que el trabajo se distribuye por igual entre todos los procesadores se tiene que $X_i(0)$ es igual al volumen total de trabajo dividido por el número de procesadores (en las aplicaciones que se consideran en [ZAK97] en las que las tareas son iteraciones de bucles, $X_i(0)=I(N_{a,d})/P$), y además, $\gamma_i(0)= X_i(0)$ y $\lambda_i(0)=1$ para todos los procesadores, $i=1,...,P$. A partir de esas condiciones iniciales, y considerando las expresiones que hemos proporcionado para $X_i(j)$ y $\gamma_i(j)$ se puede obtener recursivamente la nueva distribución de tareas en cada punto de sincronización. El programa paralelo finalizará cuando no haya más trabajo que asignar, es decir $\Gamma(\eta) = 0$, para un valor de $j=\eta$ que, por lo tanto

corresponderá al número de puntos de sincronización que se han producido a lo largo de la ejecución del programa.

A partir de las expresiones que se han derivado se puede obtener la cantidad de trabajo reasignado, $\alpha(j)$, durante una sincronización (iteraciones en [ZAK97]), que se calcularía teniendo en cuenta que:

$$\alpha(j) = \frac{1}{2}\left(\sum_{i=1}^{P} \left| \gamma_i(j) - X_i(j) \right| \right)$$

y también se calcularía el número de mensajes que se necesitan para mover los datos, $\beta(j)$, a partir de los valores de la nueva distribución y la anterior.

Por lo tanto, el coste total de movimiento de datos asociado a la distribución de trabajo (iteraciones en [ZAK97]) sería igual a $\kappa(j)=\beta(j)L+\alpha(j)(DC/B)$. En [ZAK97], donde se redistribuyen iteraciones de bucles que trabajan con matrices, se tendría que:

$$\kappa(j) = \beta(j)\varsigma + \alpha(j)\sum_{a} [DC_a / B]$$

donde a varía dentro del conjunto de matrices que tienen que redistribuirse.

Coste de Envío de Instrucciones. Este coste sólo tiene que considerarse en los esquemas centralizados (GCDLB y LCDLB), ya que es en esos esquemas en los que el procesador central tiene que enviar el trabajo y las instrucciones de movimiento de datos a los procesadores. En esos esquemas centralizados, el número de instrucciones se puede considerar igual a $\beta(j)$, es decir igual al número de mensajes que hacen falta para mover datos, dado que las instrucciones sólo se envían a los procesadores que tienen que enviar datos. Por lo tanto, el coste de enviar instrucciones será $\psi(j)=\beta(j)L$ para los esquemas centralizados, y $\psi(j)=0$ para los esquemas distribuidos.

Coste Total. Una vez resuelto el conjunto de relaciones de recurrencia que se han proporcionado se puede obtener el coste de movimiento de datos, y calcular el número de

puntos de sincronización. A partir de ahí se puede expresar el coste total en el caso de las *estrategias globales* como:

$$TC = \eta(\xi + \delta) + \sum_{j=1}^{\eta} [\kappa(j) + \psi(j)]$$

donde, recordamos, ξ es el coste de sincronización, η es el número de sincronizaciones, δ es el coste de cálculo de redistribución, y $\kappa(j)$ y $\psi(j)$ son, respectivamente, el coste de movimiento de datos y el coste del envío de instrucciones para la sincronización número j.

En el caso de las estrategias locales, el análisis que se ha realizado hasta ahora es válido casi en su totalidad, como ahora veremos, y además se tiene un coste diferente por cada uno de los grupos. Así, el coste total de un grupo, *g*, vendría dado por:

$$TC_g = \eta_g(\xi + \delta) + \sum_{j=1}^{\eta} [\kappa_g(j) + \psi_g(j) + \Delta_g(j)]$$

donde $\Delta_g(j)$ es un factor de retardo para cada grupo que modela el hecho de que el procesador central pasa de un grupo a otro sólo cuando ha terminado de calcular la redistribución de los procesadores de un grupo y ha enviado las instrucciones correspondientes a cada procesador. Este factor de retardo depende del tiempo de sincronización de los diferentes grupos y una expresión más detallada para el mismo puede encontrarse en [ZAK96]. En el caso de esquemas locales distribuidos $\Delta_g(j)=0$, dado que no hay procesador central (puede existir una cierta desviación con respecto a la expresión anterior debido al solapamiento de la comunicación de sincronización en los grupos, pero no se ha modelado).

Así pues, dada una aplicación paralela, para obtener una estimación del coste asociado a un procedimiento de distribución dinámica de carga partir del modelo que se ha presentado es necesario conocer la evolución que se produce en la carga externa. Sólo en ese caso se podrán obtener las medidas de las velocidades efectivas medias de los procesadores, que dependen del nivel de carga que puedan tener los procesadores. Así pues, para comparar un procedimiento de distribución de carga en una aplicación paralela con otros, el modelo podría proporcionar una medida del coste medio de dicho procedimiento dada una distribución representativa de la carga externa (en [ZAK97] se considera una carga externa que evoluciona aleatoriamente). Además, si se quisiera tener una estimación del comportamiento general del

procedimiento habría que considerar un conjunto de aplicaciones paralelas suficientemente representativas, para las que habría que obtener una estimación realista de los parámetros de la aplicación que utiliza el modelo que hemos descrito. Por tanto, resulta evidente la dificultad de realizar una selección óptima de un procedimiento de distribución dinámica de carga a partir de un modelo lo suficientemente preciso de su comportamiento en una aplicación.

1.3.3.2 Esquema general para calcular el coste de comunicación

En esta sección describimos un planteamiento diferente para evaluar cuantitativamente un procedimiento de distribución dinámica de carga. Aquí se considera que, debido a la propia naturaleza de la distribución dinámica de carga, que debe recoger las condiciones de cambios de estado del sistema y el desconocimiento de la carga real de trabajo de la aplicación, es difícil obtener una expresión precisa del coste del procedimiento.

Así, en [KUM94c] se describe una estrategia de análisis que permite obtener una cota superior del coste de comunicación asociado a un procedimiento de distribución dinámica de carga en el que las operaciones de distribución de carga son iniciadas por el procesador que solicita recibir trabajo (*receptor*). En esta aproximación se considera que el coste de comunicación se debe a las peticiones de trabajo y a las transferencias de trabajo (hay que tener en cuenta que se considera una distribución de carga iniciada por el receptor). Teniendo esto en cuenta, el número de transferencias de trabajo debe ser menor que el número de solicitudes y por lo tanto, se puede tomar como cota superior del coste de comunicación el número total de solicitudes de trabajo multiplicado por el coste de la solicitud y de la propia transferencia de trabajo (U_{com}). En [KUM94c] se asume, además, que los mensajes tienen un tamaño constante.

En el procedimiento de distribución dinámica de carga que se considera para derivar la cota se considera que el procesador que termina su trabajo va solicitando dicho trabajo a los otros procesadores. Cuando encuentra un procesador con un trabajo W_i, el trabajo se divide en dos partes W_j y W_k que verifican que existe una constante α tal que $W_j > \alpha W_i$ y $W_k > \alpha W_i$. Por lo tanto, si un procesador recibe al menos una solicitud de trabajo, su carga habrá disminuido

en un factor *(1-α)*. Si se asume que en un número de solicitudes de trabajo *V(P)*, con *P* igual al número de procesadores, todo procesador ha recibido al menos una solicitud de trabajo se tendrá que $V(P) \geq P$. En general, el valor de V(P) dependerá del procedimiento de distribución de carga.

En el procedimiento de distribución de carga utilizado para obtener la cota del coste de comunicaciones [KUM94c] se supone que, inicialmente, todo el trabajo, *W*, se encuentra en uno de los procesadores. Después de V(P) solicitudes, el trabajo disponible en cualquier procesador es menor que $(1-\alpha)W$. Después de 2V(P) solicitudes, será $(1-\alpha)^2 W$, y así sucesivamente. Después de $(\log_{(1/1-\alpha)}(W/\varepsilon))V(P)$ veces, el trabajo que queda en cada procesador es menor que ε (el valor mínimo de trabajo que se considera que un procesador puede transferir). Por tanto, el número total de solicitudes de trabajo se puede hacer aproximadamente igual a $(\log_{(1/1-\alpha)}(W))V(P)$ y por tanto, una cota superior para la sobrecarga de comunicación puede ser:

$$T_{com} = U_{com} \times V(P) \times \log_{\frac{1}{1-\alpha}}(W)$$

Si consideramos que el tiempo de ejecución del programa en un procesador es $T(W,1)=W \times U_{calc}$, donde U_{calc} es el coste de un cálculo elemental, y que en el tiempo paralelo sólo tenemos la sobrecarga de comunicación asociada al procedimiento de distribución dinámica de carga, es decir $T(W,P)=1/P(W \times U_{calc}+(\log_{(1/1-\alpha)}(W)) \times V(P) \times U_{calc})$, entonces la expresión de la eficiencia vendrá dada por

$$E(W,P) = \frac{S(W,P)}{P} = \frac{T(W,1)}{P \times T(W,P)} = \frac{1}{1 + \dfrac{U_{com} \times \log_{\frac{1}{1-\alpha}} W \times V(P)}{U_{calc} \times W}}$$

Esto significa que, al tener en cuenta el coste de comunicación asociado a la distribución de carga, para que la eficiencia se mantenga constante al aumentar el número de procesadores debe aumentarse el tamaño del problema de forma que se verifique que $W \approx (U_{com}/U_{calc}) \times V(P) \times \log(W)$.

Como se ha dicho, el valor de V(P) dependerá del procedimiento de distribución de carga: en [KUM94c] se derivan estimaciones de V(P) para distintos procedimientos. El valor de U_{com}, dependerá fundamentalmente del sistema de interconexión entre los procesadores y del tamaño de los mensajes a intercambiar. También en [KUM94c] se muestran valores de U_{com} en función de P, para distintas redes de interconexión.

La cota del coste de comunicación que se ha descrito permite comparar procedimientos de distribución de carga para los que se pueda estimar el valor de V(P). Esta estimación puede ser complicada en muchos casos, aunque las expresiones que se proporcionan en [KUM94c] para distintos procedimientos son de bastante utilidad para generar otros que se puedan asimilar a ellos. No obstante, la evaluación que se haría del procedimiento de distribución de carga utilizando esta cota no se considera el coste asociado a la decisión de cómo repartir la carga entre los procesadores, ni la calidad de la distribución final que se considera, que es esencial sobre todo en el caso de que exista un nivel de interdependencia importante entre las distintas tareas.

1.4 La computación evolutiva y la distribución de carga

A partir de la descripción que se ha realizado en la sección anterior es evidente que el diseño de un procedimiento eficiente de distribución dinámica de carga para una aplicación determinada puede resultar complejo. Eso es debido, por un lado, a la variedad de políticas que hay que precisar, para cada una de las cuales no sólo existen distintas alternativas, sino que existen parámetros que hay que prefijar, presentándose comportamientos bastante diferentes cualitativamente según los valores que se escojan. Por otro lado, están las dificultades que plantean tanto el modelado de los procedimientos de distribución dinámica de carga como la aplicabilidad de esos modelos. Existe por tanto un espacio de diseño de gran complejidad y con una estructura difícil de describir formalmente a través de modelos cuantitativos que permitan aplicar procedimientos de optimización eficaces.

El trabajo de investigación que se presenta en esta memoria tiene como objetivo la utilización de los *algoritmos evolutivos* para explorar el espacio de diseño de los procedimientos de distribución dinámica de carga y determinar hasta qué punto es posible

extraer conclusiones e invariantes para las distintas aplicaciones paralelas. También se analizarán otros posibles usos de los algoritmos evolutivos en el ámbito de la distribución de carga. Concretamente, se considerará la eficiencia de los procedimientos de distribución dinámica basados en algoritmos evolutivos, comparándolos con otros *procedimientos tradicionales* de distribución de carga.

Los algoritmos evolutivos imitan la evolución natural que, precisamente, es el proceso a través del que se desarrollan complejas y bien adaptadas estructuras orgánicas. Esa evolución es el resultado de la interacción entre la creación de nueva información genética como resultado de un proceso no determinista, de su evaluación, y de su selección, de forma que los individuos que mejor se adaptan a las condiciones que originan el resto de los individuos y su entorno son los que tienen más probabilidad de tener descendencia y transmitir su código genético. Así, los algoritmos evolutivos son más flexibles y adaptables al problema que se aborda que otros procedimientos clásicos. Estos procedimientos clásicos se basan en la existencia de un modelo formal preciso del problema a resolver al que se puedan aplicar los métodos matemáticos disponibles, tales como la programación lineal o no-lineal en el caso de problemas de optimización. Esta aproximación clásica, usualmente, necesita simplificaciones de la formulación matemática original, y estas simplificaciones pueden alejar las soluciones obtenidas de las soluciones del problema real. Así pues, los algoritmos evolutivos constituyen una herramienta general adaptable para la resolución de problemas [BAC97] que se ha mostrado eficaz en muchos dominios de aplicación.

Como se ha puesto de manifiesto en la sección anterior, el problema de encontrar un procedimiento de distribución dinámica de carga es difícil de modelar, y los modelos que se obtienen suelen incluir simplificaciones importantes. Por ello, hemos considerado que los algoritmos evolutivos constituyen una buena herramienta para aproximarnos a este problema.

1.5 Conclusiones del Capítulo

En este capítulo se ha presentado el problema de la distribución de carga, en cuyo ámbito se ha desarrollado el trabajo de investigación que se presenta en esta memoria. Para poner de manifiesto la problemática que plantea este problema se ha empezado describiendo las etapas del proceso de diseño de una aplicación paralela en la Sección 1.1. A partir de esa descripción se ha introducido el problema de la distribución de carga. Puesto un procedimiento de distribución de carga persigue un reparto equilibrado del trabajo a realizar para conseguir una mejor utilización de los recursos de cómputo y con ello mejorar las prestaciones que se alcanzan en la ejecución del correspondiente programa paralelo, en la Sección 1.2 se describen las métricas más importantes para evaluar las prestaciones de un programa paralelo, se definen los componentes del tiempo de ejecución de un programa paralelo en un procesador, y se analizan los conceptos de escalabilidad, eficiencia y coste. En la Sección 1.3 se clasifican los procedimientos de distribución de la carga teniendo en cuenta, entre otros aspectos, el momento en que se realiza la distribución (distribución estática o dinámica), la implementación centralizada o distribuida del procedimiento, la utilización de información local o global en los procedimientos de distribución de carga, etc. Puesto que en esta Tesis nos centramos en los procedimientos de distribución dinámica se han descrito las características de estos procedimientos, las fases que se pueden distinguir en un procedimiento de distribución dinámica, y las políticas que determinan el comportamiento y las prestaciones de estos procedimientos. Para concluir la Sección 1.3 se han descrito dos modelos cuantitativos descritos en la bibliografía para la distribución dinámica de carga. La presentación de estos modelos nos ha permitido poner de manifiesto las dificultades de disponer de una caracterización de los procedimientos de distribución dinámica de carga que permita una selección adecuada de la mejor opción a la hora de diseñar una aplicación paralela determinada. A partir de esta situación, la Sección 1.4 justifica la idea de recurrir a la computación evolutiva para explorar el espacio de diseño de la distribución dinámica de carga.

En los siguientes capítulos se describen las aportaciones a que nos ha conducido el trabajo de investigación realizado. Así, el Capítulo 2 se centra en la distribución de carga dinámica centralizada, proponiéndose un nuevo procedimiento que nos permite establecer

comparaciones con otros procedimientos centralizados basados en computación evolutiva que se han descrito en la literatura con anterioridad. En el Capítulo 3 se considera la distribución dinámica distribuida y se presenta una taxonomía de los procedimientos distribuidos para la distribución dinámica de carga. A partir de esta taxonomía se ha podido desarrollar un algoritmo genético que permite explorar el espacio de diseño de estos procedimientos de distribución de carga, permitiendo no sólo su comparación mutua sino también la extracción de conclusiones respecto a posibles tendencias o invariantes en los valores más adecuados de parámetros de los procedimientos de distribución de carga, y a su relación con las distintas aplicaciones. Los resultados experimentales obtenidos se muestran y se comentan en el Capítulo 4, para terminar, en el Capítulo 5, con las principales aportaciones realizadas y las conclusiones del trabajo, así como con la descripción de las posibles líneas de trabajo futuro.

Capítulo 2

Equilibrado de carga dinámico y centralizado

En este capítulo se considera el uso la computación evolutiva para realizar la distribución dinámica de carga con procedimientos centralizados. Así, describiremos uno de los procedimientos que utilizan algoritmos genéticos para realizar la carga dinámica que se han propuesto más recientemente [ZOM01]. También se proporciona un nuevo algoritmo para la distribución dinámica de carga en clusters de computadores heterogéneos. Se trata de un procedimiento centralizado y adaptativo que no utiliza algoritmos de tipo evolutivo sino que se basa en la estrategia de ofertas. Este procedimiento nos servirá de punto de referencia para extraer una serie de conclusiones con relación al uso de la computación evolutiva al comparar sus prestaciones y características con el procedimiento de distribución descrito en [ZOM01]. En cuanto a la estructura del capítulo, la sección 2.1 describe con detalle el problema de la distribución de carga dinámica y centralizada, presenta algunas de las estrategias más comunes para abordar este problema (estrategia de difusión y estrategia de ofertas) y proporciona un modelo para la distribución de carga centralizada que permite estimar las prestaciones del procedimiento de distribución. La sección 2.2 se centra en la computación evolutiva y su utilización para resolver el problema de la distribución de la carga en plataformas paralelas. En esta misma sección se describe el algoritmo propuesto en [ZOM01]

para la distribución dinámica de carga utilizando algoritmos genéticos. En la sección 2.3 se presenta y analiza el algoritmo basado en la estrategia de ofertas que hemos desarrollado y que se utilizará para realizar comparaciones con el procedimiento basado en computación evolutiva descrito en la sección anterior. La sección 2.4 muestra los resultados experimentales obtenidos de la ejecución de los algoritmos de distribución de carga descritos en las secciones anteriores y se comparan sus prestaciones. Por último, en la sección 2.5 se detallan las conclusiones y las aportaciones de este capítulo.

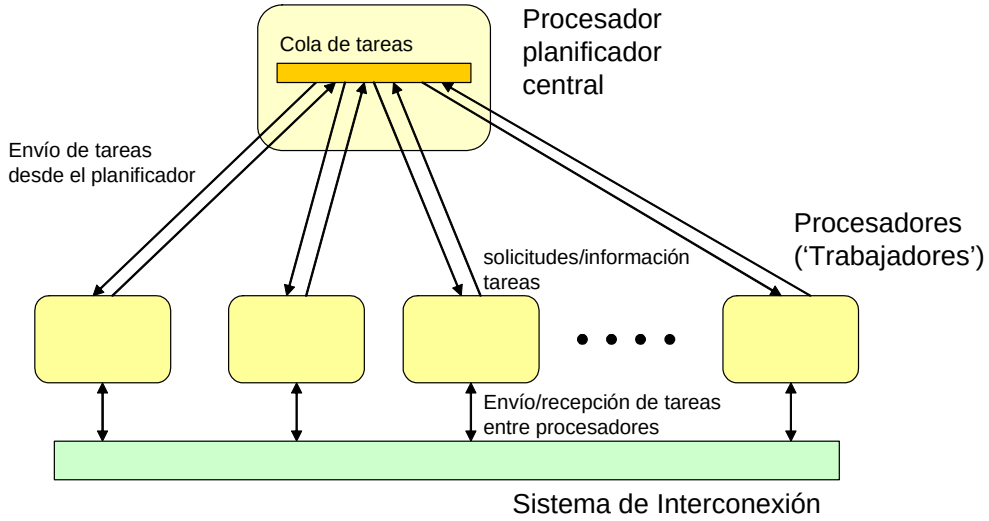


Figura 2.1. Esquema de un procedimiento centralizado de distribución de carga

2.1 Procedimientos de distribución de carga dinámica y centralizada

En este capítulo consideramos los algoritmos centralizados para la distribución dinámica de la carga. En estos procedimientos, se utiliza una información de carga global gestionada por un único procesador, denominado *planificador central*. Éste procesador también se encarga de tomar todas las decisiones relativas a la distribución de la carga de trabajo a partir de la información que recibe de los procesadores. La Figura 2.1 proporciona un esquema general correspondiente a un procedimiento centralizado de distribución dinámica de carga. El planificador central asigna tareas a los procesadores de la plataforma para que las procesen.

Según los distintos procedimientos, los procesadores pueden enviar al procesador central (1) información de su estado de carga para que éste pueda tomar las decisiones que garanticen la distribución equilibrada de la carga, (2) solicitudes de carga cuando terminan las que se les envió, o (3) tareas que en principio se asignaron a un procesador pero que deben devolverse al procesador central para que éste las reenvíe a otros procesadores. En un procedimiento centralizado los procesadores también pueden intercambiar tareas entre ellos, siguiendo las órdenes que les envía el planificador central. Si bien la implementación de un procedimiento centralizado de carga puede ser más sencilla, el coste asociado a las necesidades de memoria y comunicación para mantener el estado global de carga del sistema puede ser inaceptablemente elevado en plataformas con un número grande de procesadores. Como ejemplo, en [CER97] se propone un procedimiento centralizado de distribución dinámica de la carga que consta de cuatro fases. En primer lugar se define un *mecanismo de activación* que se inicia a través de la comunicación, síncrona o asíncrona, entre el procesador central y los demás procesadores. Una vez activado el procesador central, los procesadores evalúan su estado de carga. A partir de la información relativa a la carga de cada procesador, el procesador central toma la decisión correspondiente según la política que implemente. Por último, se produce la migración de tareas entre los procesadores. En el procedimiento de [CER97] se utiliza un paradigma de maestro-trabajador donde el proceso maestro es el procesador central, responsable de recoger de los parámetros de la carga, decidir sobre la forma de distribuir la carga, y enviar la información correspondiente a los procesos. Entre las alternativas para distribuir la carga que se comparan están las que dependen del desequilibrio de carga promedio o las que dependen del desequilibrio máximo, y las que utilizan información de carga instantánea o información histórica.

Como en cualquier procedimiento dinámico de distribución de carga, en un procedimiento centralizado se pueden distinguir una serie de fases, que se describieron en el Capítulo 1, a través de las que se implementan las funciones y las políticas que definen el procedimiento de distribución de la carga. Estas fases eran (1) la *evaluación* de la carga, (2) *iniciación* de la distribución, (3) cálculo del *volumen de trabajo a transferir*, (4) *selección* de tareas, y (5) *migración* de tareas. En un procedimiento centralizado se utiliza un único procesador para controlar la forma en que se coordinan estas fases. Así, si bien la evaluación de la carga puede realizarse en cada procesador a través de un índice de carga local (*load*

index), las decisiones relacionadas con la iniciación de las operaciones de distribución de carga, la determinación del volumen de trabajo que se transfiere y la selección de tareas, así como la determinación de los procesadores entre los que se transfieren las tareas son controladas por un único procesador. Además, en cuanto a la medida de la carga del procesador, índices sencillos como el tiempo de respuesta suelen proporcionar una información suficiente y pueden determinarse de forma eficiente, lo cual es importante si se tiene en cuenta que la medida de la carga del procesador debe realizarse frecuentemente.

También vimos en el capítulo anterior que existen cuatro políticas o reglas que definen los detalles esenciales para implementar cada una de las cinco fases anteriores. Por un lado está la *política de información*, que se refiere a la forma de captar, almacenar y gestionar la información necesaria para tomar todas las decisiones relacionadas con la distribución de la carga. Además se tienen otras tres políticas que caracterizan la forma de realizar la distribución de la carga propiamente dicha: la *política de transferencia*, la *política de localización*, la *política de distribución*, y la *política de selección*.

La *política de información*, especifica la forma a través de intercambiar y gestionar la información necesaria para repartir la carga de trabajo. Esta política debe mantener un compromiso entre el coste asociado al acceso y almacenamiento de la información del estado global de carga del sistema y el mantenimiento de una visión actualizada y precisa de dicho estado. En un procedimiento centralizado de distribución dinámica de la carga global el estado de la carga se almacena en el procesador central, que recibe dicha información comunicándose con el resto de procesadores del sistema. Por otra parte, la política de información también establece si las decisiones relacionadas con la distribución de carga se toman a partir de la información de carga de todos los procesadores o únicamente de un subconjunto de ellos. Existen tres tipos de políticas de información relevantes: la política de intercambio de información *periódica* y la de intercambio de información *a demanda*. En la *política periódica centralizada*, los procesadores envían información de carga al procesador central tras determinados intervalos de tiempo, aunque esa información no sea necesaria en ese instante. La ventaja de esta política está en que, en el momento en que se deba iniciar la distribución de la carga se dispone de una información bastante actualizada de dicha carga, evitándose el correspondiente retraso que se produciría si es necesario actualizar el estado de carga del sistema. Por supuesto, la clave de la política de intercambio periódico de

información estriba en la frecuencia con que se suministra dicha información. A medida que crece la frecuencia se dispondría de una información más actualizada de la carga pero a costa de una sobrecarga de comunicación mayor que puede reducir el beneficio final que proporciona la propia distribución de carga. En la política de *información a demanda,* la información de la carga se comunica al procesador central justo antes de que se vaya a tomar la decisión acerca de realizar una operación de distribución de carga. De esta forma, si bien se reduce la comunicación asociada al mantenimiento de la información de la carga del computador, la política de información aumenta el tiempo necesario para realizar la distribución de carga.

La *política de transferencia*, establece las condiciones en las que debería transferirse trabajo de un procesador a otro. La transferencia se denomina *iniciada por el emisor*, cuando es el procesador que está sobrecargado el que solicita al procesador central (en el caso de una distribución de carga centralizada) transferir tareas a otros procesadores iniciando la distribución de carga. Por otra parte, cuando es un procesador poco cargado el que solicita trabajo al procesador central (igual si la distribución de carga es centralizada) la transferencia se llama *iniciada por el receptor*. Si el procedimiento de distribución de carga combina las características de la transferencia iniciada por el emisor y el receptor, la política de transferencia es *simétrica*. Los procedimientos de distribución de carga eficientes con política de transferencia simétrica se comportan como los de transferencia iniciada por el emisor en situaciones de poca carga en los procesadores, y como los de transferencia iniciada por el receptor cuando las cargas de los procesadores son elevadas [ANT04].

La política de transferencia está relacionada con la *política de localización*, que permite determinar los procesadores que participan en la distribución de la carga. En general, los procedimientos de distribución de carga utilizan información local para comprobar si hay que iniciar el equilibrado de la carga (política de transferencia), e información remota o un procesador central para identificar los procesadores que deben intervenir en la transferencia (política de localización). Así pues, para definir estas políticas se consideran los índices de carga que se utilicen para cada procesador, y los recursos que necesita cada tarea. Para evaluar el correspondiente índice de carga de un procesador se debe estimar su carga en el momento en que se está ejecutando el programa paralelo. Una posibilidad para estimar la carga es utilizar el tamaño de las colas de tareas que esperan para su ejecución en cada procesador.

Entre las políticas de localización están las que utilizan *un umbral*, de forma que un procesador se toma como emisor (envía tareas a otros procesadores) o receptor (recibe tareas de otros procesadores) si su carga es, respectivamente, superior o inferior a un umbral preestablecido. También es posible utilizar *dos umbrales*, de forma que un procesador será emisor si su índice de carga es superior al umbral mayor y será receptor si su índice de carga es inferior al umbral menor. Otra alternativa, política de localización basada en el menos cargado (*shortest-based*), consiste en seleccionar como procesador receptor aquél que tiene un índice de carga menor. Por otra parte, a la hora de determinar los procesadores que intervienen en la redistribución de la carga hay que tener en cuenta la posibilidad de que se produzcan inestabilidades. Éstas pueden aparecer cuando varios procesadores sobrecargan a otro procesador que estaba poco cargado al transferirle tareas en el proceso de redistribución de la carga. Esto se puede evitar mediante una selección de procesadores ligeramente cargados de manera arbitraria, en vez de elegir el procesador con la carga mínima, y la utilización de una *política de localización aleatoria* para determinar el procesador receptor. En cualquier caso, la política de localización puede ser *global*, si puede designar como emisor o receptor a cualquier procesador de la máquina, o *local*, en el caso de que se pueden emparejar emisores y receptores con cierto grado de vecindad.

La *política de distribución* establece la forma de redistribuir la carga entre los procesadores que han sido designados como emisores o receptores mediante la política de localización, de la que, por tanto, depende. Finalmente, la *política de selección* establece qué tareas se eligen para ser transferidas. Si la tarea que se está ejecutando puede interrumpirse y ser transferida a otro procesador se habla de *selección con corte forzado* (*preemptive*), mientras que si sólo se pueden seleccionar las tareas que están esperando a empezar su ejecución la selección es *sin corte forzado* (*no preemptive*).

2.2. La computación evolutiva en la distribución dinámica y centralizada de carga

La *computación evolutiva* incluye una serie de técnicas con el denominador común de estar inspiradas en los principios de la evolución natural [BACK97]. Aunque sus orígenes pueden situarse en los años 50, empezaron a resurgir a partir de los 90, a partir de tres aproximaciones

fuertemente relacionadas, pero que se han desarrollado de forma independiente: los *algoritmos genéticos* [GOL89, HOL75, MIC92], la *programación evolutiva* [FOG62, FOG66, FOG92], y las *estrategias evolutivas* [REC73, SCH95]. En el ámbito de los algoritmos genéticos han aparecido nuevas variantes o ramas como resultado de las distintas aplicaciones y de la propia dinámica de investigación en algoritmos genéticos. Así están los *sistemas clasificadores* (*classifier systems*) [HOL86, SER90] y la *programación genética* [KOZ92]. En [BAC97] se describen todas estas aproximaciones a la computación evolutiva, junto con una amplia bibliografía hasta el año 1999. También puede consultarse [MIC00]. En esta memoria nos centraremos en los algoritmos genéticos, que pueden considerarse como la aproximación a la computación evolutiva más extendida y que ha sido la más ampliamente utilizada en el ámbito de la distribución de carga y planificación de tareas [ZOM99, ZOM01, ZOM01b, KID94], fundamentalmente en la distribución estática y en la dinámica centralizada. La estructura general de un algoritmo evolutivo se muestra a continuación [MIC00]:

```
t=0;
inicializar(P(t));
evaluar(P(t));
while (not final) do
{
        P'(t+1)=seleccionar(P(t));
        P(t+1)=transformar(P'(t+1))
        evaluar(P(t+1)) ;
        t=t+1 ;
}
```

En el algoritmo anterior, *P(t)* es una población de μ individuos, es decir un conjunto de μ soluciones del problema, en la generación *t* (esto es, en la iteración *t* del algoritmo). En cada iteración se evalúa la calidad de cada una de las soluciones de la población con arreglo a una medida o *función de idoneidad* o *fitness*. Teniendo en cuenta la idoneidad de cada una de las soluciones de *P(t)* se selecciona un conjunto de soluciones, *P'(t+1)*, al que se aplican una serie de operaciones de transformación que permiten definir la población *P(t+1)*, que se utilizará en la nueva iteración del algoritmo (la nueva generación). Se pueden encontrar en la bibliografía otras descripciones generales de la estructura de un algoritmo evolutivo. Por ejemplo ([BAC97]), en algunos casos se aplican primero las transformaciones sobre la población *P(t)*, para generar una subpoblación, *P'(t)*, de λ individuos que, tras ser evaluada se

une al conjunto *Q* de soluciones de *P(t)* (*Q* puede ser igual a *P(t)* o incluso ser un conjunto vacío) para seleccionar de entre la unión de *P'(t)* y *Q* (*P'(t)*$\cup$*Q*) los individuos de la nueva población P(t+1). En cualquier caso, para definir un algoritmo evolutivo hay que especificar las operaciones de transformación que se aplican a los individuos de la población, la función de idoneidad que se utiliza para evaluarlos, y la forma de seleccionar los individuos que constituyen la población para la siguiente generación.

En un algoritmo genético, las transformaciones que se aplican son la *mutación* aleatoria y el *cruce* (*crossover*) de soluciones elegidas aleatoriamente, y existen diversas formas para seleccionar a los individuos para la constituir la nueva población. Una forma de realizar la selección es el denominado *método de la ruleta* donde se asigna una probabilidad de selección a cada individuo con arreglo a su idoneidad. Además existen procedimientos de selección elitistas que mantienen en la nueva población a los individuos más idóneos (un subconjunto de los individuos más idóneos de una generación pasan directamente a la siguiente generación). Así pues, un *algoritmo genético* (GA) puede contemplarse como un algoritmo de búsqueda que combina la *explotación* de resultados obtenidos en iteraciones anteriores y la *exploración* de nuevas regiones del espacio de búsqueda. Para eso, un GA utiliza una población cuyos individuos son los puntos del espacio de búsqueda, es decir, las soluciones posibles al problema planteado. A esa población se aplica el *principio de la supervivencia* del individuo mejor adaptado junto con una forma aleatoria pero estructurada de generar nuevos individuos en la población. Como se ha indicado más arriba, en cada nueva generación de individuos, la población incorpora nuevos individuos que se han creado a partir de los individuos que existían en la población de partida, que constituía la generación anterior. Para obtener los individuos de la nueva generación primero se *seleccionan* individuos que representan las mejores soluciones de la generación presente. Así, los individuos de la nueva población se generan a partir de dichas soluciones aplicando dos operadores: la *mutación*, y el *cruce* (*crossover*) de soluciones seleccionadas. Por tanto, son las características de las operaciones de selección, mutación, y cruce son las que establecen la mezcla de exploración y explotación que determina la eficacia de un algoritmo genético. La selección es la que permite la explotación, mientras que la mutación y el cruce permiten la exploración de nuevas posibilidades al modificar algunas de las soluciones que se han seleccionado. Los algoritmos genéticos no son simples búsquedas aleatorias (*random walks*) sino que, en cierto modo,

explotan la información histórica de que se dispone para estimar qué nuevas soluciones pueden proporcionar una mejora en la calidad de las soluciones obtenidas. En muchos de los procedimientos de optimización existentes se va pasando de un punto a otro del espacio de soluciones utilizando una regla de transición para determinar el siguiente punto a considerar. En un algoritmo genético se trabaja con un conjunto de puntos simultáneamente. De esta forma, la posibilidad de que el procedimiento quede atrapado en un óptimo local se ve reducida.

A continuación nos centraremos en la descripción detallada de un procedimiento dinámico y centralizado para la distribución de carga basado en un algoritmo genético [ZOM01]. A lo largo de esa descripción se presentarán los detalles específicos del algoritmo genético, incluyendo los operadores que se aplican, y la forma de seleccionar las soluciones para constituir la población de la siguiente generación. Esas descripciones se circunscriben a la aplicación de distribución de carga considerada, para una descripción detallada de los algoritmos genéticos en general se puede consultar, por ejemplo, [GOL89].

2.2.1. Distribución de carga dinámica centralizada basada en algoritmos genéticos.

En la mayoría de los casos la aplicación al problema de la distribución de carga de los algoritmos genéticos y otras metaheurísticas como el enfriamiento simulado y la búsqueda tabú, se ha centrado en la distribución estática [ZOM99]. No obstante, se han propuesto algunos procedimientos de distribución dinámica centralizada basados en algoritmos genéticos [GON94, KID94, ZOM01]. A continuación describiremos con detalle el procedimiento propuesto en [ZOM01] para ilustrar el uso de los algoritmos genéticos en este ámbito y las prestaciones que cabe esperar de esta alternativa. Este procedimiento servirá como referencia para comparar con otro algoritmo de distribución dinámica centralizada de carga que hemos propuesto y que se describe en la Sección 2.3.

El procedimiento de distribución dinámica de carga descrito en [ZOM01] utiliza un algoritmo genético para seleccionar las tareas que deben pasar de un procesador a otro. Para ello, el algoritmo genético ejecuta una serie de iteraciones durante un intervalo de tiempo prefijado cuando se verifica la condición de iniciar operaciones de distribución de carga. Es

decir, cuando se inicia la ejecución del algoritmo genético para redistribuir la carga, esta ejecución no finaliza cuando el algoritmo alcanza la convergencia (no se producen cambios apreciables en la solución que ha alcanzado en iteraciones sucesivas), sino una vez ha transcurrido un tiempo prefijado para ello. Este tiempo se determina de forma que la sobrecarga asociada a la ejecución del algoritmo genético para determinar el nuevo reparto de tareas no anule la mejora que supone la nueva distribución de carga. El problema es que, según se indica en [ZOM01], a veces el algoritmo genético no obtiene una mejor distribución de tareas tras el tiempo de ejecución prefijado. En ese caso habría que continuar la ejecución durante un nuevo intervalo de tiempo, reduciéndose la eficiencia del procedimiento.

Para describir la forma en que un algoritmo genético se ha aplicado a un problema determinado es necesario detallar una serie de aspectos que definen la implementación concreta del algoritmo genético para ese problema. Estos aspectos, que se van a describir a continuación, son la forma de codificar las soluciones, la función de idoneidad utilizada, y las características de los operadores de selección, mutación y cruce.

Función de idoneidad. A través de esta función se evalúa la calidad de una solución en el contexto del objetivo que se pretende alcanzar. En el caso de la distribución de carga, el objetivo que se persigue es realizar una asignación de tareas para las que el tiempo de ejecución se minimice, con una distribución equilibrada de la carga entre todos los procesadores y una utilización máxima de los procesadores. Así, en [ZOM01] la función de idoneidad, *f*, se define como:

$$f = \frac{1}{t_{ejec}} \times u_{med} \times \frac{CA}{P}$$

donde t_{ejec} es el tiempo que tardaría el procesador más cargado en terminar de ejecutar sus tareas, u_{med} es la utilización media de los procesadores, *CA* es el número de colas aceptables, y *P* el número de procesadores. Para ilustrar el significado de cada una de las variables de la función de idoneidad utilizaremos la Figura 2.2, en la que se consideran cuatro procesadores entre los que se han distribuido una serie de tareas, a cada una de las cuales se asigna una carga. En la figura no sólo se indica la carga de las tareas que se asignan a cada procesador para su ejecución futura sino que también se muestra la carga de cada procesador correspondiente a la tarea que está ejecutando. De esta forma, el tiempo necesario para

completar el trabajo con la distribución de carga indicada en la figura y el estado de carga existente es t_{ejec}=35, es decir el valor máximo del tiempo que necesitaría cada procesador para completar su parte de trabajo. La *utilización* del procesador i-esimo, u_i, se define como el cociente entre el tiempo, t_i, que necesita ese procesador para completar las tareas que se le han asignado y el tiempo necesario para completar todo el trabajo t_{ejec}: $u_i = t_i / t_{ejec}$. Así, u_1=10/35, u_2=24/35, u_3=35/35, y u_4=27/35. La media de las utilizaciones de todos los procesadores es la utilización media. En el ejemplo de la figura se tiene que $u_{med}=(u_1+u_2+u_3+u_4)/4$=0.686. Junto con el tiempo necesario para completar todo el trabajo y la utilización de los procesadores, en la función de idoneidad también se tiene en cuenta que aunque la asignación de tareas tenga un tiempo de ejecución bajo y una utilización de procesadores alta, puede dar lugar a una sobrecarga en alguna de las colas. Así se definen dos umbrales, uno que indica que el procesador está poco cargado (umbral inferior) y otro que indica que el procesador está muy cargado (umbral superior). La cola de un procesador es aceptable desde el punto de vista de la distribución de la carga si el tiempo que necesitaría para completar las tareas asignadas en esa distribución es menor que el umbral superior y mayor que el umbral inferior. Por ejemplo, si se toma un umbral inferior de 20 y un umbral superior de 30, el procesador P1 está poco cargado y el procesador P3 está muy cargado, sólo las colas de los procesadores P2 y P4 son aceptables, y por tanto *CA*=2. Con esto *CA/P*=0.5. Cuanto mayor sea este porcentaje, mejor es la distribución de tareas desde el punto de vista del equilibrado de carga (menos necesidad de tener que volver a redistribuir la carga).

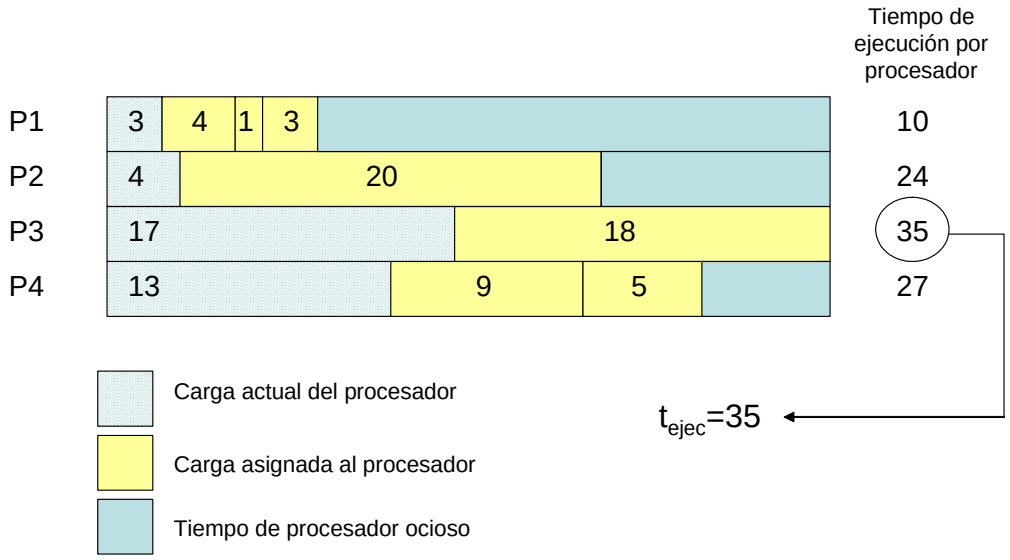


Figura 2.2. Ejemplo de distribución de tareas para ilustrar la función de idoneidad descrita en [ZOM01]

Codificación de las soluciones. Para describir los operadores de mutación y cruce que se utilizan en un algoritmo genético es necesario conocer la forma a través de la cual un individuo de la población codifica una solución al problema que se aborda. La Figura 2.3(a) muestra una asignación de tareas a procesadores, para cada tarea se indica su número y, entre paréntesis, el tamaño de dicha tarea. Como se observa, la codificación que se establece es una codificación decimal en lugar de binaria que, para este tipo de problemas, no es muy adecuada dado que da lugar a cadenas demasiado largas para incorporar toda la información que se necesita [ZOM01]. Además, se trata de una codificación bidimensional que, fácilmente, se puede transformar en una cadena unidimensional, tal y como se muestra en la Figura 2.3(b). A estas cadenas se aplican las operaciones de selección, mutación y cruce del algoritmo genético. Dado que el procedimiento de distribución de carga es dinámico, debe determinar la mejor asignación de tareas a medida que éstas van apareciendo. No obstante, como en un momento dado pueden existir muchas tareas esperando a que se les asigne un procesador se utiliza una técnica de *ventana deslizante* (*sliding-window*) de forma que, en cada momento sólo se consideran las tareas que existen dentro de una ventana, con un tamaño prefijado. De esta forma el número de tareas incluidas en cada individuo de la población es igual al tamaño

de la ventana. Una vez que el algoritmo genético ha asignado las tareas a los procesadores, la ventan se desplaza, incorporando nuevas tareas para el nuevo proceso de asignación.

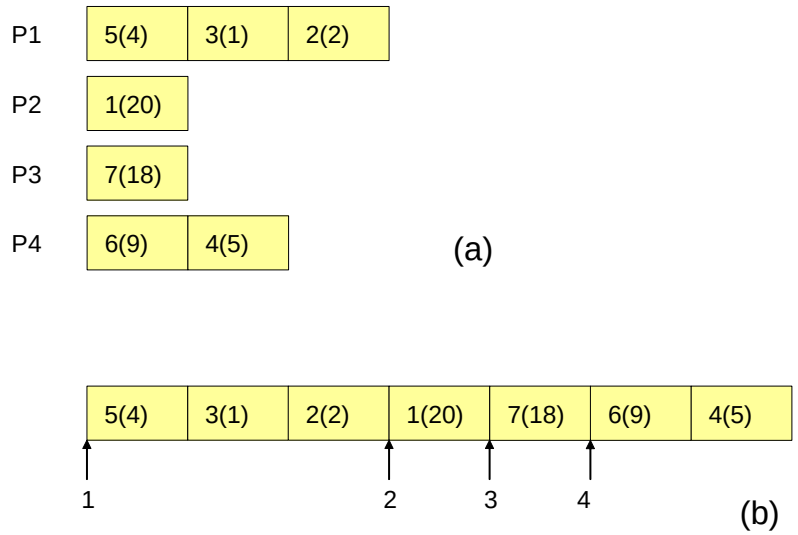


Figura 2.3. Codificación bidimensional (a) y monodimensional (b) de las soluciones

Selección. La selección de soluciones se realiza mediante el procedimiento de la ruleta [GOL89], que empieza asignando a cada individuo un trozo de ruleta cuyo tamaño es proporcional a su idoneidad: la fracción de ruleta que representa ese trozo se obtiene dividiendo la idoneidad del individuo entre la suma de las idoneidades de toda la población. Una vez repartida la ruleta entre los individuos se generan números aleatorios que corresponderán a distintas zonas de la ruleta, seleccionándose el individuo asignado a esa zona.

Mutación. Se utiliza la denominada *mutación por intercambio*, que se implementa permutando dos tareas seleccionadas aleatoriamente. Para asegurar que realmente se genera una nueva solución tras la mutación, cada tarea se elige de un procesador distinto, que también se elige aleatoriamente. De la codificación de cada individuo, esta operación necesita sólo el número de tarea y de procesador.

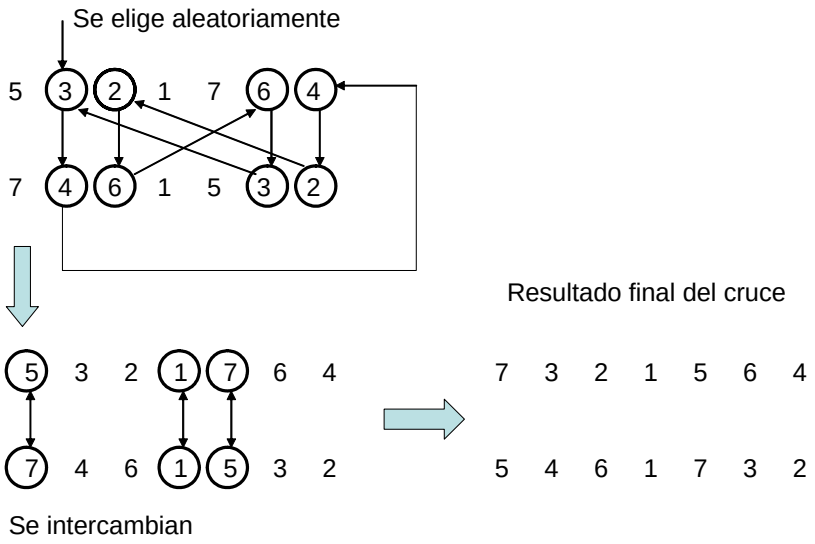


Figura 2.4. Descripción de la operación de cruce cíclico

Cruce. En primer lugar se divide la población en dos grupos, de forma que para implementar la propia operación de cruce se eligirá un individuo de cada grupo. La operación de cruce que se utiliza se denomina *cruce cíclico* (*cycle crossover*) en la que, en cada uno de los dos individuos resultantes cada tarea se ha tomado de uno de los individuos de partida, encontrándose en la misma posición que tenía en dicho individuo de partida. En la Figura 2.4 se proporciona un esquema que ilustra la forma de realizar el cruce. Sólo se necesitan los números de tarea para realizarla.

Procedimiento de distribución de la carga. A través del algoritmo genético cuyos elementos se han descrito, se consigue realizar una asignación óptima de las tareas. No obstante, hay que tener en cuenta que en un procedimiento de distribución de carga hay que definir otros aspectos como la política de transferencia, la de información, etc. A continuación se consideran todos estos aspectos.

En primer lugar hay que definir cuándo iniciar el mecanismo de distribución de carga que, en este caso, se realiza a través de un algoritmo genético. Es decir, hay que definir la *política de transferencia*. Para ello hay que tener en cuenta que, dado que el procedimiento que estamos describiendo [ZOM01] realiza la asignación de tareas conforme van apareciendo

sin considerar la posibilidad de migración de tareas entre procesadores, si se ejecuta el mecanismo de distribución de tareas muy frecuentemente se puede sobrecargar a los procesadores. Por otro lado, si se asignan tareas con poca frecuencia pueden aparecer procesadores que se queden sin trabajo antes de que se vuelva a realizar una nueva asignación. En [ZOM01] se inicia una nueva asignación de tareas mediante el algoritmo genético cuando el procesador central detecta que un procesador ha terminado de procesar todas la tareas de su cola. De esta forma, es difícil que un procesador se mantenga ocioso, ya que se le asignarán nuevas tareas al ejecutar el algoritmo genético.

Para que el procesador central pueda tomar correctamente las decisiones de equilibrado de carga correspondientes es preciso que se disponga de información actualizada de la carga del sistema. Es decir, hay que definir una adecuada *política de información*. En este caso, el procesador central recoge información de todos los procesadores sólo cuando se envían las tareas a los correspondientes procesadores una vez el algoritmo genético ha realizado la asignación de tareas. Con esa información el procesador central actualiza la tabla de información de carga y, cada vez que transcurre una unidad de tiempo, comprueba la información de la tabla para determinar si alguna de las tareas asignadas puede haber terminado. Además, se decrementa en una unidad el tiempo asignado a cada una de las tareas que se están ejecutando en cada procesador.

Una cuestión importante que hay que considerar es el valor de los umbrales de carga. Estos valores afectan al resultado del algoritmo genético puesto que se tienen en cuenta a la hora de evaluar la función de idoneidad, además, si se utilizan umbrales muy bajos para indicar la condición de sobrecarga se producirá la distribución de carga con mucha frecuencia, y si estos umbrales son demasiado elevados, la distribución de carga deja de ser efectiva. En el procedimiento de [ZOM01] los umbrales se ajustan cada vez que se asignan las tareas mediante el algoritmo genético. Para determinar los umbrales adecuados se tiene en cuenta la carga media del sistema. Esta carga media se calcula a partir de la suma de la carga total del sistema en el momento en que el algoritmo genético va a iniciar la asignación de tareas y la nueva carga en la que se incluyen las nuevas tareas incorporadas en la ventana deslizante. La suma obtenida se divide por el número de procesadores. Obviamente, si tras la asignación de tareas realizada por el algoritmo genético la carga de los procesadores es igual a la carga media el sistema está equilibrado. No obstante, es muy restrictivo utilizar este valor de la

carga media como umbral ya que es muy difícil de alcanzar, sobretodo en sistemas grandes. Así, es más realista relajar las condiciones de carga equilibrada y utilizar dos umbrales, para indicar si el procesador está débil o fuertemente cargado, y dar más flexibilidad al procedimiento de asignación. De esta forma, cada procesador se puede encontrar débilmente cargado (su carga es inferior al umbral inferior, T_L), fuertemente cargado (su carga es superior al umbral superior, T_H), o normalmente cargado (su carga está entre los dos umbrales), y sólo necesitaría informar al procesador central cada vez que cambio de un estado a otro. Por tanto, disponer de una *política adecuada de localización* mediante umbrales es muy útil de cara a reducir las comunicaciones en el procedimiento de distribución de la carga. Por otra parte, los umbrales pueden ser *fijos*, si no cambian can la carga media del sistema, o *adaptativos*, que tienen en cuenta dicha carga. En este caso, cada vez que se produce un cambio en los umbrales el procesador central debe enviarlos al resto de los procesadores. En [ZOM01], los umbrales se fijan a partir de las expresiones $T_H=H\times L_{med}$ y $T_L=L\times L_{med}$, donde H es menor que uno y representa en qué cantidad la carga del procesador en cuestión puede ser menor que la carga media para que se considere poco cargado, y L es mayor que uno, indicando la cantidad en que la carga del procesador puede ser mayor que la carga media antes de considerarse sobrecargado. En [ZOM01], L=0.8 y H=1.2, es decir, que un procesador se considera poco cargado o sobrecargado con un 20% por debajo o por arriba de la carga media, respectivamente. Para ese porcentaje es para el que se han obtenido los mejores resultados.

Para terminar esta descripción detallada de un procedimiento centralizado de distribución dinámica de carga basado en algoritmos genéticos proporcionamos algunas de las conclusiones que se extraen en [ZOM01] en relación con sus prestaciones. En primer lugar, se realiza una comparación del procedimiento basado en algoritmos genéticos con el algoritmo FF [HU82] considerando un aumento gradual en el número de tareas a distribuir. En ambos algoritmos se observa un incremento lineal en el tiempo de ejecución total a medida que se incrementa el número de tareas. No obstante, el procedimiento genético se comporta mejor que el FF ya que el tiempo de ejecución es menor para dicho procedimiento y, además, la diferencia de tiempos se va haciendo cada vez mayor con el incremento de tareas. En cuanto a la utilización de los procesadores, mientras que el procedimiento genético consigue porcentajes del 85-99%, el algoritmo FF obtiene valores del 80-91%. Con respecto a la

dependencia de las prestaciones con respecto a los parámetros del procedimiento genético se puede concluir que:

- Al aumentar el tamaño de la ventana deslizante el tiempo de ejecución mejora, aunque hay que tener en cuenta que los incrementos de ventana se acompañan de incrementos en el número de procesadores para mantener una cierta relación entre el número de tareas y de procesadores (si el número de procesadores fuera muy elevado no habría mucho trabajo para los procesadores). Lo que se puede concluir es que el incremento de carga se redistribuye correctamente entre los procesadores adicionales que se incorporan. Sin embargo, al aumentar la ventana el promedio de utilización de los procesadores empeora: en el procedimiento genético, la mejora del tiempo de ejecución aumentando el número de tareas que se consideran (ventanas mayores) y el número de procesadores utilizado se hace a costa de reducir la utilización de los procesadores.
- A medida que, en cada ejecución del algoritmo genético, el número de generaciones se incrementa de 5 a 20, el tiempo de ejecución total se reduce al conseguirse distribuciones de mejor calidad. Por encima de 20 no se aprecian mejoras, y el tiempo asociado a su procesamiento afectaría negativamente. La mejora que se observa en la utilización de los procesadores tiene un comportamiento similar.
- El cambio en el tamaño de la población (se incrementa de 5 a 40 individuos) no afecta prácticamente al tiempo de ejecución total y conlleva un coste de procesamiento elevado. En cambio, la utilización de los procesadores sí mejora debido a que se consigue una mejor exploración del espacio de búsqueda.

En la sección siguiente se describe un procedimiento centralizado de distribución dinámica de carga que hemos desarrollado. Evaluaremos las prestaciones de este procedimiento y las compararemos con una implementación del procedimiento descrito en [ZOM01] para, de esta forma, ilustrar los posibles beneficios que aporta un procedimiento de distribución de carga basado en computación evolutiva.

2.3. El algoritmo EFP de distribución de carga dinámica y centralizada.

Como se ha indicado más arriba, los procedimientos para la distribución dinámica de carga pueden ser centralizados o distribuidos. En un procedimiento centralizado, todas las decisiones respecto a la distribución de la carga se toman en un procesador central, mientras que en un procedimiento distribuido todos los procesadores implementan una parte del procedimiento, participando en la toma de decisiones para el reparto de la carga. A continuación se presenta un procedimiento centralizado para la distribución dinámica de la carga. Este procedimiento se basa en la denominada estrategia de ofertas [ZHO88] y se puede utilizar para una red computadores heterogénea.

La estrategia de ofertas [ZHO88] considera que, a la hora de distribuir la carga de una aplicación paralela, puede considerarse que los procesadores compiten por adquirir carga y recursos, y que por tanto, el problema puede plantearse como una subasta en la que los procesadores pujan por la carga. Así, un procesador iniciaría una subasta de carga o de recursos solicitando ofertas a los procesadores del sistema. Después, se seleccionaría la mejor de las ofertas recibidas y el procesador correspondiente recibiría la carga y el recurso. Los procesadores que inician subastas pueden ser procesadores sobrecargados que necesitan deshacerse de parte de su carga, o procesadores poco cargados que ponen sus recursos desocupados a disposición de otros procesadores que disponen de carga. En el caso de que la subasta se plantee un procesador sobrecargado, éste selecciona una parte de su carga para ser transferida, mientras que los procesadores poco cargados u ociosos emiten sus ofertas según su situación de carga actual y los recursos de que disponen. Transcurrido de un intervalo de tiempo establecido, el procesador que realiza la subasta comprueba si hay alguna oferta adecuada y, si todavía está sobrecargado transfiere la carga al mejor procesador. Si no hay ninguna oferta adecuada, el procesador sobrecargado envía otra solicitud de ofertas a un conjunto distinto de procesadores. También es posible que los procesadores emitan ofertas para varias subastas. Entonces la estrategia debe garantizar que los procesadores sobrecargados transfieran su carga de manera que el sistema se mantenga estable.

En [CHE02] se propone un algoritmo de distribución de carga basado en la estrategia de ofertas descrita para un entorno de tipo GRID en el que se supone que hay computadores

interconectados a través de redes de área local que, a su vez se interconectan entre ellas configurando una plataforma heterogénea y distribuida geográficamente en un área de una extensión considerable. En este caso se considera una red de estaciones de trabajo dividida en dos niveles, un nivel WAN (*Wide Area Network*), o de red de área amplia, en el que un procedimiento distribuido global se encarga de repartir la carga, y un nivel LAN (*Local Area* Network), o de red local, con un procedimiento centralizado y local para distribuir la carga entre los computadores interconectados mediante una red de área local. Por tanto, en cada red de área local residen dos procedimientos de distribución de carga, uno distribuido global, y otro centralizado local.

En la siguiente sección se describe un procedimiento de distribución dinámica de carga que hemos desarrollado a partir de la estrategia de ofertas. Este procedimiento se comparará con el procedimiento basado en un algoritmo genético descrito en la sección anterior.

2.3.1. Descripción del algoritmo EFP

El algoritmo que hemos denominado algoritmo EFP (*Exploitation of the Fastest Processor*) es un algoritmo centralizado de distribución dinámica de la carga asociada a programas paralelos basados en el paradigma SPMD (*Single Program Multiple Data*). Con este procedimiento se pretende conseguir la máxima utilización y explotación de la potencia de los procesadores en un sistema paralelo, que puede ser heterogéneo. Además, el algoritmo EFP distribuye las tareas paralelas dinámicamente de manera que la aplicación paralela se ejecute en el menor tiempo posible. La Figura 2.5 se muestra el diagrama de flujo del procedimiento propuesto considerando que N proporciona una medida del tamaño del problema que aborda el programa paralelo, y que se pueden utilizar hasta P procesadores, P_0, P_1,..., P_{P-1}. El procesador más rápido es el que se selecciona como procesador central (si todos los procesadores son iguales se toma como central uno cualquiera), asignándosele el índice 0 (P_0). El resto de procesadores se ordenan en base a su capacidad de cómputo de forma que P_1 será el menos potente y P_{P-1} el más potente. Los pasos del procedimiento en el procesador central se describen a continuación:

1. Dividir el problema en un número de tareas igual a *G* e introducirlas en una cola. El valor de *G* se determina a partir de la expresión:

$$G(P) = \lceil n \times P \times rand() \rceil \times (P \times (P\text{-}1)/2)$$

donde $\lceil x \rceil$ es el menor número entero mayor o igual a *x*; *n* es un entero positivo entre 1 y 3; y *rand()* es una función que devuelve un real entre 0 y 1. El valor de *G* crece con el número de procesadores y debe ser menor que el tamaño del problema, N. Por lo tanto, cuanto más procesadores hay, mayor es el número de tareas que se crean y menor será el volumen de trabajo para cada una Es decir, y como es lógico, consideraremos una división del trabajo con una granularidad más fina cuanto mayor sea el número de procesadores. A través del entero *n* podemos prefijar distintos valores de granularidad para un valor fijo de *P*, según interese. En los experimentos que hemos realizado se han utilizado valores para *n* iguales a 1, 2, o 3, como se ha indicado antes.

2. El procesador central, P_0, realiza una distribución escalonada de las tareas entre los procesadores asignando tareas a cada procesador de forma que el procesador *i*-ésimo, para *i*=1,2,..,P-1, recibirá *i* tareas.

3. Una vez asignadas las tareas a los demás procesadores, y mientras éstos las están ejecutando, el procesador central ejecuta tareas de su cola de tareas y atiende las interrupciones provenientes de los demás procesadores. En el caso de que haya una interrupción.

- Se determina el procesador fuente de la interrupción.
- Se reciben los resultados de trabajo realizado por el procesador, junto con información relativa a la velocidad del procesador.
- Si la cola de tareas está vacía, se ejecuta el paso 5.
- Calcular el *factor de velocidad*, *S*, del procesador y enviarle un número de tareas proporcional a dicho factor. Es decir se envían más tareas cuanto más rápido es el procesador.

4. Si la cola de tareas no está vacía ir al paso 3.

5. El procesador central espera recibir los resultados de todos los procesadores que van concluyendo su trabajo y les envía una señal de final de procesamiento conforme van terminando.

En el diseño del algoritmo EFP se ha considerado que la plataforma de cómputo puede ser heterogénea, con procesadores (o computadores) diferentes en cuanto a su capacidad de cómputo. Para implementar el algoritmo es necesario disponer de información para poder decidir acerca de la potencia de los procesadores. Por ejemplo, en nuestra implementación se han utilizado tres parámetros para decidir acerca de la potencia de cómputo: la frecuencia de reloj del procesador, la capacidad de las caches internas, y la memoria RAM del nodo. Al procesador con más capacidad se le asigna el papel de procesador central, donde se va a ejecutar el procedimiento de distribución de carga. Al resto de procesadores se les asignan índices de menor a mayor capacidad de cómputo (el procesador P_1 es el menos potente, y así sucesivamente, hasta el P_{P-1}).

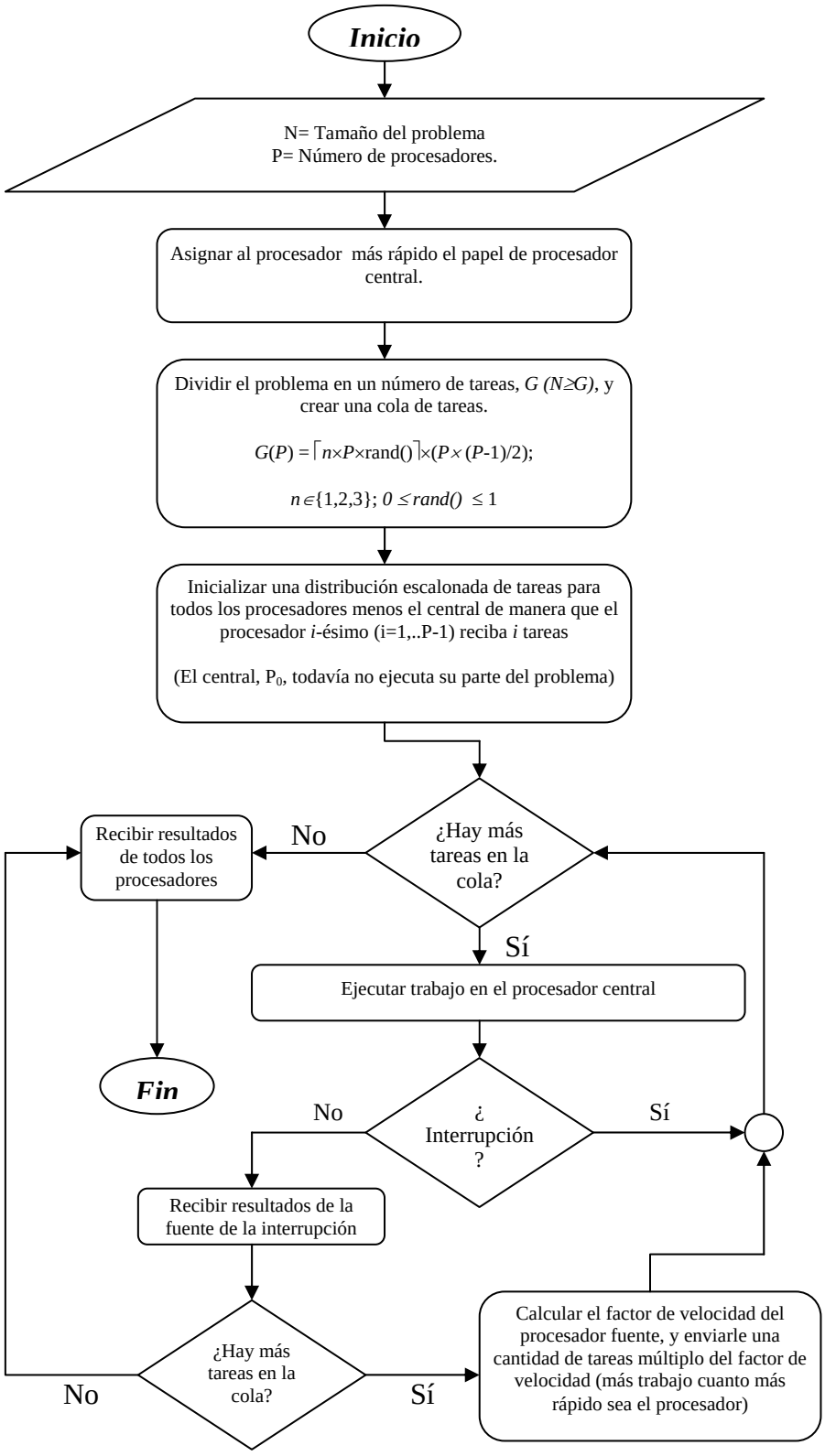


Figura 2.5: Organigrama del algoritmo EFP de distribución dinámica centralizada

Lo primero que hace el procesador central es calcular la *granularidad* y definir las tareas en base a esa granularidad. Las tareas se sitúan en una cola desde donde van a ser repartidas dinámicamente entre los restantes *P-1* procesadores, teniendo en cuenta su capacidad de cómputo. Para determinar la granularidad el procesador central tiene en cuenta el tamaño del problema, *N*, y el número de procesadores a utilizar (*P* procesadores, considerando también el procesador central). De esta forma, el problema se divide en un número, *G*, de tareas y crea una cola con esas tareas. Como se ha indicado más arriba ese número de tareas se define según la expresión $G(P) = \lceil n \times P \times rand() \rceil \times (P \times (P\text{-}1)/2)$, donde *n* es entero positivo igual a 1, 2, ó 3, y *rand()* devuelve un real entre 0 y 1. Como resulta evidente cuantos más procesadores haya mayor será *G*. Es decir, habrá más tareas puesto que existen más procesadores entre los que repartir el trabajo, y por tanto más fina será la granularidad. Se pueden controlar el valor de la granularidad a través del factor entero *n*.

El procesador central envía tareas a todos los procesadores en una primera *ronda*, a través de la que se distribuyen ($P(P\text{-}1)/2$) de trozos de trabajo. Es decir, en esta primera ronda, el procesador *i*-ésimo, para i=1,..,P-1, recibe *i* tareas. De esta forma, el procesador más lento nunca tendrá más trabajo que el más rápido. Por otra parte, mediante la distribución escalonada se reduce la probabilidad de que varios procesadores terminen simultáneamente sus tareas e interrumpan simultáneamente al procesador central para enviar resultados y solicitar más trabajo. Esto es importante, sobre todo si se tiene una red de tipo *bus compartido* donde se ocasionarían conflictos si se intentan realizar accesos simultáneos. Estos conflictos sobrecargan de la red y aumentan el tiempo de comunicación con lo que el rendimiento del sistema se ve afectado negativamente. Una vez se han distribuido las tareas, el procesador central continúa ejecutando tareas que toma de la cola de tareas, al tiempo que permite que los demás procesadores le interrumpan cuando necesitan mandar resultados y recibir más trabajo. Realmente, el procedimiento que se ejecuta en el procesador central simula las interrupciones mediante una comprobación periódica de mensajes de la existencia de mensajes de interrupción enviados por otros procesadores. Si se detecta una petición de interrupción de otro procesador, el procesador central ejecuta una rutina de servicio de interrupción a través de la cual se determina el procesador fuente de la interrupción y se reciben los resultados y demás datos que envíe dicho procesador. Concretamente, los procesadores determinan los tiempos correspondientes a computación, comunicación, y espera, y la cantidad de trabajo

realizado. Toda esta información se envía al procesador central para que decida el volumen de trabajo que va a enviar al procesador cuya interrupción está atendiendo. El procesador que realiza su trabajo más rápidamente recibirá una cantidad mayor de trabajo en el siguiente envío, y, al contrario, el procesador que realiza su trabajo más lentamente recibirá una cantidad menor de trabajo. Cuando ya no quedan tareas en la cola, el procesador central termina la tarea que está ejecutando, e irá recibiendo todos los resultados del resto de los procesadores del sistema a medida que vayan terminando.

A continuación se proporcionan los resultados experimentales obtenidos. A través de esos resultados se pone de manifiesto los experimentos que el procesador más rápido siempre realiza una cantidad de trabajo mayor, y que el tiempo en el que los procesadores terminan sus tareas es similar debido a la adecuada distribución que realiza el algoritmo. Es decir, la utilización de los procesadores para este procedimiento de distribución de carga es elevada.

2.4. Resultados experimentales

En esta sección se describen los experimentos y los resultados obtenidos por el algoritmo EFP para la distribución dinámica y centralizada de carga y se comparan con los que proporciona un procedimiento basado en un algoritmo genético [ZOM01].

Para realizar el trabajo experimental se ha utilizado un *cluster* de computadores con una cierta heterogeneidad. Concretamente, se trata de un cluster con nueve nodos, cuyas características se dan en la Tabla 2.1, conectados mediante una red *Fast Ethernet*. El programa paralelo se ha escrito en C, utilizando la biblioteca de paso de mensajes MPI [MPI94] para implementar las funciones de comunicación.

Tabla 2.1 : Características de los nodos del cluster utilizado

Número de nodos	Velocidad de CPU (MHZ)	Tamaño del caché (KB)	Memoria RAM (MB)
1	PIII 1000	512	118
8	PII 333	512	120

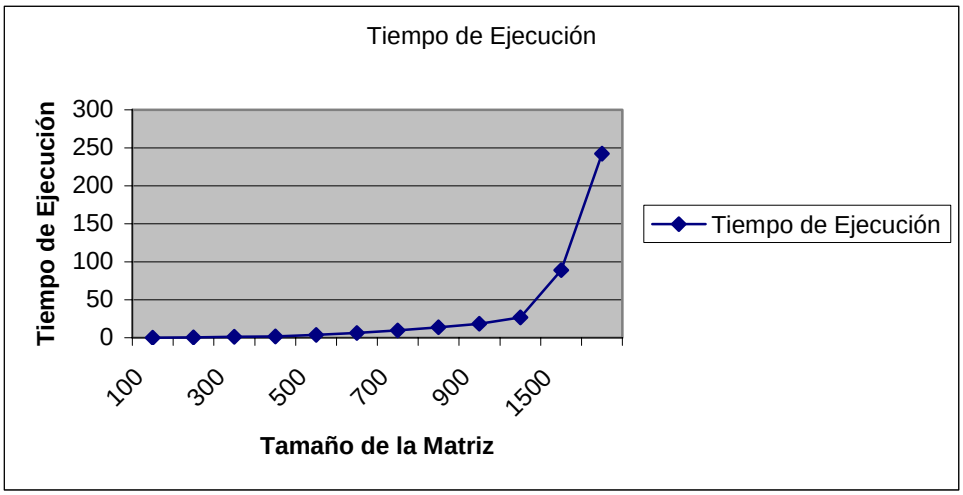


Figura 2.6: Incidencia del tamaño de la matriz en el tiempo de ejecución

Los experimentos se han realizado utilizando un programa paralelo de multiplicación de matrices como programa de prueba. Esta elección ha venido motivada por la sencillez del problema, además de ser la multiplicación de matrices una de las operaciones más frecuentes del álgebra lineal. Por otra parte, la carga de trabajo del algoritmo utilizado para la multiplicación de matrices es escalable y muy fácil de modificar. El tiempo de cómputo de la multiplicación de matrices es de orden $O(n^3)$, para matrices cuadradas $n \times n$, y se puede aumentar o reducir la carga de trabajo cambiando el número de filas/columnas, n.

La Figura 2.6 muestra el tiempo de ejecución de multiplicación de matrices en los nueve procesadores del cluster, utilizando el algoritmo EFP para distribuir la carga. Se puede observar el incremento de orden $O(n^3)$ en el tiempo de ejecución del programa.

A continuación se describen los resultados obtenidos y se comparan con los del procedimiento descrito en [ZOM01], en el que se utiliza un algoritmo genético para optimizar la distribución de carga. Con esto pretendemos ilustrar la viabilidad de la computación evolutiva en la distribución de carga dinámica y centralizada, comparando sus prestaciones con las del procedimiento que hemos desarrollado. Para realizar la comparación se ha considerado un número de generaciones igual a 10 para el algoritmo genético que se describe en [ZOM01], este número está dentro del rango que se considera en dicho artículo como rango razonable. En cuanto a los demás parámetros del procedimiento, se han utilizado los

valores para los que se han obtenido los mejores valores: Número de procesadores 9, tamaño de la ventana 18, número de generaciones 10, tamaño de la población 10, tamaño máximo de la tarea 20, coeficiente del umbral superior H 1,2 y coeficiente del umbral inferior L 0,8. Probabilidad de mutación 0,3 y probabilidad de cruce 0,7.

Las Tablas 2.2 y 2.3 muestran los datos correspondientes a la ejecución del algoritmo para matrices de tamaño 200x200 utilizando el algoritmo EFP y el descrito en [ZOM01], respectivamente. En cada tabla se indica, para cada uno de los nueve procesadores utilizados (0,1,...,8), el número de filas de la matriz que ha procesador, el tiempo que ha tardado en hacerlo, y los tiempos de cálculo, comunicación y ocio en que se distribuye el tiempo de ejecución. El tiempo de ejecución del programa paralelo es el máximo de los tiempos totales de todos los procesadores, que corresponde al tiempo del procesador central, al que hemos asignado el número 8. En el tiempo de ejecución del procesador central se incluye el tiempo necesario para calcular la distribución de tareas. Cuanto menor sea este tiempo más tiempo queda libre el procesador central para participar en el procesamiento de tareas del problema a resolver. Las Figuras 2.7 y 2.8 muestran gráficamente los datos que proporcionan, respectivamente, las Tablas 2.2 y 2.3.

Tabla 2.2: Ejecución con matrices 200×200 utilizando el algoritmo EFP

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	24	0.29	0.13	0.12	0.04
1	22	0.29	0.12	0.12	0.05
2	19	0.29	0.07	0.21	0.01
3	20	0.29	0.08	0.19	0.02
4	24	0.29	0.14	0.10	0.06
5	22	0.28	0.12	0.09	0.08
6	22	0.29	0.12	0.08	0.09
7	22	0.29	0.12	0.07	0.11
8	25	0.29	-	-	-

Tanto en las Figuras 2.7 y 2.8 como en las Tablas 2.2 y 2.3 se pone de manifiesto que el tiempo de comunicación es muy alto comparado con el tiempo de ejecución. Además, la

suma del tiempo de comunicación y tiempo de espera constituyen la mayor parte del tiempo total. Esto es debido a que, en este caso, el tamaño del problema es relativamente pequeño, y por lo tanto no existe mucha computación.

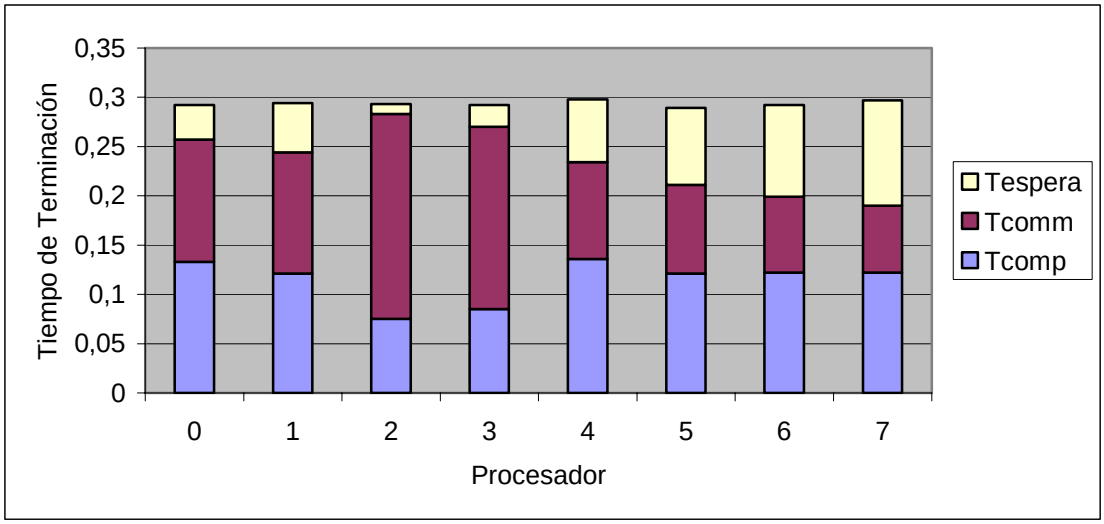


Figura 2.7 : Tiempo de cálculo, comunicación y ocio o espera para matrices 200×200 elementos utilizando el algoritmo EFP

Al comparar el procedimiento de distribución de carga EFP con el basado en el algoritmo genético [ZOM01] se observan mejores tiempos de ejecución totales para el procedimiento EFP (0.299 segundos frente a 0.600 segundos), además, mientras que la diferencia de tiempo entre el procesador que termina el primero y el que termina el último es de un 20.0% en el caso del procedimiento descrito en [ZOM01], es de sólo un 3.3% para el procedimiento EFP. Si consideramos únicamente el tiempo de cálculo, la diferencia entre el procesador que ha realizado más trabajo y el que menos es de un 36.1% en el caso del procedimiento EFP y de, nada menos que un 85.7% en el caso del procedimiento genético. No obstante, si observamos las Figuras 2.7 y 2.8, comprobamos que, mientras que en el caso del procedimiento EFP los tiempos de comunicación son los que representan un mayor porcentaje del tiempo de sobrecarga, en el caso del procedimiento genético son los tiempos de espera los que consumen un mayor porcentaje. Esto pone de manifiesto el peor equilibrado de carga que se consigue.

Por otra parte, si consideramos el volumen de trabajo que ha completado el procesador número 8, medido a partir del número de filas de la matriz que se han procesado (segunda columna de las tablas) vemos que éste es mayor en el procedimiento EFP que en el procedimiento genético (25 filas frente a 16). Esta situación, que se verifica para todos los tamaños de matriz que hemos utilizado en nuestros experimentos, pone de manifiesto la menor carga que supone para el procesador central el procedimiento de distribución EFP.

Tabla 2.3: Ejecución de matrices 200×200 utilizando el algoritmo [ZOM01]

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de ocio (s)
0	14	0,52	0,02	0,02	0,48
1	35	0,48	0,16	0,01	0,30
2	33	0,50	0,15	0,01	0,34
3	20	0,47	0,06	0,08	0,32
4	30	0,48	0,11	0,01	0,36
5	18	0,52	0,09	0,16	0,27
6	21	0,54	0,10	0,08	0,36
7	13	0,51	0,06	0,08	0,37
8	16	0,60	-	-	-

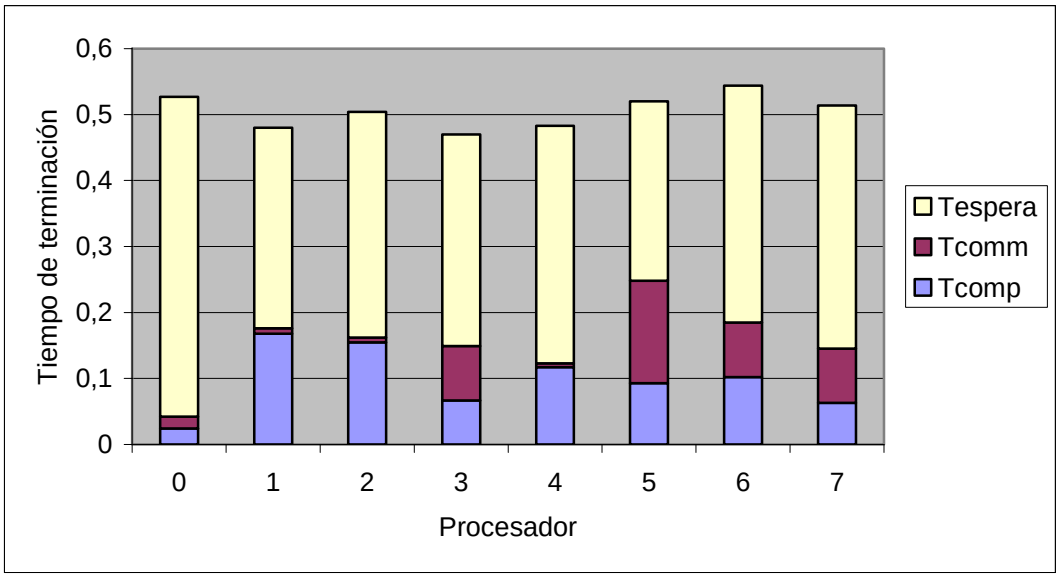


Figura 2.8: Tiempo de cálculo, comunicación y ocio o espera para matrices 200×200 con el algoritmo descrito en [ZOM01]

Las Tablas 2.4 y 2.5 proporcionan los datos correspondientes a la ejecución del programa de multiplicación de matrices para matrices de 400x400 con el algoritmo de distribución de carga EFP y con el presentado en [ZOM01]. Como en el caso de matrices de 200x200, para cada uno de los nueve procesadores utilizados se indica el tiempo total de ejecución y los tiempos de computación, comunicación y ocio o espera, junto con el número de filas que ha procesado cada procesador, para dar una idea de la calidad de la distribución de tareas realizada. Los tiempos que se indican en las Tablas 2.4 y 2.5 se muestran gráficamente en las Figuras 2.9 y 2.10, respectivamente.

Tabla 2.4: Ejecución de matrices 400×400 utilizando el algoritmo EFP

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	38	1.85	1.45	0.25	0.14
1	37	1.83	1.42	0.20	0.20
2	46	1.80	1.22	0.54	0.03
3	40	1.79	1.24	0.45	0.09
4	32	1.82	1.37	0.20	0.25
5	36	1.86	1.38	0.17	0.31
6	34	1.80	1.29	0.14	0.36
7	34	1.84	1.29	0.12	0.42
8	103	1,88	-	-	-

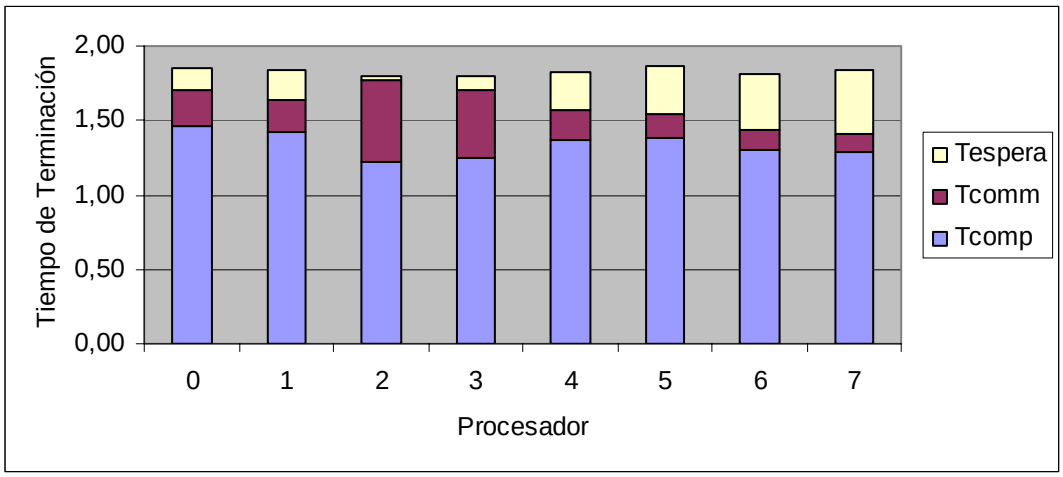


Figura 2.9 : Tiempo de computación, comunicación y espera para tamaño de matrices 400×400 elementos utilizando el algoritmo EFP

En este caso, si comparamos el tiempo de ejecución total, observamos que en ambos casos se consiguen tiempos similares, incluso se obtiene un tiempo algo menor para la distribución de carga realizada con el procedimiento genético descrito en [ZOM01] que con el procedimiento EFP (1.85 segundos frente a 1.88 segundos). Además, los tiempos de comunicación y espera con el procedimiento descrito en [ZOM01] son menores que los que se obtienen con el procedimiento EFP. Hay que tener en cuenta que, en procedimiento basado en el algoritmo genético, la evaluación de la carga de los procesadores no necesita el volumen de comunicación que utiliza el procedimiento EFP. Así, se recuerda que en el algoritmo descrito en [ZOM01] el procesador central recoge información sólo cuando se envían las tareas a los correspondientes procesadores una vez que el algoritmo genético ha realizado la asignación de tareas. Con la información actualiza la información de la carga. El procesador central modifica la tabla de que dispone cada vez que transcurre una unidad de tiempo, decrementando en una unidad el tiempo asignado cada tarea que se está ejecutando en cada procesador y comprobando se alguna puede haber terminado. El resto de procesadores no interrumpen su trabajo para comunicarse con el procesador central, únicamente comprueban si éste les ha enviado trabajo, cada vez que terminan una tarea. En el procedimiento EFP que hemos implementado se podría llevar a cabo un planteamiento similar, y de esta forma, se habría reducido considerablemente el tiempo de comunicación que precisa el procedimiento. No obstante, hay que tener en cuenta que, este planteamiento es menos tolerante a cambios en las condiciones en que se ejecutan las tareas, y a la falta de un conocimiento preciso del tiempo de ejecución previsible para cada tarea. En cualquier caso, como hemos mostrado, los tiempos de ejecución totales son muy parecidos.

Tabla 2.5: Ejecución de matrices 400×400 utilizando el algoritmo descrito en [ZOM01]

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	32	1,28	1,03	0,03	0,22
1	58	1,81	1,66	0,02	0,13
2	46	1,47	1,32	0,01	0,14
3	49	1,12	0,96	0,01	0,15
4	47	1,27	1,11	0,01	0,15
5	47	1,66	1,50	0,01	0,15
6	47	1,50	1,34	0,01	0,15
7	35	1,15	0,99	0,01	0,15
8	39	1,85	-	-	-

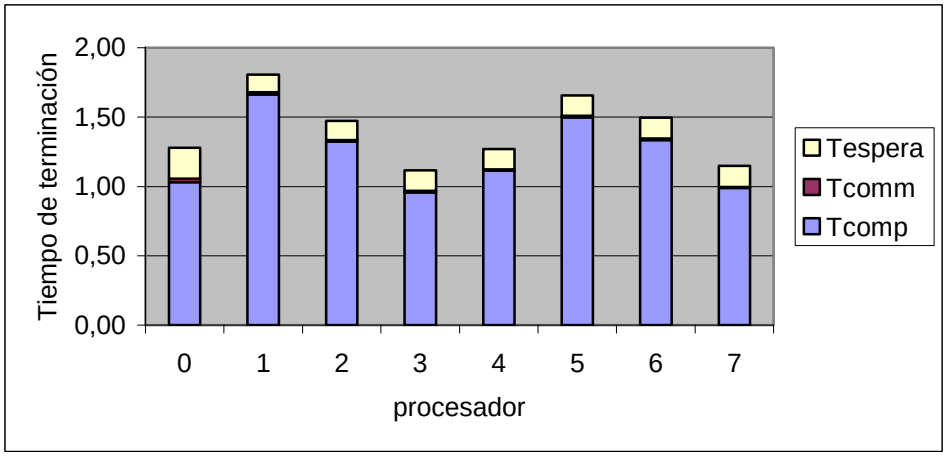


Figura 2.10: Tiempo de computación, comunicación y espera para matrices de 400×400 con algoritmo descrito en [ZOM01]

Por otra parte, si observamos las Figuras 4.9 y 4.10, podemos ver como las diferencias entre los tiempos de ejecución de los procesadores son mayores en el caso del procedimiento de [ZOM01] que con la distribución de carga basada en el algoritmo EFP: 38.1% frente a un 3.8%, es decir, hay un factor de 10 de diferencia. Si se tienen en cuenta sólo los tiempos de computación, en el procedimiento descrito en [ZOM01] hay una diferencia del 42.2%, mientras que en el procedimiento EFP hay una diferencia del 16.2%. También para este tamaño de problema se puede concluir que el procedimiento EFP proporciona una mejor distribución de la carga pero, al necesitar más tiempo de comunicación, esa ventaja se ve

anulada. De esta manera, ilustramos la importancia que la sobrecarga que introduce el procedimiento de distribución de carga tiene en las prestaciones finales.

Tabla 2.6: Ejecución de la multiplicación de matrices 800×800 utilizando el algoritmo EFP

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	75	13.66	12.16	0.92	0.57
1	76	13.99	12.30	0.90	0.79
2	99	13.35	11.04	2.17	0.13
3	84	13.26	11.18	1.73	0.34
4	65	13.51	11.74	0.76	1.01
5	72	13.72	11.65	0.83	1.23
6	69	13.50	11.17	0.87	1.45
7	68	13.44	11.01	0.74	1.68
8	192	14,02	-	-	-

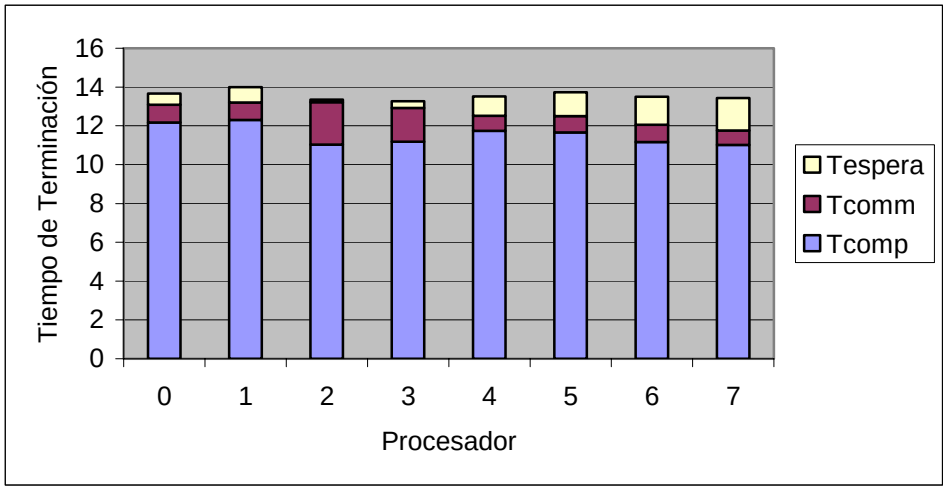


Figura 2.11 : Tiempo de computación, comunicación y espera para matrices de 800×800 utilizando el algoritmo EFP

Las Tablas 2.6 y 2.7 proporcionan los resultados experimentales para un tamaño de matrices de 800x800, respectivamente, para el procedimiento de distribución EFP y para el procedimiento descrito en [ZOM01]. Las Figuras 2.11 y 2.12 muestran gráficamente esos resultados. Las conclusiones que se pueden extraer de aquí son similares a las que hemos

indicado más arriba. El tiempo de ejecución con nueve procesadores es mejor para el procedimiento EFP que para el procedimiento basado en el algoritmo genético (14.02 segundos frente a 16.05 segundos). No obstante, los tiempos correspondientes a la sobrecarga de comunicación y al tiempo de ocio o espera son menores para el procedimiento genético descrito en [ZOM01]. La distribución de carga que consigue el procedimiento EFP es mejor que la que se proporciona con el procedimiento de [ZOM01]. Por ejemplo, la diferencia en tiempos de ejecución entre el procesador que termina el primero y el que termina el último es del 45.0% para el procedimiento descrito en [ZOM01] y de sólo el 4.6% para el procedimiento EFP. Si se tienen en cuenta sólo los tiempos de cómputo las diferencias son del 46.9% para el procedimiento genético y del 10.5% para el procedimiento EFP.

Tabla 2.7: Ejecución de la multiplicación de matrices 800×800 utilizando el algoritmo descrito en [ZOM01]

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	82	8,88	8,47	0,03	0,38
1	103	15,37	14,88	0,18	0,31
2	104	15,40	14,98	0,05	0,38
3	84	8,81	8,21	0,18	0,42
4	124	16,02	15,46	0,15	0,42
5	60	10,66	10,13	0,13	0,4
6	81	12,12	11,61	0,10	0,41
7	78	11,76	11,18	0,17	0,42
8	84	16,05	-	-	-

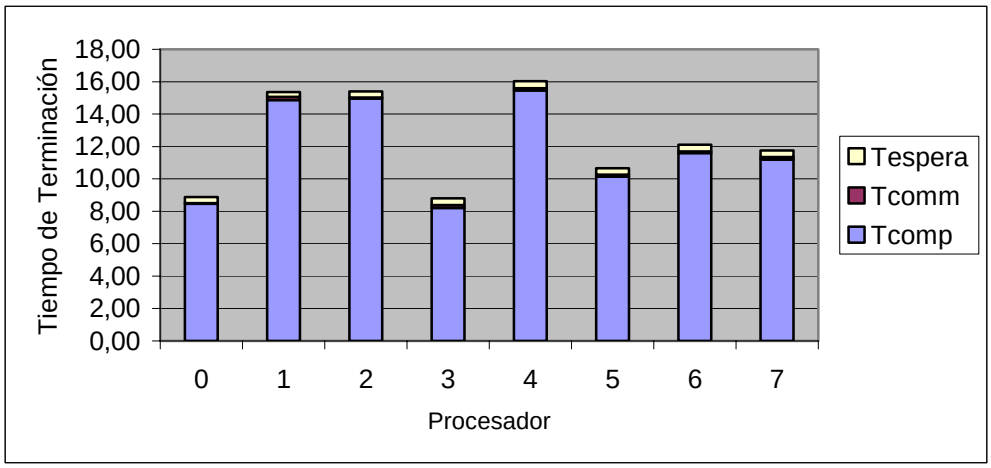


Figura 2.12: Tiempo de computación, comunicación y espera para matrices 800×800 utilizando el algoritmo descrito en [ZOM01]

Para concluir con esta comparación de prestaciones con distintos tamaños de problema consideramos el caso de matrices de 1000x1000. Las Tablas 2.8 y 2.9 muestran, respectivamente, los tiempos para el procedimiento EFP y para el procedimiento genético descrito en [ZOM01]. Las Figuras 2.13 y 2.14 muestran gráficamente los resultados de las Tablas 2.8 y 2.9, respectivamente. Se pueden extraer las mismas conclusiones que en el caso anterior. El tiempo de ejecución total es menor para el procedimientos EFP que para el procedimiento genético: 26.86 segundos frente a 31.21. Sin embargo, los tiempos de comunicación y de espera son menores para el procedimiento genético. En cualquier caso, el porcentaje de tiempo de comunicación y tiempo de procesamiento se van reduciendo a medida que aumenta el tamaño del problema tanto para el procedimiento EFP como para el procedimiento genético. En cuanto a las diferencias entre los tiempos de ejecución y de cómputo en los procesadores, en el procedimiento EFP la diferencia en tiempo de ejecución entre el procesador que acaba primero y el que acaba el último es de un 6.98% y la diferencia entre los tiempos de cómputo entre los procesadores es de un 12.39%. Estos valores son menores que los que se obtienen con el procedimiento genético: un 44.63% para las diferencias entre los tiempos de ejecución, y 45.61% para las diferencias entre los tiempos de computación.

Tabla 2.8: Ejecución de multiplicación de matrices 1000×1000 utilizando el algoritmo EFP

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	84	24.99	23.41	0.69	0.88
1	89	26.78	24.80	0.74	1.23
2	122	26.00	23.45	2.34	0.20
3	106	25.66	24.21	0.91	0.54
4	77	26.24	23.94	0.71	1.58
5	83	25.69	23.12	0.64	1.93
6	84	26.33	23.40	0.63	2.28
7	78	24.91	21.73	0.54	2.64
8	277	26.86	-	-	-

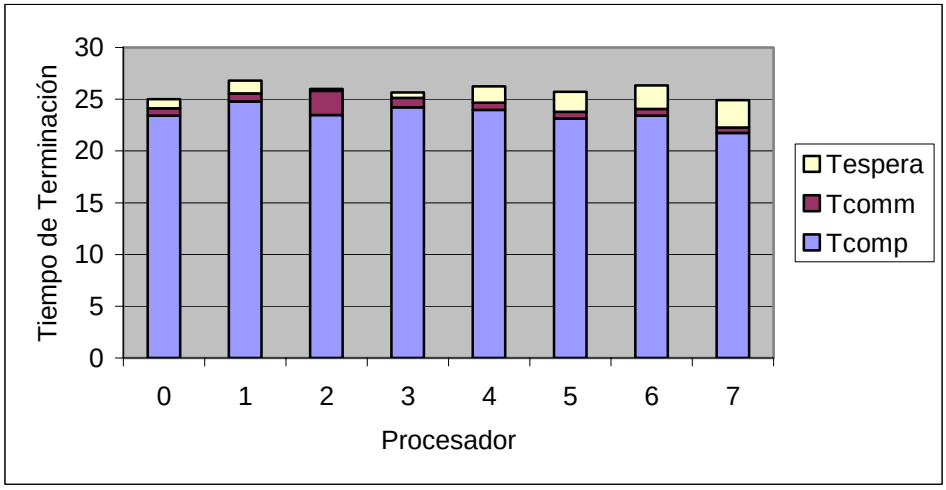


Figura 2.13 : Tiempo de computación, comunicación y espera para matrices 1000×1000 utilizando el algoritmo EFP

Tabla 2.9: Ejecución de multiplicación de matrices 1000×1000 utilizando el algoritmo descrito en [ZOM01]

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	91	21,65	20,97	0,13	0,55
1	133	31,12	30,50	0,20	0,42
2	105	24,64	24,06	0,14	0,44
3	161	25,88	25,09	0,27	0,52
4	135	27,97	27,24	0,19	0,54
5	69	19,49	18,84	0,14	0,51
6	72	17,23	16,59	0,11	0,53
7	120	28,38	27,63	0,20	0,54
8	114	31,21	-	-	-

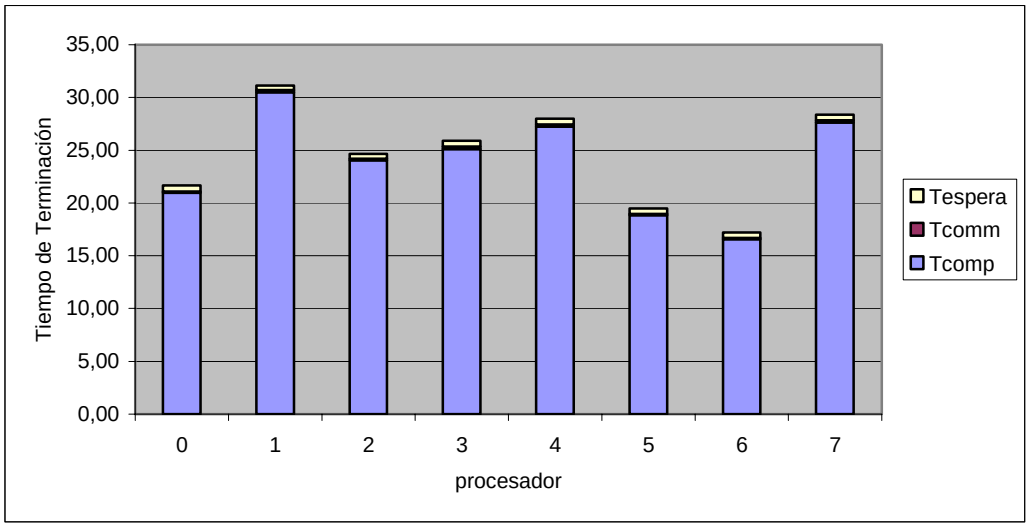


Figura 2.14: Tiempo de computación, comunicación y espera para matrices 1000×1000 utilizando el algoritmo descrito en [ZOM01]

Al aumentar de 10 a 15 el número de generaciones que utiliza el algoritmo genético del procedimiento descrito en [ZOM01] se consigue una mejor distribución de la carga que, a pesar del incremento que se produce en la sobrecarga asociada a la decisión respecto a la distribución de la carga, permite reducir el tiempo de ejecución total del algoritmo paralelo. Así, se tienen 29.85 segundos en lugar de 31.21 segundos (un 4.36%). Además, la diferencia

entre los tiempos de ejecución del procesador que termina primero y el que termina el último es del 19.77% y la diferencia entre los tiempos de computación del que realiza más trabajo y el que realiza menos de 37.48%. Se tiene, por tanto una mejora apreciable con respecto al procedimiento que utiliza 10 generaciones. No obstante, las prestaciones del procedimiento EFP siguen siendo mejores que las del procedimiento descrito en [ZOM01] con 15 generaciones.

Tabla 2.10: Ejecución de matrices 1000×1000 utilizando el algoritmo descrito en [ZOM01] con 15 generaciones

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de Comunicación (s)	Tiempo de espera (s)
0	104	27,49	23,90	0,09	3,50
1	144	29,67	27,04	0,17	2,47
2	119	28,93	27,30	0,16	1,47
3	100	24,16	15,60	0,21	8,34
4	85	23,77	17,11	0,14	6,52
5	97	28,04	26,41	0,15	1,48
6	119	28,98	27,37	0,12	1,49
7	103	26,37	23,69	0,19	2,49
8	129	29,85	-	-	-

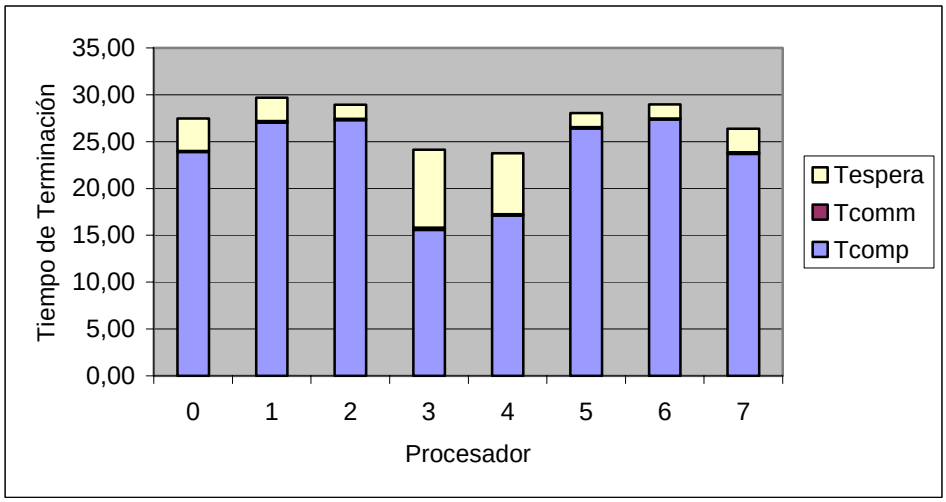


Figura 2.15: Tiempo de computación, comunicación y espera para matrices 1000×1000 con el procedimiento descrito en [ZOM01] y 15 generaciones

Tabla 2.11: Ejecución de matrices 1000×1000 utilizando el algoritmo descrito en [ZOM01] con 20 generaciones

Procesador	Filas ejecutadas	Tiempo total (s)	Tiempo de computación (s)	Tiempo de comunicación (s)	Tiempo de espera (s)
0	126	30,77	29,06	0,14	1,56
1	119	29,94	27,31	0,15	2,48
2	130	30,50	29,80	0,21	0,49
3	83	23,68	15,96	0,16	7,56
4	94	25,66	18,94	0,17	6,55
5	125	32,82	32,10	0,17	0,55
6	88	25,91	20,21	0,15	5,55
7	111	27,31	25,58	0,17	1,56
8	124	32,98	-	-	-

En la Tabla 2.11 se muestran los resultados obtenidos cuando se ejecutan 20 generaciones del algoritmo genético que calcula la distribución de tareas en el procedimiento descrito en [ZOM01]. Como se observa, en este caso el tiempo de ejecución es mayor que para 10 y 15 generaciones. La diferencia entre el tiempo del procesador que termina el primero y el que termina el último es de un 27.85%, y la diferencia entre los tiempos de computación del procesador más cargado y del menos cargado es del 50.28%. Los resultados, en este caso son peores que los obtenidos para un número menor de generaciones.

Tabla 2.12: Tiempos de ejecución, ganancia de velocidad y eficiencia para el procedimiento EFP y el procedimiento genético descrito en [ZOM01]

Numero de procesadores	Tiempo de ejecución de EFP	Tiempo de ejecución [ZOM01]	Ganancia de velocidad EFP	Ganancia de velocidad [ZOM01]	Eficiencia EFP	Eficiencia [ZOM01]
1	75,00	75,00	1,00	1,00	1,00	1,00
2	55,00	67,08	1,36	1,12	0,68	0,56
3	46,00	62,98	1,63	1,19	0,54	0,40
4	41,00	58,14	1,83	1,29	0,46	0,32
5	36,00	47,60	2,08	1,58	0,42	0,32
6	34,00	45,04	2,21	1,67	0,37	0,28
7	31,00	43,36	2,42	1,73	0,35	0,25
8	29,00	33,10	2,59	2,27	0,32	0,28
9	27,00	30,79	2,78	2,44	0,31	0,27

En la Tabla 2.12 se comparan las ganancias de velocidad y las eficiencias para los dos procedimientos que comparamos considerando distinto número de procesadores, en el caso del producto de matrices 1000x1000. Además, las Figuras 2.16, 2.17, y 2.18 comparan, respectivamente, los valores de los tiempos de ejecución, las ganancias, y las eficiencias, para esos procedimientos, a medida que aumenta el número de procesadores. A partir de esos resultados, resulta evidente que el procedimiento de distribución EFP mejora al procedimiento genético descrito en [ZOM01].

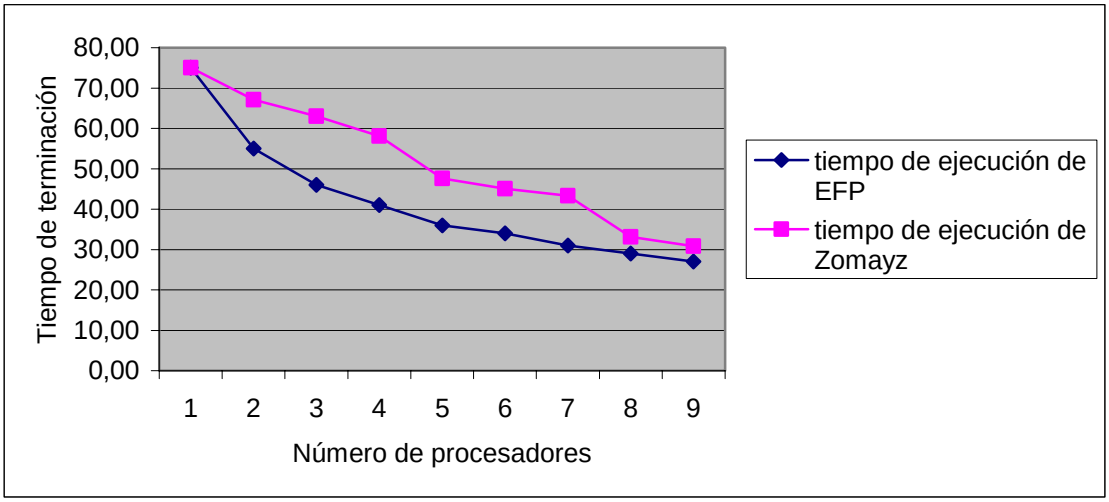


Figura 2.16: Comparación de la evolución del tiempo de ejecución frente al número de procesadores, P, utilizando para matrices 1000×1000.

En cuanto a la evolución de la ganancia de velocidad que se muestra en la Figura 2.17, como se observa, se produce un crecimiento aproximadamente lineal para dicha ganancia (con una pendiente mayor para el algoritmo EFP). Como se puede comprobar los valores de la ganancia de velocidad son muy bajos. Esto se debe a que se comparan los tiempos de ejecución paralela con el tiempo de ejecución secuencial en el mejor procesador de la plataforma que, como se ha indicado más arriba es heterogénea.

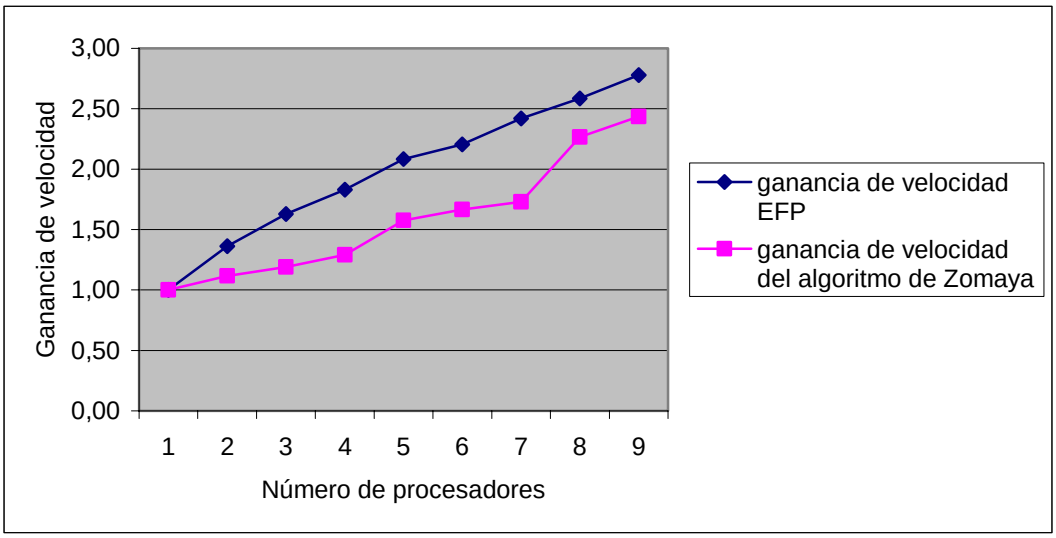


Figura 2.17: Comparación de la evolución de la ganancia de velocidad frente al número de procesadores, P, utilizando para matrices 1000×1000.

El procedimiento genético podría obtener distribuciones de carga más equilibrada entre los procesadores si se deja que ejecute más iteraciones. No obstante, esto ocasionaría un aumento en el tiempo de sobrecarga que ocasiona el procedimiento de distribución de carga, con lo que posiblemente no se mejorarían los tiempos de ejecución finales.

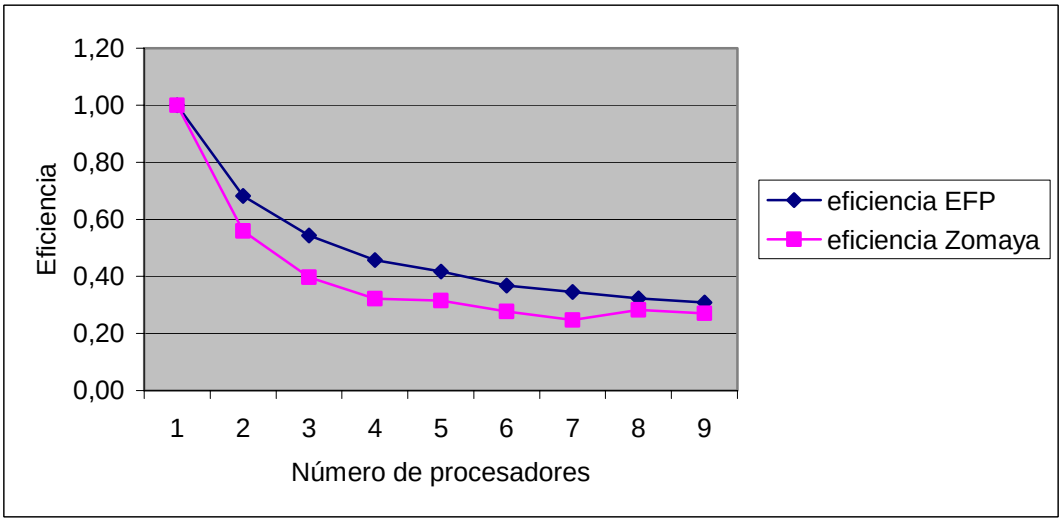


Figura 2.18: Comparación de la evolución de la eficiencia frente al número de procesadores, P, utilizando para matrices 1000×1000.

2.5 Conclusiones

En este capítulo se ha abordado la distribución dinámica centralizada de la carga. En este caso existe un procesador central que es el que toma las decisiones respecto a la forma de repartir la carga entre los procesadores a partir de la información que recibe del resto de procesadores de la plataforma. El objetivo del capítulo ha sido ilustrar el uso de la computación evolutiva para abordar esta forma de equilibrar la carga entre los procesadores. Para ello, tras hacer una presentación detallada de las características generales de la distribución dinámica centralizada en la Sección 2.1, en la Sección 2.2 se ha descrito un algoritmo de distribución dinámica de carga centralizado que se describe en [ZOM01] y que constituye uno de los mejores ejemplos de este tipo de procedimientos de equilibrado de carga.

Para comparar el procedimiento de distribución genético de [ZOM01] hemos utilizado un procedimiento centralizado de distribución de carga basado en la estrategia de ofertas que hemos desarrollado, y que se describe en la Sección 2.3. Este procedimiento se ha denominado procedimiento EFP. La estrategia de ofertas es una de las dos estrategias más frecuentemente utilizadas para la distribución de carga dinámica y centralizada, y como hemos comprobado a partir de los resultados obtenidos por nuestro procedimiento es capaz de proporcionar un buen equilibrado de la carga, consiguiendo que prácticamente todos los procesadores terminen en tiempos similares. Además, el procedimiento de carga desarrollado se ha manifestado muy eficaz en plataformas heterogéneas, donde se han realizado los experimentos.

La comparación experimental que se ha llevado a cabo entre el procedimiento genético descrito en [ZOM01] y el procedimiento EFP se describe en la Sección 2.4 para el caso del producto de matrices, y utilizando una plataforma heterogénea. Los resultados obtenidos muestran que el procedimiento EFP es capaz de obtener mejores tiempos de ejecución que el procedimiento genético para prácticamente todos los tamaños de matrices considerados (para matrices 400x400 los tiempos son mejores en el caso del procedimiento genético pero los valores obtenidos por los dos procedimientos son muy próximos). El procedimiento genético no es capaz de obtener distribuciones de carga muy equilibradas, si bien, la reducida sobrecarga de comunicación que presenta en comparación con el procedimiento EFP hace que las diferencias entre los resultados no sean demasiado abultadas en cuanto a los tiempos de

ejecución de los procedimientos paralelos. No obstante, hay que tener en cuenta que la forma de reducir la comunicación en el procedimiento descrito en [ZOM01] se basa en una suposición respecto a la potencia de los procesadores, que no tiene en cuenta posibles eventualidades con respecto a variaciones en las características de los nodos de procesamiento. Por otra parte, la sobrecarga asociada a la determinación de la mejor distribución en el procedimiento genético consume una mayor cantidad de recursos en el procesador central, como así lo pone de manifiesto la menor cantidad de tareas que puede ejecutar con respecto al procedimiento EFP. Es posible mejorar las prestaciones del procedimiento genético hasta un cierto punto si se aumenta el número de generaciones que emplea el algoritmo genético para calcular la distribución de tareas. No obstante, esta tendencia desaparece a partir de un número de generaciones (entorno a 20 en nuestros experimentos) porque la sobrecarga que implica el aumento de cálculo asociado al algoritmo genético supera a la mejora que se consigue con la distribución de la carga obtenida.

Consideramos que el uso de los procedimientos genéticos para la distribución de carga tiene más sentido en aquellas aplicaciones en las que se tengan tareas con tiempos de procesamiento muy diferentes, donde sea muy difícil prefijar una política de distribución de tareas razonable a priori.

En cualquier caso, hay que tener en cuenta que los procedimientos centralizados presentan un cuello de botella en el procesador central. Este cuello de botella limita la escalabilidad de los algoritmos paralelos que lo utilizan, de forma que no resulta una opción viable en el caso de plataformas con una gran cantidad de procesadores distribuidos, donde además los costes de comunicación pueden ser elevados entre procesadores más o menos alejados. En los dos siguientes capítulos se considera el equilibrado dinámico y distribuido de la carga, y se considera el uso de los algoritmos genéticos para estudiar las características de las distintas alternativas propuestas para implementar esta forma de equilibrado de carga.

Capítulo 3

Una taxonomía para las estrategias distribuidas de equilibrado dinámico de carga

Este capítulo y el siguiente se dedican a los procedimientos distribuidos para la distribución dinámica de carga. A medida que aumenta el número de procesadores y computadores que pueden utilizarse para configurar una plataforma de cómputo, que proporcione el nivel de prestaciones que requieren determinadas aplicaciones, los procedimientos centralizados de distribución de carga dejan de ser eficaces. Por una parte, esto se debe a que el procesador central se convierte en un cuello de botella que debe almacenar y gestionar la distribución de carga para un número elevado de procesadores. Por otro lado, el coste asociado a las necesidades de comunicación entre el procesador central y el resto de procesadores puede llegar a ser inaceptablemente alto a medida que los sistemas incorporan más procesadores y se encuentran más distribuidos geográficamente, como es el caso de plataformas de tipo GRID. Esta situación ha motivado el desarrollo de procedimientos distribuidos o descentralizados para realizar la distribución. En la Sección 3.1 de este capítulo se describen las características de los procedimientos distribuidos de equilibrado de carga y se hace referencia a los trabajos que han aparecido sobre esta línea de investigación. La Sección 3.2 describe la clasificación de los procedimientos distribuidos de equilibrado de carga que proponemos. La clasificación descrita permite una parametrización del espacio de diseño de los procedimientos de distribución dinámica de carga. Aprovechando esa descripción del espacio de diseño, se

pueden aplicar técnicas de computación evolutiva para explorar aspectos de dicho espacio y tratar de determinar estrategias, valores de parámetros, tendencias, invariantes, etc., que ayuden a diseñar procedimientos óptimos de distribución dinámica de carga. Precisamente el Capítulo 4 se dedica a los resultados obtenidos y a la descripción del procedimiento desarrollado a partir de un algoritmo genético. Para finalizar el presente capítulo, la Sección 3.3 describe la utilización de la computación evolutiva en problemas de optimización dinámica, como el que consideramos aquí, y la Sección 3.4 proporciona las conclusiones más importantes que se han alcanzado en el capítulo. Antes de pasar a la Sección 3.1 es importante indicar una cuestión de terminología, en este capítulo y en el siguiente, a no ser que se indique lo contrario, cuando hablamos de procedimientos de distribución de carga nos referimos a procedimientos distribuidos de distribución dinámica de carga. Por otra parte, indistintamente hablaremos de distribución de carga o equilibrado de carga, para de esta forma evitar la aparente repetición que se tiene al hablar de procedimiento *distribuido* de *distribución* de carga.

3.1. Equilibrado dinámico y distribuido de carga

Los procedimientos centralizados para la distribución dinámica de carga tienen una desventaja importante debida a que el *planificador central* sólo puede enviar tareas secuencialmente a los procesadores, y una vez se ha terminado el envío inicial de tareas a los procesadores, sólo puede responder una a una a las peticiones de tareas de los procesadores. El planificador central se convierte en un posible cuello de botella, limitando la escalabilidad del procedimiento de distribución de carga y, por tanto, su aplicabilidad a plataformas configuradas a partir de un número elevado de procesadores y computadores distribuidos. Así pues, los procedimientos centralizados funcionan bien si hay pocos procesadores entre los que distribuir las tareas y/o las tareas son intensivas en cuanto a cálculo. Si hay muchos procesadores y tareas con una granularidad muy fina es conveniente distribuir el trabajo del procesador central, es decir, utilizar un procedimiento distribuido de equilibrado de carga.

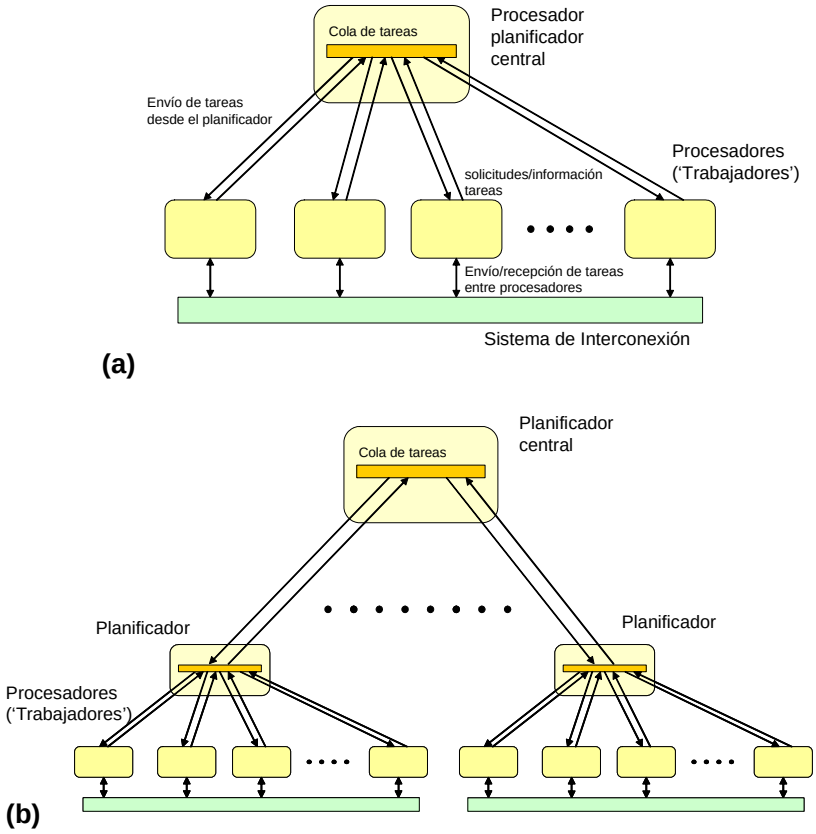


Figura 3.1. Procedimientos centralizado (a) y semi-distribuido (b)

Una primera aproximación al diseño de un procedimiento distribuido de distribución de carga consiste en utilizar varios *planificadores* entre los que se divide el trabajo de la aplicación. Cada uno de esos planificadores se encarga de distribuir la carga que le ha correspondido entre un subconjunto de procesadores de la plataforma que se le asignan. Un procedimiento de distribución de carga de nivel superior se encarga de repartir las tareas entre los distintos subconjuntos de procesadores, esto es, entre los distintos *planificadores*. La Figura 3.1 compara este procedimiento distribuido (Figura 3.1(b)) con un procedimiento centralizado (Figura 3.1(a)). Este tipo de procedimiento se denomina *semi-distribuido* [ZNA94], y puede repetirse en niveles sucesivos, dando lugar a una estructura en árbol en cuyas hojas estarían los procesadores que procesan el trabajo y cuyos nodos internos

corresponderían a los distintos procesadores planificadores encargados de la distribución de tareas.

En un *procedimiento completamente distribuido*, una vez se reparten las tareas entre los procesadores, éstos se encargan tanto del procesamiento de las tareas que se les asignan, como de la redistribución de las tareas para dar respuesta a los cambios que pueden producirse en el estado de carga de los procesadores. Como ya se ha indicado, estos cambios pueden deberse a la propia dinámica de la plataforma, como sería el caso de las modificaciones del estado de carga de los distintos nodos por la presencia de otras aplicaciones, o por fallos en algún procesador u otro elemento del hardware de la plataforma. Además, también pueden deberse a las características de la aplicación, si es difícil prever el coste computacional de las tareas, se generan nuevas tareas, etc. En el Capítulo 1 (Sección 1.3.2) se describieron las etapas de un procedimiento de distribución dinámica de la carga y las políticas que se utilizan en dichas etapas. Igual que en el Capítulo 2 (Sección 2.1) se describieron esas etapas y las correspondientes políticas en el caso de un procedimiento centralizado, a continuación se van a particularizar para un procedimiento distribuido.

En un procedimiento distribuido, todos los procesadores deben implementar, de un modo u otro los procedimientos correspondientes a las fases de (1) evaluación de la carga, (2) iniciación de la distribución, (3) cálculo del volumen de trabajo a transferir, (4) selección de tareas, y (5) migración de tareas. Para llevar a cabo esa implementación distribuida de las fases del procedimiento de distribución de carga, los procesadores necesitan intercambiar información acerca de su estado de carga, debiendo existir un compromiso entre el coste de comunicación asociado al equilibrado de la carga y el efecto positivo previsible de dicho equilibrado. A continuación se consideran, desde el punto de vista su implementación distribuida, cada una de las fases de un procedimiento de distribución de carga y las políticas que intervienen en el mismo. También presentaremos las alternativas más relevantes en cada una de esas políticas de cara a definir una taxonomía de procedimientos distribuidos de equilibrado de carga que nos permitan luego llevar a cabo un análisis de su espacio de diseño mediante algoritmos genéticos. Pasamos, en primer lugar, a reconsiderar las fases del procedimiento:

1. Evaluación de la carga. Cada procesador puede evaluar la carga que tiene asociada sin necesidad de que se lleve a cabo ninguna comunicación entre procesadores. Así, como ya se

indicó en el Capítulo 1, si el volumen de trabajo asociado a la tarea j-ésima es l_j, la carga del procesador i-ésimo, al que se han asignado las tareas contenidas en el conjunto T_i, será

$$L_i = \sum_{j \in T_i} l_j$$

Toda la explicación realizada en la Sección 1.3.2 acerca de la forma de obtener esta medida y de la conveniencia de utilizar métricas sencillas, que puedan evaluarse en tiempos reducidos, son válidas aquí.

2. Inicio de la distribución. En un procedimiento distribuido, cada procesador debe determinar el momento en que debe iniciar las operaciones necesarias para redistribuir la carga. Para ello es preciso detectar si existe una distribución desigual de la misma, y estimar si el coste asociado a la redistribución es mayor o menor que el beneficio que se puede obtener. En la Sección 1.3.2 se utilizaba un nivel de equilibrio de carga ($eq=L_{med}/L_{max}$) definido a partir del cociente de la media de las cargas de los procesadores del computador (L_{med}) y la carga del procesador más cargado (L_{max}). Cuanto mayor sea L_{max} con respecto al promedio de carga eq sería más pequeño, y el procedimiento de distribución de carga debería iniciarse cuando el valor de eq se sitúe por debajo de valor eq_{min} que se especifique. Para evaluar el desequilibrio de carga de esta forma los procesadores deben comunicarse entre ellos para determinar los valores de L_{med} y L_{max}, y debe existir alguna forma de sincronización para que los valores de carga que se utilicen sean coherentes con el estado de carga real que tiene cada procesador en el momento del cálculo. En la Sección 1.3.2 se indicaron algunas posibilidades para reducir esta sobrecarga. Por ejemplo, se pueden utilizar valores de carga de grupos de procesadores vecinos para determinar valores de L_{med} locales para los procesadores de cada grupo, e iniciar la distribución de carga en cada procesador si su carga supera a L_{med} en un determinado umbral. Esta opción, no obstante, no garantiza la distribución de carga equilibrada en todo el computador [WAT98]. Otra posibilidad es utilizar L_{max}/eq_{min}: se iniciaría la distribución de la carga cuando la carga de algún procesador (L_i, para algún i) sea mayor que L_{max}/eq_{min}. Sin embargo, esta medida sólo es suficientemente indicativa si la carga total del programa paralelo se mantiene prácticamente constante. En [LIN87, MUN95, WIL91] se utilizan aproximaciones de este tipo, aunque con dos umbrales. Así, un procesador iniciaría una redistribución de la carga si tiene una carga mayor que el umbral de carga superior o menor

que el umbral inferior. En cualquier caso, hay que tener en cuenta que el coste de comunicación va a depender bastante de los recursos de la plataforma paralela que se utilice. Por ejemplo, muchos computadores paralelos facilitan la estimación de la carga media y máxima al disponer de recursos que permiten realizar las operaciones de suma y máximo global con una complejidad de orden $log_2(P)$, siendo P el número de procesadores. La mejora que supone iniciar la redistribución de la carga se puede estimar utilizando eq_{min}, o guardando información de las distribuciones de carga pasadas. El coste estaría asociado a la comunicación que se requiere para mantener actualizada esa información en cada procesador, debiéndose establecer un compromiso entre la necesidad real de actualizar dicha información y el coste de hacerlo.

3. Cálculo del volumen de trabajo a transferir. La mayoría de los procedimientos que se han propuesto en la literatura intentan aprovechar la localidad de las comunicaciones, estableciendo transferencias de trabajo únicamente entre procesadores vecinos. Se puede considerar que los algoritmos más significativos presentados en la literatura están basados en el *equilibrado jerárquico* (*hierarchical balancing*, *HB*) [HOR93], el *intercambio dimensional* (*dimensional exchange*, *DE*) [XU97], y las *técnicas de difusión* (*diffusive techniques*, *DT*) [WIL93]. Todos ellos se describieron someramente en la Sección 1.3.2. Estos procedimientos deben implementarse de forma distribuida (por ejemplo la resolución de la ecuación de difusión en los métodos de difusión), de manera que cada procesador determine si debe ceder o recibir trabajo. Además, se requiere la comunicación que permita a cada procesador disponer de valores lo suficientemente actualizados de la carga en otros procesadores. Cuanto menor sea la información de otros procesadores (no vecinos) que requiera un procedimiento, mejor.

4. Selección de tareas. Una vez determinado el volumen de trabajo a intercambiar, la selección de tareas que dan cuenta de ese volumen se realizaría en cada procesador. Sólo se necesitaría comunicación en el caso de que se tengan que enviar tareas entre dos procesadores para conseguir la transferencia de trabajo neta que se haya determinado. En este caso debe existir comunicación entre los procesadores que intercambien tareas. Esta es otra razón para que los procesadores que puedan intercambiar sus tareas sean vecinos, tal y como se indicó para el cálculo del volumen de trabajo a transferir.

5. Migración de tareas. El coste de comunicación asociado a la migración de las tareas depende de la topología de la red y de la ubicación de los procesadores entre los que se mueven las tareas. Lo deseable es que las tareas migren entre procesadores vecinos o lo más próximos posible según la topología de la red.

Una vez analizadas las etapas del procedimiento de distribución de carga en relación con su implementación distribuida, pasamos a continuación a considerar las políticas que se implementan en esas etapas en el marco de un procedimiento distribuido. Como se ha indicado en el Capítulo 1, las políticas a definir son la *política de información*, la *política de transferencia*, la *política de localización*, la *política de distribución* y la *política de selección*. De estas políticas, la política de información está relacionada con la actualización de la información del estado del sistema que tienen los procesadores, mientras que las restantes cuatro políticas definen propiamente las características de la distribución de carga. Así la política de transferencia se refiere al *cuándo* se produce la redistribución de la carga, la de localización al *dónde* (los procesadores que transfieren tareas), la de distribución al *cómo*, y la de selección al *qué* (las tareas que migran).

- La *política de información* determina las características del intercambio de información que debe producirse entre los procesadores para que éstos puedan tener una imagen lo más actualizada posible del estado de carga de toda la plataforma. El conjunto de información local y remota que tiene un procesador es la que le permite determinar si debe iniciar una redistribución de carga, el volumen de trabajo que debe transferir, si debe actuar como emisor o receptor, etc. Las dimensiones más importantes para caracterizar una política de información están relacionadas con el marco espacial y con el marco temporal. Así, una política de información debe fijar *la topología* que definen los procesadores con información de estado mutua. Así, un procesador puede tener información de todos los procesadores de la plataforma, de aquellos con los que tiene relaciones de vecindad según la red de interconexión, o de grupos de procesadores que puedan establecerse con arreglo a otros criterios (por ejemplo, un grupo de procesadores seleccionados aleatoriamente), que incluso pueden cambiar temporalmente. Además, la política de información debe establecer los instantes en los que los procesadores deben intercambiar información. Así, la política de información puede ser *periódica* o *a demanda*. En la política de información

periódica, los procesadores intercambian información de su carga con una frecuencia que denominaremos frecuencia de equilibrado de carga (LBF, de *load balancing frequency*). En el caso de la política de información a demanda la información de estado se intercambia cada vez que se tiene que realizar una redistribución de la carga, ya que se supone que es el momento en que se van a producir cambios en las cargas de los procesadores que no se pueden determinar localmente.

- La *política de transferencia* establece las condiciones bajo las que debe producirse la migración de las tareas entre los procesadores. Para esto, cada procesador utilizaría la información de carga (local y remota) a la que nos hemos referido al hablar de la etapa de inicio de la distribución de carga (usualmente utiliza sólo información de carga local) y establece si debe transferir tareas o solicitar que se le envíen. Así, según quién inicie las operaciones de equilibrado de carga la política de transferencia puede ser *iniciada por el emisor*, *iniciada por el receptor*, o *simétrica*. En el caso de la transferencia iniciada por el emisor, es el procesador que está sobrecargado y tiene que enviar tareas a otros procesadores el que comienza las operaciones de distribución de carga. Si la transferencia es iniciada por el receptor, el procesador que necesita trabajo es el que inicia las operaciones de distribución de carga. Finalmente, en la política de transferencia simétrica, el procesador que está sobrecargado y va a enviar trabajo a otro procesador y el que solicita trabajo deben sincronizarse para realizar la transferencia de uno a otro. En [HAC87] se evalúan las prestaciones de un procedimiento con transferencia iniciada por el receptor para distintas situaciones de carga, y en [EAG86] se muestra que una política de transferencia iniciada por el emisor se comporta mejor sistemas con carga de baja a moderada, mientras que en el caso de cargas elevadas es la política de transferencia iniciada por el receptor la que proporciona las mejores prestaciones. Según esto, un procedimiento con transferencia simétrica eficiente debería comportarse como un procedimiento con transferencia iniciada por el emisor en el caso de niveles de carga bajos o medios, y como un procedimiento con transferencia iniciada por el receptor cuando los niveles de carga sean elevados [ANT04].
- La *política de localización* determina los procesadores que intervienen en las transferencias de tareas que tienen lugar en el momento de redistribuir la carga. Esta selección de procesadores se realiza también de acuerdo con la información de que dispone cada procesador acerca de su carga y de la carga de los otros procesadores de la

plataforma. Esta política está relacionada con la de transferencia, ya que la decisión acerca de si un procesador tiene un nivel de carga que le hace emisor o receptor corresponde, precisamente, a la política de localización. Se pueden distinguir cuatro tipos de políticas de localización: las políticas de *un umbral*, las de *dos umbrales*, las de *nivel mínimo/máximo de carga*, y las *aleatorias*. En las políticas de un umbral un nodo es emisor o receptor según su carga esté por encima o por debajo, respectivamente de un determinado nivel que se toma como umbral. En las políticas de localización de dos umbrales existen dos umbrales, uno de ellos (el umbral superior) establece el valor por encima del cual el procesador se considera sobrecargado, y el otro (el umbral inferior) representa el valor por debajo del cual es procesador tiene un nivel de carga bajo. Un procesador será emisor si su nivel de carga está por encima del umbral superior, y receptor si está por debajo del umbral inferior. A la hora de establecer los umbrales hay que tener en cuenta que la carga de la aplicación puede variar con el tiempo, por lo tanto, es conveniente utilizar umbrales que cambien dinámicamente, adaptándose al nivel de carga del sistema. Ya en el Capítulo 2, al describir el procedimiento centralizado basado en algoritmos genéticos propuesto en [ZOM01], nos referimos a esta cuestión de los umbrales adaptativos. Como vemos, en las políticas de uno o dos umbrales, la selección del papel de un procesador se hace en base a la información local de carga y no implican intercambio de mensajes entre procesadores. En las políticas de nivel mínimo/máximo de carga, el emisor selecciona como receptor al procesador del sistema con el menor nivel de carga, y el receptor selecciona como emisor al procesador con el mayor nivel de carga. En este caso, sí que es necesario disponer de información de carga de los procesadores remotos para determinar el procesador más o menos cargado. El último tipo de política de localización es la aleatoria, en la que el emisor (receptor) elige al receptor (emisor) de forma aleatoria. En este caso tampoco se necesita información de la carga de los procesadores remotos aunque, es previsible que esta política no ofrezca buenas prestaciones. En [SHI90] se evalúan distintas políticas de localización.

- La *política de selección* es la que determina qué tareas pueden elegirse para ser transferidas desde un procesador que ha sido designado emisor a partir de la política de localización (y en algunos casos también a través de la política de distribución, como veremos). El volumen de trabajo que debe transferirse a través de las tareas que se seleccionan se determina a través de la política de distribución. Existen dos alternativas

para esta política. Por un lado está la política de selección *con corte forzado* (preemptive), en la que, entre las tareas que se pueden elegir se incluye la tarea que se está procesando. En cambio, en la selección de tareas sin corte forzado (no preemptive) sólo se pueden seleccionar las tareas que están pendientes de ser procesadas. Las políticas de selección sólo necesitan información que permita estimar la carga de trabajo de cada una de ellas.

- La *política de distribución* es la que determina la forma de equilibrar el trabajo entre los procesadores que la política de localización ha seleccionado como emisores o receptores. Esta política está relacionada con las etapas de evaluación la carga de trabajo, selección de tareas, y de migración. Por lo tanto, las opciones que pueden contemplarse para esta política incluyen las alternativas, descritas someramente en la Sección 1.3.2, que se han propuesto para realizar esas etapas. A continuación, en la Sección 3.1.1, se describen las opciones más relevantes para la distribución de tareas en el contexto de su implementación distribuida. En la política de distribución las tareas que pueden elegirse para constituir el trabajo a intercambiar son las que permite la política de selección de tareas. Como se ha dicho, en la selección de los procesadores emisores interviene, a veces, ésta política de distribución. Esto ocurre cuando para conseguir una determinada transferencia neta de carga tanto el procesador que debe emitir carga como el que debe recibir tienen que transferir tareas (ambos sería emisores) aunque sólo el seleccionado como emisor reduce su carga, y el seleccionado como receptor la aumenta [WAT98].

Según lo que se ha descrito, las características deseables de un procedimiento distribuido de equilibrado de carga son las siguientes: disponer de un conocimiento preciso de las condiciones de carga, requerir un intercambio de información limitado, tener un comportamiento simétrico en cuanto a la transferencia de carga, y ser escalable. En la literatura se pueden encontrar bastantes propuestas de procedimientos distribuidos, a continuación se describen las alternativas más importantes en las que se pueden agrupar y se indican las referencias a los procedimientos más representativos dentro de cada línea.

3.2 Procedimientos distribuidos propuestos en la bibliografía

Los procedimientos distribuidos se pueden agrupar de formas muy diversas. En esta Sección describiremos en primer lugar los procedimientos de difusión, intercambio dimensional, y los arbitrarios, en el que el elemento de diferenciación es la estrategia de distribución que, como se ha descrito en la sección anterior, está relacionada con la selección del trabajo a transferir y la forma de distribuir el mismo entre los procesadores. Usualmente, estas estrategias de distribución de carga se consideran como algo distinto de la etapa de medición de carga y de los mecanismos de asignación de las tareas a los procesadores. Las estrategias que se han propuesto en los distintos procedimientos se evalúan tanto mediante modelos matemáticos, como por simulaciones, o simplemente por aproximaciones intuitivas. De esta forma, resulta complicado comparar las diferentes estrategias para encontrar la mejor, dado un algoritmo paralelo [NIS93]. También se hará referencia a los procedimientos adaptativos que cambian sus parámetros con arreglo a las condiciones de carga de los procesadores, y los híbridos que combinan características de procedimientos centralizados y distribuidos para evitar los problemas de cada uno de ellos.

3.2.1 Procedimientos de difusión

En la difusión natural [COR99, CYB89, ELS00, GHO96b, WIL93] la carga del trabajo se va transfiriendo periódicamente desde los procesadores sobrecargados a los procesadores con poca carga. Cada procesador utiliza información sólo de procesadores vecinos y las tareas sólo emigran entre vecinos inmediatos. Así pues, tanto el dominio de la información como el espacio de migración son locales. Cada procesador realiza sus operaciones de equilibrado de carga independientemente. Entre las variaciones que se han propuesto para la difusión se encuentran la difusión iniciada por el emisor (SID) y la difusión iniciada por el receptor (RID). En la SID la difusión se inicia en un procesador muy cargado, mientras que en la RID es un procesador con poca carga la que la inicia. Según se ha puesto de manifiesto en algún trabajo [WIL93] las prestaciones de los procedimientos RID son mejores que las de los SID.

Como ejemplo, describiremos el algoritmo RID que se describe en [WIL93], donde también se describe un algoritmo SID basado en los mismos principios de funcionamiento. En el caso del algoritmo RID, un procesador i inicia el equilibrado de carga si su carga (la

cantidad de trabajo que tiene asignada) está por debajo de un umbral L_{bajo}. A continuación se calcula la carga media del procesador i y la de sus vecinos, $\overline{L_i}$. Si la carga del procesador *i*, L_i, es menor que la carga media en una diferencia igual a otro umbral prefijado, L_{umbral} (es decir $L_i+L_{umbral}\leq\overline{L_i}$) se pasa al siguiente paso del equilibrado de carga. En este paso, a cada vecino, *j*, se le asigna un peso de exceso de carga, w_j, según la formula:

$$w_j = \begin{cases} L_j - \overline{L_i} \text{ si } L_j > \overline{L_i}, \\ 0 \text{ si no.} \end{cases}$$

El peso total de exceso de carga de todos los vecinos W_i es igual a:

$$W_i = \sum_{j=1}^{J} w_j$$

y la cantidad de la carga solicitada por el procesador i al vecino j, δ_j, viene dada por la expresión:

$$\delta_j = (\overline{L_i} - L_i)\frac{w_j}{W_i}$$

Las solicitudes de carga se envían a los vecinos correspondientes, y cada vecino al que se le haya solicitado carga transferirá la mitad su carga actual, como máximo [WIL93]. Con esto se evita la inestabilidad en la evolución de la carga de los procesadores, y se reduce el efecto de la falta de actualidad en la información al ir cambiando las condiciones de carga de los procesadores cuando transfieren tareas de unos a otros.

Todos los procesadores informan a sus vecinos de sus niveles de carga y actualizan esta información dinámicamente. La calidad de la información de carga que se maneja depende, obviamente, de la frecuencia con que se actualice. Pero una frecuencia muy alta de actualización puede implicar un coste de comunicación inaceptable. Por lo tanto hay que encontrar un equilibrio entre la actualidad de la información y la frecuencia con que se renueva. Hay que tener en cuenta que, en la estrategia de difusión, el estado de equilibrio de carga a que se llegue puede verse afectado por el hecho utilizar información de carga inexacta por no haberse actualizado con la suficiente frecuencia. Se dice que la información de carga sufre un envejecimiento debido a que los cambios de carga en los procesadores se producen

de forma asíncrona. Esto puede conducir a que un procesador dé lugar a transferencias innecesarias de carga y a que no se llegue a un estado final estable para la carga. En [CHE94] se describe cómo evitar ese problema mediante una estrategia RID adaptable que ajusta automáticamente el factor de difusión de la carga según las fluctuaciones de carga en el sistema.

La difusión tiene algunas ventajas. Por una parte, se trata de un procedimiento completamente distribuido que, por tanto, evita cuellos de botella. Además sólo necesita que se produzca comunicación entre vecinos. No obstante, precisamente por esa razón, el movimiento de la carga es relativamente lento: un procesador poco cargado no podría obtener carga de forma inmediata si todos sus vecinos están poco cargados, aún cuando haya procesadores sobrecargados en otra parte en el sistema [CYB89].

3.2.2 Procedimientos de intercambio dimensional

Como se indicó en el Capítulo 1, este tipo de procedimientos se propusieron inicialmente como algoritmos de equilibrio de carga para multicomputadores con red hipercubo [CYB89]. En estas plataformas, una operación de distribución de carga consta de *log(P)* pasos de equilibrado, entre parejas de procesadores, donde *P* es el número de procesadores. Cada paso de equilibrado se corresponde con cada una de las *log(P)* dimensiones del hipercubo. Todos los pares de procesadores de la misma dimensión intercambian su información de carga y determina el promedio de tareas que tienen asignadas. El sistema completo llega al equilibrio de carga tras una única iteración.

En [HOS90] se presenta una generalización del método para cualquier topología de interconexión. Este método se basa en el coloreado de arcos de grafos no-dirigidos. Los arcos del grafo se colorean utilizando el mínimo número de colores de forma que no haya dos arcos contiguos (conectados al mismo nodo) que tengan el mismo color. En estas condiciones, cada dimensión queda definida por todos los arcos con el mismo color. En una topología que no es de tipo hipercubo, el método de intercambio dimensional puede no alcanzar una distribución uniforme [HOS90]. Para una topología *k-aria n-cubo*, el número de iteraciones que se necesita depende linealmente del número total de procesadores en una dimensión. Para reducir el número de iteraciones, en [WU96] se describe un método directo que consigue el

equilibrado de la carga en una iteración, permitiendo que cada nodo determine el estado de sistema global mediante la operación de reducción de la suma. En [XU95] se ha parametrizado el método de intercambio dimensional según la cantidad de la carga que migra entre pares de procesador. Así, se define el parámetro de intercambio de la carga del trabajo, λ, cuyo valor óptimo, λ_{opt}, es el que da lugar a la convergencia más rápida en el equilibrado. Una distribución equitativa de la carga de trabajo entre parejas de procesadores, es decir $\lambda_{opt} = 1/2$, sólo produce resultados óptimos para la estructura hipercubo. Para otras topologías y tamaños de red, el valor de λ_{opt} se varía. Por ejemplo, para una malla λ_{opt} es $1/(1+\sin(\pi/k))$, donde k es la longitud máxima en una dimensión de la topología.

for color := 1 to max_color **do**
    Nodos i y j, intercambian información de carga l_i y l_j
    if $(l_i - l_j) > 1$ **then**
        enviar $\lfloor(\lambda l_i - \lambda l_j)\rfloor$ tareas al nodo j
    endif
    if $(l_j - l_i) > 1$ **then**
        enviar $\lfloor(\lambda l_j - \lambda l_i)\rfloor$ tareas al nodo i
    endif
endfor

Figura 3.2. Intercambio dimensional para topología arbitraria

La versión del algoritmo del método de intercambio dimensional para una topología arbitraria de la Figura 3.2 utiliza cargas de trabajo expresadas mediante un números enteros. En esta descripción, el equilibrio absoluto de la carga no se puede conseguir: se considera que un par de procesadores tiene una carga equilibrada cuando la diferencia entre sus niveles de carga no sea mayor que uno. Aunque los resultados teóricos muestran que el método de intercambio dimensional supera a la estrategia de la difusión natural, en la práctica, el método de intercambio dimensional no escala bien en sistemas paralelos muy grandes [WIL93]. Esto se debe a que los estudios teóricos no consideran el coste de la sincronización global en que se incurre siempre que se lleva a cabo el equilibrado de carga.

3.2.3 Algoritmos arbitrarios.

Los algoritmos arbitrarios son bastante simples, eficaces, e independientes de la topología. Existen dos variantes significativas de los algoritmos arbitrarios. En la primera de ellas, una tarea que se acaba de crear se envía a un procesador elegido aleatoriamente [CHA94, SHU96]. En la segunda alternativa, una tarea migra sólo cuando la carga de trabajo supera un cierto límite [EAG86]. En [SHU96] se muestra que los algoritmos arbitrarios presentan un rendimiento adecuado. Además, la política de sondeo que suelen utilizar es bastante útil para distribuir de la carga [SAN94, LEE97, SHE02]: cuando un procesador se queda sin trabajo envía una petición a un procesador seleccionado aleatoriamente, repitiéndose la operación hasta que el procesador reciba trabajo o no quede más trabajo en el sistema (ha terminado de ejecutarse el programa). Se trata de un procedimiento totalmente distribuido, en el que cada procesador tiene la misma probabilidad de ser seleccionado, y no presenta cuellos de botella ocasionados por el control centralizado.

Un inconveniente consiste en que el coste de la comunicación puede ser bastante grande debido al número imprevisible de peticiones arbitrarias que pueden generarse. Además, en el peor caso, no existen garantías de que cualquier procesador que solicite trabajo lo reciba alguna vez y se pueda alcanzar el equilibrado de la carga. Por otra parte, la falta de localidad puede causar una cierta degradación en el rendimiento del procedimiento debido al tráfico de comunicación, dado que la probabilidad de tener que transferir una tarea a procesadores remotos es alta. Este es característico, sobre todo, en los algoritmos arbitrarios que migran tareas inmediatamente después de su creación.

3.2.4 Procedimientos adaptativos

En [XU93, XU90] se propone un procedimiento adaptativo de equilibrado de carga en el que adaptan dos parámetros importantes para el equilibrado de la carga: la frecuencia de actualización de información de carga y el valor del umbral utilizado para indicar si un procesador está muy cargado. El sistema usa un procesador dedicado que recibe y transmite periódicamente la información de carga. La duración entre actualizaciones sucesivas, que se denomina ventana de tiempo, varía según el estado de carga de sistema. Todas las demás políticas son completamente distribuidas. La política de localización determina que el

procesador con una carga mínima, o que ha recibido una menor cantidad de tareas, es el candidato para receptor. Además el valor del umbral adaptable y el dominio de migración se pueden determinar a partir de la política de tipo local o global que se describe a continuación.

La política local permite que exista migración sólo entre vecinos inmediatos, con un valor de umbral igual a la carga media entre vecinos (incluyendo al propio procesador). La política global considera la carga de todo el sistema. Aunque el procesador que recibe toda la información de la carga de los procesadores actualiza periódicamente dicha información, las decisiones que se toman para realizar el equilibrado de la carga puede que no estén basadas en una información totalmente correcta de la carga ya que cualquier cambio que ocurra en el intervalo de tiempo de una ventana sólo es conocido por los procesadores implicados en la migración. Por esta razón, la decisión respecto a la condición de equilibrado de carga puede no ser óptima. En [MEL96a, MEL96b] se mejora el método añadiendo un retardo local además del retardo global asociado a la ventana de tiempo. A través de estos retardos se determina cuándo un procesador debería enviar su información de carga y cuándo el procesador que recibe toda la información debe transmitir el estado de todo el sistema. De esta forma se pueden detectar cambios en la carga local que conduzcan a fluctuaciones en la carga del sistema.

3.2.5 Aproximaciones híbridas

La mayor parte de procedimientos de distribución de carga que se han descrito corresponden a una aproximación puramente centralizada o puramente distribuida. Un ejemplo claro de la aproximación centralizada es el que se describen en [NI85], basado en un modelo maestro-trabajador. En un procesador se ejecuta el proceso maestro, que distribuye la carga de trabajo entre los demás procesadores (también se podría incluir el propio procesador), que realizan la computación de las tareas del programa (procesos trabajadores). Cuando un proceso trabajador se queda sin tareas solicita más al maestro. Este procedimiento tiene una escalabilidad limitada ya que el procesador maestro constituye un cuello de botella. Lo mismo ocurre con cualquier aproximación centralizada y por ello se buscan aproximaciones distribuidas eficientes. No obstante, como se ha ilustrado más arriba, los procedimientos distribuidos también presentan problemas. Por ejemplo, en el caso de una técnica tan utilizada como la difusión, se pueden tomar decisiones subóptimas de migración de carga. Por ejemplo,

un procesador sobrecargado no puede ceder trabajo a un procesador menos cargado del que no sea vecino, por lo que un procesador puede permanecer sin trabajo durante un tiempo sustancial, aun cuando haya tareas en el sistema. Este efecto es particularmente significativo en los problemas de computación estructurada en árbol debido a que en ellos el cálculo comienza a partir de una única tarea. Así, el equilibrio global puede alcanzarse más lentamente que en una aproximación centralizada.

Por tanto, se han propuesto procedimientos híbridos de distribución de carga que combinan características de los procedimientos centralizados y de los distribuidos para superar la dificultados que aparecen en cada caso de manera que se consiga la escalabilidad de procedimientos distribuidos con la calidad de información de carga que proporcionan los modelos centralizados. En esta aproximación se pueden incluir los procedimientos semi-distribuidos de la Figura 3.1(b) y los procedimientos adaptativos que hemos descrito antes [XU93, XU90]. En [AHM91] se propone una aproximación semi-distribuida que divide la red en regiones independientes centradas en algunos procesadores de control. Los procesadores centrales son planificadores que planifican tareas de forma óptima dentro de sus dominios locales, y permiten la migración de tareas a otros dominios si la carga del procesador menos cargado en su dominio es mayor que el umbral establecido. Las tareas se transfieren si existe una diferencia significativa entre la carga acumulada en el dominio local y la del dominio remoto.

En [WU91] se propone un algoritmo en el que cada procesador ejecuta localmente subconjuntos de tareas primero *en profundidad* y se mantiene un fondo global de tareas del que un procesador ocioso puede obtener una tarea. En [WU91] se muestra que, bajo ciertas suposiciones, el algoritmo propuesto es óptimo con respecto al número de tareas que tienen que ser transferidas, pero el mantenimiento del fondo global es costoso y difícil de implementar sobre una máquina de memoria distribuida. No obstante, la simulación en una plataforma de memoria compartida escalable de un algoritmo similar [NAS96] muestra una buena escalabilidad.

3.3. Optimización de las estrategias de distribución de carga

A través de la descripción realizada de los distintos procedimientos de equilibrado de carga dinámica se ha intentado poner de manifiesto la complejidad que encierra el diseño de un procedimiento de distribución dinámica de carga que proporcione las mejores prestaciones posibles para una aplicación paralela determinada. Por una parte, el conjunto de políticas que hay que tener en cuenta y la interdependencia entre las mismas es una de las razones de esa complejidad, pero no menos importante es la falta de modelos precisos y manejables que ayuden a estimar el nivel de prestaciones que puede esperarse al utilizar un procedimiento de distribución determinada o establecer comparaciones fiables entre diferentes alternativas de implementación. Aparte de la propia complejidad del sistema paralelo a modelar, una parte importante de la responsabilidad de esta situación se encuentra en la propia naturaleza dinámica del problema, que exige que todo modelo eficiente deba tener en cuenta la diversidad de condiciones en las que puede trabajar el sistema.

Nos enfrentamos a un problema de optimización de un proceso dinámico, que en este caso, es el procedimiento distribuido de equilibrado dinámico de la carga en una aplicación paralela. En esta Tesis se ha abordado el problema mediante la computación evolutiva, que constituye una herramienta eficiente para la exploración de amplios espacios de búsqueda. Así, es posible alcanzar algunas conclusiones sobre las características de estos espacios a fin de ayudar en la determinación de las mejores alternativas de solucionar el problema en consideración. Para implementar esta aproximación se ha diseñado una descripción adecuada del espacio de soluciones (el espacio de diseño de los procedimientos distribuidos de equilibrado de carga dinámica), que hace posible su exploración automática mediante un algoritmo genético. La evaluación de idoneidad de cada solución (un programa paralelo que implementa uno de los procedimientos de distribución de carga posibles) se realiza ejecutando el programa en la plataforma paralela correspondiente.

En esta sección proponemos una clasificación de los procedimientos distribuidos de equilibrado dinámico de carga que nos permitirá parametrizar dichos procedimientos. De esta forma, es posible transformar un procedimiento genérico de equilibrado dinámico distribuido que hemos desarrollado en el procedimiento que se desee, a partir de los parámetros que caracterizan a dicho procedimiento en la clasificación propuesta. La aproximación que

seguimos es similar en muchos aspectos a la que plantea la programación genética [KOZ92]. Esta rama de los algoritmos genéticos cuyo objetivo es generar programas óptimos. Para ello utiliza poblaciones cuyos individuos son programas que se evalúan con arreglo a una función de idoneidad que representa lo bien que cada programa resuelve el problema en cuestión. A la población de programas se aplican las operaciones de mutación y cruce, para generar nuevos programas factibles, y las de selección para mantener una población con las soluciones más idóneas (los mejores programas). En nuestro caso, los programas a optimizar son los procedimientos distribuidos de equilibrado dinámico de carga. Mediante la clasificación que describimos a continuación y la parametrización asociada a la misma se facilita el diseño de operadores genéticos eficientes a la hora de generar soluciones factibles. Si bien la metodología que describimos aquí se centra en un tipo determinado de procedimientos de distribución dinámica de carga, consideramos que puede extenderse a otros procedimientos, y que las conclusiones generales respecto al método que se pueden extraer en este trabajo pueden extenderse a esos dominios.

En la siguiente sección (Sección 3.3.1) se presenta la clasificación que proponemos para los procedimientos distribuidos de equilibrado dinámico de carga. Después, la Sección 3.3.2 describe el procedimiento genérico de distribución de carga que se puede particularizar al introducir los parámetros que caracterizan un procedimiento concreto según la clasificación de la Sección 3.3.1.

3.3.1 Clasificación de los procedimientos distribuidos

Como se ha indicado para usar los algoritmos genéticos en la exploración del espacio de diseño de los procedimientos distribuidos de equilibrio dinámico de carga se parte de una taxonomía definida a partir de las alternativas que se han propuesto para cada una de las políticas que deben implementarse en un procedimiento. Como en toda taxonomía, puede haber detalles de los procedimientos de distribución de carga no reflejados, y también pueden describirse opciones que no tienen sentido en el caso de una plataforma o una aplicación determinada. En cualquier caso, consideramos que la aproximación que planteamos aquí sigue siendo válida, y lo *único* que habría que hacer es incorporar nuevas dimensiones en la

taxonomía (i.e. nuevos parámetros a tener en cuenta en la codificación de cada procedimiento de distribución de carga), o eliminar las opciones que no tengan sentido.

Así, la taxonomía propuesta parte de las políticas incluidas en cualquier estrategia de distribución dinámica de la carga [DOS96, BEN91]: *política de información*, *política de transferencia*, *política de localización*, *política de distribución*, y *política de selección*. Estas políticas se han descrito detalladamente en anteriores secciones. A continuación indicaremos las alternativas que se tienen en cuenta en cada una de ellas y la interrelación que existe entre alternativas de distintas políticas.

Política de información. Como ya se ha indicado, mediante esta política se fija la forma según la cual cada procesador actualiza la información relativa al estado de carga (y/o cualquier otra medida que se utilice para la distribución de carga. Para la política de información se consideran dos alternativas, la *topología* de intercambio de información y la *frecuencia* con la que la información se actualiza.

La topología se refiere a las relaciones espaciales que cada procesador tiene con los procesadores de los que puede tener información de estado. Las opciones que se han considerado aquí son las siguientes:

(a) *Todos*, en el caso de que un procesador pueda recibir información de todos los procesadores

(b) *Vecinos*, si cada procesador sólo tiene información de estado de los procesadores que están conectados directamente con él a través de la red de interconexión del computador paralelo.

(c) *Grupos*, cuando un procesador recibe información de un grupo de procesadores de la plataforma. Aquí se abre un abanico de opciones definido por el criterio que se utiliza para configurar el grupo. Así, se pueden configurar grupos seleccionando a los procesadores de forma *aleatoria*, o incluyendo aquellos procesadores que no estén a más distancia (en número de enlaces de red) de un procesador dado, etc. Las opciones para esta alternativa que se tengan en cuenta van a depender del tipo de plataformas en las que se esté interesado, ya que la topología de interconexión de los procesadores determina las posibilidades más relevantes. Por ejemplo, en una plataforma en la que

todos los procesadores estén conectados con todos ni siquiera tendrían sentido las tres opciones que se han indicado para la topología de la política de información.

La otra dimensión a considerar en la política de información es la frecuencia con la que se actualiza. Aquí se distinguen dos alternativas:

(a) *Periódica,* si la información se actualiza periódicamente. En este caso hay que fijar un parámetro, que denominamos LBF (*load balancing frequency*) y que indica cada cuánto tiempo se produce esa actualización.

(b) *A demanda,* cuando la información se actualiza en el momento en que se produce algún tipo de cambio en el estado de la misma, por ejemplo, cada vez que se produce un proceso de equilibrado de la carga. Cuando se utiliza esta opción, el criterio que se usa para establecer el momento de actualización puede depender de las características de otras políticas, como por ejemplo la *política de distribución*.

Política de Transferencia. Es la que determina el momento y el lugar en el que se debe iniciar la redistribución de la carga entre los procesadores. Las alternativas que se consideran en esta política son las siguientes:

(a) *Iniciada por el emisor,* en el caso de que sea un procesador que determina que está sobrecargado y debe enviar trabajo a otro, u otros procesadores, el que inicie las operaciones necesarias para equilibrar la carga.

(b) *Iniciada por el receptor,* si es un procesador que determina que tiene un nivel de carga bajo (por ejemplo, está ocioso al no tener más tareas que procesar) y debe solicitar trabajo a otros procesadores el que inicia el procedimiento de equilibrado de la carga.

(c) *Simétrica*, en el caso de que las operaciones de equilibrado de carga puedan ser iniciadas tanto un procesador sobrecargado como un procesador con poca carga. En este caso debe existir una sincronización entre el emisor y el receptor.

Como ya hemos indicado, esta política está estrechamente relacionada con la política de localización que veremos a continuación. De hecho, según sea la alternativa considerada para la política de transferencia, así serán posibles o no determinadas opciones para la política de localización.

Política de Localización. Es la que establece el criterio según el cual un procesador debe incorporarse a una operación de equilibrado de carga como emisor o receptor. Así, cuando se utiliza una política de transferencia iniciada por el emisor, si la política de localización establece que un procesador debe enviar tareas a otro procesador, éste procesador iniciará las operaciones de equilibrado de carga. Si la política de localización establece que establece que debe recibir trabajo entonces se incorporaría al conjunto de procesadores que actuarán como receptores pero no daría lugar al inicio del equilibrado de la carga. Las alternativas consideradas para la política de localización son las siguientes:

(a) *Política de un umbral*, que selecciona un procesador como emisor o como receptor según su índice de carga esté por encima o por debajo, respectivamente, de un valor que se toma como umbral. Esta alternativa implica la necesidad de fijar un valor del umbral. Si este valor se mantiene fijo durante todo la ejecución de la aplicación se habla de *umbral fijo*, y si puede cambiar con arreglo a alguna magnitud, como puede ser el índice de carga medio de los procesadores del sistema, se denomina *umbral adaptativo*.

(b) *Política de dos umbrales*, que utiliza dos umbrales para determinar si un procesador interviene como emisor o receptor en el equilibrado de la carga. Así, existe un umbral inferior que marca el nivel del índice de carga del procesador por debajo del cual se considera que tiene poca carga y, por lo tanto debe participar como receptor, y un umbral superior por encima del cual el procesador se considera sobrecargado y debe participar como emisor. En esta política también se requiere fijar dos parámetros (los umbrales), e igual que en la política de un umbral, estos dos umbrales pueden mantenerse fijos durante toda la ejecución o adaptarse a las condiciones de ejecución.

(c) *Política de mínimo/máximo de carga*, en la que un procesador actúa como receptor si es el que tiene el valor mínimo para el índice de carga y como emisor si tiene el valor máximo del índice de carga.

(d) *Política aleatoria*, en la que se elige aleatoriamente el procesador que actúa como emisor o como receptor.

Una vez se determina el procesador o procesadores que inician la operación de equilibrado de carga y el resto de procesadores que intervienen en el mismo, queda por

decidir el volumen de trabajo que deben intercambiarse los procesadores, las tareas que corresponden a ese volumen de trabajo y los detalles de la migración de tareas. Las políticas de selección y de distribución son las que definen estas cuestiones.

Política de Selección. Esta política establece qué tareas son las que pueden elegirse para ser transferidas. Se consideran dos opciones para esta política:

(a) *Selección con corte forzado* (*pre-emptive*), en la que puede elegirse entre todas las tareas asignadas al procesador, incluyendo la que se está procesando en el momento de iniciar el equilibrado de carga.

(b) *Selección sin corte forzado* (*non pre-emptive*), en la que sólo las tareas que están asignadas al procesador y que no se están ejecutando pueden seleccionarse para ser transferidas.

Política de distribución. Una vez se ha determinado qué tareas pueden ser seleccionadas para migrar de un procesador a otro, queda todavía por determinar el conjunto de tareas que hay que seleccionar para reunir el volumen de trabajo a transferir desde un procesador y a qué procesadores va a ir ese trabajo. Esto es lo que define la política de distribución. En la descripción de etapas de un procedimiento de distribución de carga que se hace en [WAT98] y que se ha descrito en el Capítulo 1, la política de distribución es la que determina las características de las etapas de *cálculo del volumen de trabajo a transferir*, *selección de tareas* (aquí también interviene la política de selección), y *migración de tareas*. Por lo tanto, en esta política se considerarán como alternativas cada uno de los procedimientos que se han propuesto en la bibliografía para calcular el trabajo a transferir, habida cuenta que en estos procedimientos se establece también los procesadores entre los que se pueden intercambiar tareas, y que la determinación de las tareas que constituyen un volumen de trabajo no puede considerarse en sí una característica del procedimiento de distribución de carga, simplemente se realiza el cálculo mediante alguno de los procedimientos eficientes que existen [WAT98, PAP94]. Por tanto, dentro de esta política se considerarán, por ejemplo:

(a) Los procedimientos de difusión natural, descritos en la Sección 3.2.1.

(b) Los métodos de intercambio dimensional, descritos en la Sección 3.2.2.

Aunque pueden incluirse más alternativas que puedan diseñarse o aparezca en la literatura. Cada una de estas alternativas debe establecer:

- *El cálculo del trabajo a transferir*, es decir, la forma de determinar el volumen de trabajo que deben intercambiarse los procesadores y que se utilizará para determinar el conjunto de tareas que deben migrar.
- *La topología de migración*, es decir los procesadores entre los que pueden transferirse tareas. Por ejemplo, se puede prefijar que sólo se transfieren tareas entre procesadores *vecinos*, o entre *cualquier procesador* del sistema, o sólo entre los procesadores que formen parte de un grupo. En éste último caso tendremos la misma casuística que en el caso de la opción análoga en la política de información: se pueden configurar grupos con procesadores elegidos aleatoriamente, o incluyendo aquellos procesadores que no estén a más distancia de un procesador determinado, etc.

En la literatura se suelen definir tres tipos de procedimientos distribuidos de equilibrado dinámico de carga en base a el tipo de política de transferencia que utilizan. Así están los procedimientos iniciados por el emisor, los iniciados por el receptor, y los simétricos. En la Figura 3.2 se ilustran las alternativas que se suelen utilizar en esos tres tipos de procedimientos para las políticas de transferencia, localización, y selección [ZAK97, ZOM99, ZOM01]. Como se puede ver, se pueden definir 16 procedimientos diferentes al fijar cada alternativa.

Así, los *procedimientos iniciados por el emisor* utilizan intercambio de información a demanda y, en cuanto a política de localización implementan la política de un umbral, la de mínimo/máximo de carga, o la aleatoria. La selección de tareas puede ser tanto con corte forzado como sin corte forzado. Los *procedimientos iniciados por el emisor* también utilizan intercambio de información a demanda aunque sólo implementan la política de localización basada en un umbral. Para la selección de tareas también implementan las dos opciones posibles. Finalmente, los *procedimientos simétricos* utilizan intercambio periódico de información y una de las cuatro políticas de localización alternativas que se

han descrito. Para la selección también pueden implementar cualquiera de las dos opciones posibles.

El resto de opciones y dimensiones que no se han indicado no se especifican explícitamente en los procedimientos. Por ejemplo, la política de distribución que se utiliza puede ser de difusión o de intercambio dimensional, y las características topológicas de la política de localización quedarían determinadas por la propia política de distribución.

Una consecuencia importante de la taxonomía definida es que pone de manifiesto la posibilidad de definir nuevas nuevos procedimientos al combinar las distintas opciones de cada una de las políticas, siempre que eso sea posible. Esta cuestión quedará más clara una vez que se describa el procedimiento general de distribución de carga que se particulariza al especificar la opción que se selecciona para cada política.

Política de Transferencia	Iniciada por el emisor			Iniciada por el emisor	Simétrica			
Política de Localización	un umbral	min/max carga	aleatoria	un umbral	un umbral	dos umbrales	min/max carga	aleatoria
Política de Selección	con/sin corte forzado			con/sin corte forzado	con/sin corte forzado			
	Procedimiento de distribución iniciado por el emisor			Procedimiento de distribución iniciado por el receptor	Procedimiento de distribución simétrico			

Figura 3.3. Alternativas en las políticas de transferencia para los procedimientos de distribución de carga iniciados por el emisor, por el receptor, y simétricos

Para completar esta descripción del espacio de diseño de los procedimientos distribuidos de equilibrado dinámica de carga, a continuación indicamos los parámetros que deben fijarse para especificar completamente las alternativas de las distintas políticas. Estos parámetros son:

- *La granularidad, GRN*. Este parámetro indica el número de tareas en que se divide el problema. Puede variar entre 1 y N, donde N es el número máximo de tareas que pueden considerarse en el problema. Un valor bajo de GRN indica una granularidad gruesa dado que se crean pocas tareas y, por tanto, un valor grande indica una

granularidad fina. Se utiliza un parámetro entre 0 y 1 que relaciona el valor de N y el de GRN.

- *Los umbrales, TSD1 y TSD2*. Corresponden a los valores utilizados como umbrales en el caso de una política de localización con un umbral (TSD1) o con dos umbrales (TSD1 es el umbral inferior y TSD2 el umbral superior). Los valores de los umbrales están entre 1 y GRN. Se utilizan los parámetros TSDP1 y TSDP2, con valores entre 0 y 1, para establecer la relación entre TSD1 y TSD2, respectivamente, con GRN.
- *La frecuencia de equilibrado de carga, LBF*. Es un parámetro que puede variar entre 1 y GRN (el parámetro LBP es un parámetro entre 0 y 1 que relaciona LBF y l granularidad). Indica que durante la ejecución del programa paralelo, un procesador puede iniciar operaciones de equilibrado de carga cada LBF intervalos de tiempo.

En la Tabla 3.1 se indican los parámetros que se necesitan en cada uno de los 16 procedimientos de la Figura 3.3. En la tabla, IP significa política de información, TP política de transferencia, LP política de localización, y SP política de selección. El código asignado a cada alternativa en la Tabla 3.1 es el siguiente:

IP=0	política de información a demanda
IP=1	política de información periódica
TP=0	política de transferencia iniciada por emisor
TP=1	política de transferencia iniciada por receptor
TP=2	política de transferencia simétrica
LP=0	Política de localización con un umbral
LP=1	Política de localización con dos umbrales
LP=2	Política de localización con mínima/máxima carga
LP=3	Política de localización aleatoria
SP=0	Política de selección con corte forzado (preemptive)
SP=1	Política de selección sin corte forzado (non preemptive)

Tabla 3.1. Parámetros utilizados en los procedimientos de distribución de carga posibles indicados en la Figura 3.3

IP	TP	LP	SP	TSD1	TSD2	GRN	LBF
0	0	0	0	X		X	X
			1	X		X	X
		2	0			X	X
			1			X	X
		3	0			X	X
			1			X	X
	1	0	0	X		X	X
			1	X		X	X
1	2	0	0	X		X	X
			1	X		X	X
		1	0	X	X	X	X
			1	X	X	X	X
		2	0			X	X
			1			X	X
		3	0			X	X
			1			X	X

Tabla 3.2. Definiciones, rango, y significado de parámetros de los procedimientos de distribución de carga

Parámetro	Definición/Rango	Significado
N	-	***Tamaño del problema.*** Estimación del número máximo de tareas en las que se puede dividir el problema.
P	-	***Número de procesadores***
GRN	$\lceil GRANP \times N \rceil$ $(0<GRANP<=1)$	***Granularidad.*** El problema se divide inicialmente en GRN tareas que se introducen en una cola de trabajos
TSD1	$\lceil TSDP1 \times GRN \rceil$ $(0<TSDP1<1)$	***Umbral inferior.***
TSD2	$\lceil TSDP2 \times GRN \rceil$ $(0<TSDP2<1)$	***Umbral superior.***
LBF	$\lceil LBP \times GRN \rceil$ $(0<LBP<1)$	***Frecuencia de distribución de carga.***
IC		***Índice de carga.*** Número de tareas que residen en la cola de trabajos del procesador (cola_trabajos).
LBC		***Contador de equilibrio de carga.***
p		Índice del procesador actual

3.3.2 Un procedimiento genérico de distribución de carga

En esta sección se describe un *procedimiento general* de equilibrado dinámico y distribuido de carga. En la descripción se considerarán únicamente los aspectos relevantes relacionados con el procedimiento de equilibrado de carga, sin llegar a proporcionar detalles de la implementación específica que puedan hacer más complejo el seguimiento de las características del procedimiento que interesan aquí. Las definiciones, el rango y el

significado de los parámetros y variables que se utilizan en los procedimientos se proporcionan en las Tablas 3.2 y 3.3.

```
while (not final) do
{
        if (LBC==LBF) then
        {
                distribucion_de_carga();
                LBC=0;
        }
        if (cola_trabajos no vacía) then
        {
                procesar_tarea();
                LBC=LBC+1;
                IC=IC-1;
        }
}
```

Figura 3.4. Llamada a la función de distribución de carga

En la Figura 3.4 se muestra la llamada al procedimiento de distribución de carga. Como se puede ver, cada LBF intervalos de tiempo de procesamiento se realiza una llamada a la función de distribución de carga, *distribución_de_carga()*. Lo cual no quiere decir que se tengan que iniciar forzosamente las operaciones de redistribución de carga. Como vemos, cada intervalo de tiempo es igual al tiempo de procesamiento de una tarea. Por eso cada vez que se termina una tarea se reduce el índice de carga del procesador, IC, en una unidad. Obviamente, el tamaño de la tarea, y por tanto, la duración del intervalo de tiempo dependerá de la granularidad, GRN. Como se muestra en la Tabla 3.2, existe una relación entre LBF y GRN que se puede controlar con el parámetro LBP, que toma valores entre 0 y 1.

Tabla 3.3. Notación y significado de parámetros que se utilizan en el procedimiento descrito (Figuras 3.4-3.9)

Parámetro	Significado
E	Número de procesadores seleccionados como emisores por la política de localización
R	Número de procesadores seleccionados como receptores por la política de localización
CE	Suma de los índices de carga (IC) de todos los procesadores emisores
CR	Suma de los índices de carga (IC) de todos los procesadores receptores
IC	Índice de carga. Número de tareas que residen en la cola de trabajos del procesador

SC	Sobrecarga. $SC = IC - \frac{CE + CR}{E + R}$
$W(T_i)$	Carga de trabajo asociada a la tarea T_i

En la Figura 3.5 se describen las llamadas que realiza el procedimiento de distribución de carga a las funciones que implementan las distintas políticas. Como se puede ver, las llamadas que finalmente se realizan dependen del parámetro con que se esté haciendo la ejecución del programa para cada una de las políticas que intervienen en la distribución de carga. Así, en la función *distribucion_de_carga()* aparecen en primer lugar tres sentencias condicionales, cada una de las cuales corresponde a una de las tres alternativas que se consideran para la política de transferencia.

En el caso de que la transferencia sea iniciada por el emisor, el procesador comprueba si su índice de carga, IC, es mayor que el umbral de carga superior, TSD. Si se utiliza una política de localización con un umbral, TSD será igual a TSD1, y si se utiliza una política de localización con dos umbrales, TSD se habrá fijado igual a TSD2. Si, efectivamente, es mayor que el umbral que indica que el procesador está sobrecargado, se convierte en emisor. En este caso llama a la función *solicita_localizar_receptor()*, que se describe más adelante (Figura 3.8), para que le proporcione el receptor o el conjunto de receptores a los que debe enviar tareas, que se determinan posteriormente utilizando las funciones que siguen: *intercambiar_informacion_carga()*, *seleccionar_tareas()*, y *enviar_tareas()*. Con respecto a la información intercambiada, aparte del índice de la carga, IC, al iniciar el sistema los nodos intercambian entre si informaciones como la velocidad del CPU, la memoria disponible, tamaño del caché. Estas informaciones permiten disponer de un mejor conocimiento del estado de los procesadores, y pueden utilizarse en la localización de los procesadores adecuados para transferir carga. Además se pueden realizar los cálculos de mínimos, máximos de carga, etc. que, como veremos, se van a necesitar en algunas políticas.

```
distribucion_de_carga()
{
        if (politica_transferencia=iniciada_por_emisor)
        {
                if (IC>TSD) then
                {
                        solicita_localizar_receptor(p);
                        intercambiar_información_carga (proc_receptor);
```

```
                    seleccionar_tareas();
                    enviar_tareas(proc_receptor);
                }
        }
        if (politica_transferencia=iniciada_por_receptor)
        {
                if (IC<TSD) then
                {
                    solicita_localizar_emisor(p);
                    intercambiar_información_carga (proc_emisor);
                    solicitar_tareas (proc_emisor);
                    recibir_tareas(proc_emisor);
                }
        }
        if (politica_transferencia=iniciada_simétricamente)
        {
                    intercambiar_información_carga ();
                    localizar_emisor_receptor();
                    if (p=proc_emisor)
                    {
                        seleccionar_tareas();
                        enviar_tareas(proc_receptor);
                    }
                    if (p=proc_receptor)
                    {
                        solicitar_tareas(proc_emisor);
                        recibir_tareas(proc_emisor);
                    }
        }

}
```

Figura 3.5. Procedimiento general de distribución de carga

En el caso de que la transferencia sea iniciada por el receptor, en primer lugar se llama a la función solicita_localizar_emisor(), que describirá más adelante (Figura 3.9), para identificar el posible emisor del que pueda recibir las tareas que necesita el procesador. A continuación se intercambia información de carga con el emisor o emisores posibles a través de la función intercambiar_informacion_carga(), y se solicita el envío de tareas mediante la función *solicitar_tareas()*. Después, el procesador espera recibir estas tareas antes de proseguir con su procesamiento.

Si la transferencia se inicia simétricamente, tras intercambiar información de carga, se llama a la función *localizar_emisor_receptor()*, que se muestra en la Figura 3.7. Esta función determina si el procesador que hace la llamada es emisor o receptor y, en cada caso, devuelve uno o varios receptores o emisores, respectivamente. Si la función *localizar_emisor_receptor()* indica que el procesador no es ni emisor ni receptor, el procesador no interviene en la redistribución de carga. Si el procesador es seleccionado como emisor llama a la función de *seleccionar_tareas()* y a la de *enviar_tareas()*, y si es designado como receptor llama a las funciones *solicitar_tareas()* y después espera recibirlas al llamar a *recibir_tareas()*.

```
seleccionar_tareas()  {* Sólo la llevan a cabo los procesadores emisores *}
{
        if (seleccion=preemptive)
        {
                elegir(T);
                {* T= conjunto de tareas Ti con W(Ti)≥SC incluyendo la tarea en
                ejecución *}
        }
        if (seleccion=no_preemptive)
        {
                elegir(T);
                {* T= conjunto de tareas Ti con W(Ti)≥SC sin la tarea en ejecución *}
        }

}
```

Figura 3.6 Función con las políticas de selección de tareas

La función *seleccionar_tareas()* permite determinar el conjunto de tareas que pueden considerarse a la hora de elegir aquellas que deben emitirse (Figura 3.6). De esta forma, en la función aparecen dos opciones que corresponden a las dos alternativas consideradas para la política de selección de tareas: con corte forzado (*preemptive*) o sin corte forzado (*no_preemptive*). Además, en la función *seleccionar_tareas()* se llama a la función *elegir()*. Esta función es la que implementa la política de distribución, por lo que su forma depende de las característica concretas del tipo de procedimiento que se considere (*difusión, intercambio dimensional*, etc.), y que aquí no vamos a describir. Como se puede ver en la Figura 3.6, la función *elegir()* se llama con el parámetro T, que se refiere al conjunto de tareas que pueden

ser elegidas según la política de selección (con o sin corte forzado) y, en la implementación que hemos realizado, además tengan asociado un volumen de trabajo superior a un índice de carga *SC*, cuya definición aparece en la Tabla 3.2. Como se puede ver, en el cálculo de SC intervienen aspectos relacionados con la topología de conexión que se considere entre los procesadores (todos, los vecinos, los pertenecientes a un grupo, etc.). A la hora de decidir esta topología habrá que tener en cuenta el tipo de red que utiliza la plataforma paralela. En el caso de que el sistema de interconexión permita la conexión directa de todos los procesadores con todos, el impacto de la topología en la sobrecarga de comunicación del procedimiento será menos relevante.

A continuación pasamos a describir la forma que tienen las funciones que permiten implementar la política de localización. Como se ha puesto de manifiesto, la forma de esas funciones es distinta según el tipo de política de transferencia que se utilice. De esta forma, se pone de manifiesto la interrelación entre las dos políticas.

En la Figura 3.7 se muestra la función localizar_emisor_receptor(), que se llama cuando la política de transferencia es simétrica. Como se puede ver, aparecen las cuatro opciones correspondientes a las cuatro alternativas que teníamos para la política de localización. Así según el índice de carga IC esté por debajo o por encima de TSD1 el procesador actuará como receptor o emisor, respectivamente, en el caso de que la política de localización de un umbral. En el caso de una política de localización de dos umbrales se hace lo mismo pero se utilizan los umbrales TSD1 como umbral de carga baja y TSD2 como umbral de sobrecarga. Si la política de localización se basa en la mínima/máxima carga hay que comparar el índice de carga con los valores de carga máxima y mínima del conjunto de procesadores con los que un procesador puede intercambiar carga. Estos cálculos se hacen cuando se realizan las llamadas a la función *intercambiar_información_carga()* en el caso de que se utilice esta política de localización. El conjunto de procesadores que intervienen a la hora de hacer este cálculo viene determinado por la alternativa de *topología* de la política de información. Por supuesto, este cálculo implica una comunicación entre los procesadores que intervienen según la política de información cuyo coste depende del sistema de interconexión entre los procesadores.

Finalmente, la Figura 3.7 muestra la forma determinar el emisor y receptor en el caso de una política de localización aleatoria. También en este caso es preciso tener información de la carga máxima y mínima del correspondiente conjunto de procesadores.

```
localizar_emisor_receptor()
{
        if (politica_de_localizacion=basada_en_un_umbral)
        {
                if (IC>TSD1) { proc_emisor=p; proc_receptor=recepcion(); }
                if (IC<TSD1) proc_receptor=p; proc_emisor=emision(); }
        }
        if (politica_de_localizacion=basada_en_dos_umbrales)
        {
                if (IC>TSD2) proc_emisor=p; proc_receptor=recepcion();}
                if (IC<TSD1) proc_receptor=p; proc_emisor=recepcion();}
        }
        if (politica_de_localizacion=carga_min_max)
        {
                if (IC=carga_maxima) proc_emisor=p; {* procesador con mayor IC *}
                if (IC=carga_minima) proc_receptor=p; {* procesador con menor IC *}
        }
        if (politica_de_localizacion=aleatoria)
        {
                if (IC=carga_maxima)
                {
                        proc_emisor=p; {* procesador con mayor IC *}
                        proc_receptor=random(P); {* receptor aleatorio distinto de p *}
                }
                if (IC=carga_minima)
                {
                        proc_receptor=p; {* procesador con menor IC *}
                        proc_emisor=random(P); {* emisor aleatorio distinto de p *}
                }

        }
}
```

Figura 3.7. Políticas de localización llamadas desde la política de transferencia simétrica

En la Figura 3.7, las funciones *emisión()* y *recepción()* se llaman para determinar el índice (los índices) del emisor (posibles emisores) o receptor (posible receptores), respectivamente, para el procesador designado como receptor o emisor. Estas funciones dan lugar a comunicación entre aquellos procesadores que pueden transferirse tareas según las

alternativas topológicas de la política de distribución. Desde estas funciones, además, también se puede llamar a las funciones *localizar_emisor()* (en el caso de *emisión()*) y *localizar_receptor()* (en el caso de *recepción()*), que se describen a continuación. Como se puede ver, aunque aquí utilizaremos la misma política de localización para emisores y receptores, se podrían utilizar políticas distintas, con algunas restricciones de combinación entre ellas, o ciertas modificaciones en alguna de las políticas.

```
localizar_receptor(procesador)
{
        proc_receptor=-1;
        if (politica_de_localizacion=basada_en_un_umbral)
        {
                if (IC<TDS1) proc_receptor=p;
        }
        if (politica_de_localizacion=carga_min_max)
        {
                if (IC=carga_minima) proc_receptor=p; {* procesador con menor IC *}
        }
        if (politica_de_localizacion=aleatoria)
        {
                proc_receptor=random(P); {* receptor aleatorio distinto de p *}
        }
        if (proc_receptor!=-1) enviar(procesador,proc_receptor,IC)
}
```

Figura 3.8. Política de localización llamada desde una política de transferencia iniciada por emisor

La Figura 3.8 corresponde a la función *localizar_receptor()*, que permite determinar un procesador receptor en la política de transferencia iniciada por el emisor. Además de desde la función *recepción()* que hemos visto, esta función se llama desde la función *solicitar_localizar_emisor()* que genera las llamadas a *localizar_emisor(procesador)* desde *procesador* en todos aquellos procesadores que la política de distribución indique que pueden intercambiar trabajo con *procesador*. La función ejecutada en el procesador *p* envía a *solicitar_localizar_emisor()* en *procesador* el índice del procesador, *p,* si verifica de las condiciones para ser receptor que establece la política de localización que se utiliza.

Por último, la Figura 3.9 muestra la función *localizar_emisor()*, análoga a la función *localizar_receptor()* pero para el caso el procesador o procesadores a los que se les puede

solicitar tareas. Esta función se llama desde *solicitar_localizar_emisor()* en el caso de que la política de transferencia sea iniciada por el receptor (además se puede llamar desde *emisión(),* como hemos visto).

```
localizar_emisor(procesador)
{
        proc_emisor=-1;
        if (politica_de_localizacion=basada_en_un_umbral)
        {
                if (IC>TDS1) proc_emisor=p;
        }
        if (politica_de_localizacion=carga_min_max)
        {
                if (IC=carga_maxima) proc_emisor=p; {* procesador con mayor IC *}
        }
        if (politica_de_localizacion=aleatoria)
        {
                proc_emisor=random(P); {* emisor aleatorio distinto de p *}
        }
        if (proc_emisor!=-1) enviar(procesador,proc_emisor,IC)
}
```

Figura 3.8. Política de localización llamada desde una política de transferencia iniciada por receptor

3.4 Conclusiones

Este capítulo se ha centrado en los procedimientos distribuidos de equilibrado dinámico de carga. Una vez descritas las características generales de este tipo de procedimientos, se han contemplado éstos desde las necesidades de comunicación que presentan, y se han descrito los procedimientos más relevantes de este tipo que se han propuesto en la literatura.

A continuación se ha planteado el problema de encontrar el procedimiento de distribución de carga más eficiente para una plataforma y una aplicación dada. Se trata de un problema de optimización de gran complejidad debido a la propia naturaleza dinámica del proceso de distribución de carga, a la cantidad de alternativas dentro de las diversas políticas que deben implementarse y entre las que habría que decidir las mejores, y a la falta de modelos de prestaciones lo suficiente precisos y manejables. A esto habría que añadir la dependencia respecto a las características de la aplicación paralela de que se trate y de la plataforma paralela donde se ejecute.

Teniendo en cuenta las características que se han mencionado, las algoritmos de computación evolutiva pueden constituir una herramienta adecuada para explorar el espacio de diseño de los procedimientos distribuidos de equilibrado dinámico de carga, y extraer conclusiones respecto a las propiedades de las distintas alternativas. Por tanto, en relación con el equilibrado de carga distribuido, nuestro propósito es utilizar los algoritmos evolutivos para generar procedimientos eficientes más que desarrollar un procedimiento de distribución de carga basado en un algoritmo evolutivo ya que consideramos que los requisitos de tiempo de cómputo y comunicación que tendría no lo harían competitivo con otros procedimientos ya propuestos.

La metodología a seguir para aplicar la computación evolutiva al problema de optimización de un proceso dinámico se puede asimilar a la aproximación de la programación genética, rama de los algoritmos genéticos que busca optimizar programas de computador. De hecho, lo que se plantea en nuestro caso es la optimización de un programa: el programa que describe el procedimiento de distribución de carga. Además, en este caso se trata de un programa distribuido, ya que estamos considerando procedimientos distribuidos de equilibrado dinámico de carga. Hasta donde conocemos, esta forma de abordar el problema

del diseño de un procedimiento de equilibrado dinámico de carga no se ha planteado en estos términos en otros trabajos previos.

Para implementar nuestra aproximación al problema se ha realizado un programa que recoge las distintas alternativas de implementación de las políticas que concurren en un procedimiento distribuido de equilibrado dinámico de carga. Se puede decir que constituye un *procedimiento general de equilibrado dinámico de carga* (procedimiento distribuido). También se ha propuesto una taxonomía en la que ubicar los distintos procedimientos propuestos en la literatura. Por ejemplo, incluye procedimientos propuestos en la literatura como:

1. Procedimientos con transferencia iniciada por el emisor que actualizan la información a demanda. Dentro de este tipo se pueden distinguir tres procedimientos que pueden utilizar selección de tareas con o sin corte forzado. Uno de los algoritmos localiza al receptor aleatoriamente, otro está basado en el uso de un umbral, y el tercero determina el receptor buscando la cola de trabajo más corta entre los procesadores que determine la topología establecida por la política de distribución.
2. Procedimientos con transferencia iniciada por el receptor que utilizan intercambio de información en demanda. En este caso se distinguen dos algoritmos que también pueden realizar la selección de tareas con o sin corte forzado. Uno de los algoritmos utiliza una política de localización con un valor umbral y el otro selecciona aleatoriamete.
3. Procedimientos con transferencia simétrica que utilizan intercambio de información periódica. Aquí se pueden distinguir cuatro algoritmos, cada uno de los cuales corresponde a una de las cuatro alternativas consideradas para la política de localización: basada en uno o dos umbrales, basada en el valor mínimo/máximo de la carga, y aleatoria. Para cada procedimiento se pueden distinguir dos variantes según se utilice una política de selección de tareas con corte forzado o sin corte forzado.

Además, la taxonomía propuesta hace posible la definición de nuevos procedimientos al sugerir combinaciones que pueden no haber sido consideradas con anterioridad, y proporciona una caracterización del procedimiento de equilibrado de carga que permite utilizar los algoritmos evolutivos para explorar el espacio de posibilidades (espacio de diseño)

a partir de la correspondiente codificación. Al asignar los valores correspondientes a cada procedimiento de equilibrado de carga según la taxonomía propuesta, el procedimiento general de equilibrado dinámico de carga implementa el procedimiento específico en cuestión, cuyas prestaciones pueden pasar a ser evaluadas.

En el siguiente capítulo se describe el algoritmo genético que se ha desarrollado para explorar el espacio de diseño de los procedimientos de equilibrado de carga que define la taxonomía propuesta. También se muestran los resultados experimentales obtenidos en un cluster de computadores y, a partir de dichos resultados, se extraen las conclusiones relativas tanto al espacio de los procedimientos de equilibrado de carga, como a la metodología propuesta.

Capítulo 4

Exploración de las estrategias distribuidas de equilibrado de carga

En este capítulo se describe un algoritmo genético para explorar el espacio de diseño de los procedimientos distribuidos de equilibrado dinámico de carga que define la taxonomía presentada en el Capítulo 3. Esta taxonomía se define a partir de las alternativas de implementación de las distintas políticas que concurren en el diseño de un procedimiento de equilibrado de carga. A través del algoritmo genético se pueden extraer bastantes conclusiones sobre el comportamiento de los distintos tipos de procedimientos de equilibrado de carga para diferentes aplicaciones y plataformas.

De esta forma, el capítulo se ha estructurado de la siguiente forma. En la Sección 4.1 se realiza una revisión de las propuestas que han planteado la aplicación de algoritmos evolutivos en el marco del equilibrado dinámico y distribuido de la carga de una aplicación paralela y se sitúa en ese contexto el trabajo que se ha realizado en esta Tesis. Después, la Sección 4.2 describe el algoritmo genético que se ha elaborado para realizar la exploración del espacio de diseño de los procedimientos de equilibrado de carga, para pasar a describir los resultados obtenidos en el trabajo experimental realizado en la Sección 4.3. En esa misma Sección 4.3 se describen las aplicaciones que se han utilizado como programas de prueba y las características de la plataforma utilizada. A partir de los resultados experimentales obtenidos en la Sección 4.3 también se pueden evaluar las posibilidades de la metodología que proponemos en esta memoria y, por tanto, establecer futuras líneas de trabajo. Para terminar, la Sección 4.4 resume los principales resultados y las conclusiones que se pueden extraer del trabajo realizado en el capítulo.

4.1 Computación evolutiva y equilibrado de carga distribuido

En la Sección 1.4 se justifica el uso de técnicas basadas en algoritmos evolutivos para abordar los problemas que plantea el equilibrado dinámico de carga en las plataformas paralelas. Si la evolución natural es capaz de generar soluciones, es decir especies animales, perfectamente adaptadas a entornos complejos, es razonable pensar que los algoritmos evolutivos, inspirados en los principios de la evolución, puedan ayudarnos a resolver problemas complejos de ingeniería que implican encontrar soluciones eficientes en situaciones que se ven afectadas por perturbaciones más o menos aleatorias, dinámicas no lineales, condiciones variables, etc. Para la mayoría de esos problemas los algoritmos tradicionales de optimización no son adecuados porque requieren que el problema pueda formularse de manera que se satisfagan una serie de condiciones que, en muchos casos no se verifican en las aplicaciones realistas. Por ejemplo, la búsqueda de una solución óptima por el método del gradiente necesita que se pueda asignar al problema una función de coste diferenciable que no puede reflejar el efecto de los cambios discontinuos y repentinos [FOG00]. Los algoritmos evolutivos no imponen condiciones a las funciones de coste. Lo único que deben asegurar es que a través de ellas se pueda determinar si una solución es mejor o peor que otra. De esta forma, la computación evolutiva permite abordar problemas que se encuentran fuera de las posibilidades de las técnicas numéricas *tradicionales*.

Por otra parte, los algoritmos evolutivos presentan otras ventajas. Una de ellas es la posibilidad de encontrar soluciones suficientemente buenas (aunque quizá no las óptimas) a problemas de gran complejidad (problemas NP-completos), en tiempos razonables. Otra ventaja es su adaptabilidad a situaciones cambiantes. Por ejemplo, si hay que establecer una asignación óptima de tareas a procesadores habrá que tener en cuenta el número de procesadores disponible y su capacidad, las características de las tareas, etc. No obstante, en la mayoría de las situaciones realistas hay que tener en cuenta la posibilidad de fallos en los procesadores y demás recursos, cambios en la capacidad de cómputo por modificaciones en la carga del computador, etc. Recoger toda esta casuística para poder aplicar una aproximación tradicional puede ser excesivamente complejo. En un algoritmo evolutivo, la población de soluciones con la que se está trabajando en un momento dado sirve de punto de partida, o de base de conocimiento adquirido, para reiniciar la adaptación en estos entornos dinámicos.

Sin embargo, a pesar de las propiedades tan prometedoras de la computación evolutiva no existen demasiados trabajos que la apliquen al problema del equilibrado dinámico y distribuido de la carga en plataformas paralelas.

En [HOV03] se aborda el problema del coste de comunicación asociado a la determinación de los estados de carga de los procesadores en los procedimientos distribuidos de equilibrado dinámico. Ese coste de comunicación global tendría una complejidad de orden P^2 para una plataforma con P procesadores, lo cual da una idea cual sería de su volumen en el caso de plataformas distribuidas con un número elevado de procesadores. En [HOV03] se presenta un algoritmo evolutivo que intenta construir subgrupos de procesadores que reducen el coste de comunicación. Cada procesador tiene una vecindad definida por un círculo cuyo centro está en el propio procesador y cuyo radio determina al tamaño de la vecindad en un espacio bidimensional de direcciones (ubicación del procesador en el computador paralelo, y del computador en el ámbito de un GRID). Todos los procesadores de una vecindad pueden cooperar en la ejecución de las tareas transfiriendo tareas de unos a otros: un procesador sólo acepta tareas y solicitudes de los nodos de su vecindad. El algoritmo evolutivo que se propone permite que cada nodo ajuste su vecindad incluyendo o excluyendo nodos en la misma.

En el ámbito del equilibrado de carga distribuido existen propuestas de procedimientos basados en algoritmos evolutivos [SER97,SER98,SER99] pero sólo abordan el problema desde el punto de vista estático. En [SER99] se describe un paradigma para la computación evolutiva paralela y distribuida y se considera su aplicación en los problemas de asignación y planificación estática de tareas. Se considera el sistema como sistema multi-agente que implementa modelos basados en la teoría de juegos para la interacción entre los agentes. Cada agente participa en un juego no-cooperativo con una función de ganancia y un conjunto de acciones. Cada agente está interesado en maximizar sus ganancias, por lo que se debe compatibilizar esto con el objetivo de obtener un comportamiento global para el sistema, que resuelva el problema de optimización. Para conseguir esto en [SER99] se proponen distintos esquemas, y entre ellos están dos procedimientos coevolutivos que denomina algoritmos genéticos débilmente acoplados. La descomposición del criterio de optimización global en criterios locales para cada agente se realiza mediante programación genética. En [SER97, SER98] se muestran partes del trabajo más completo que se describe en [SER99]. La conexión del trabajo que se describe en esta tesis y el trabajo presentado en [SER97, SER98,

SER99] se encuentra en el objetivo de resolver un problema de optimización global de forma distribuida y en el uso de una aproximación inspirada en la programación genética. La aplicabilidad de esta aproximación a la distribución de carga dinámica no es trivial dado el tiempo que requeriría la toma de decisiones por parte de cada uno de los agentes evolutivos. En cualquier caso consideramos que se trata de una tendencia interesante para continuar el trabajo de investigación realizado.

El trabajo que se ha realizado en esta tesis corresponde a un planteamiento diferente de los trabajos que se han indicado. En nuestro caso se pretende utilizar los algoritmos evolutivos para determinar, dado un procedimiento distribuido de equilibrado de carga, los valores óptimos para sus parámetros de manera que se obtenga el mejor rendimiento. Para ello, se ha propuesto una taxonomía que parametriza el espacio de diseño de los procedimientos de equilibrado dinámico y distribuido de la carga, y permite personalizar las características de un procedimiento genérico que se ha desarrollado, de forma que se puedan utilizar algoritmos evolutivos en la exploración del mencionado espacio de diseño. En la siguiente sección se describen las características del algoritmo genético desarrollado.

4.2 El algoritmo genético utilizado

En esta sección se describe el algoritmo genético que se ha utilizado para explorar el espacio de diseño de los procedimientos distribuidos de equilibrado dinámico de carga. Más exactamente, el algoritmo genético que se ha desarrollado trabaja en el espacio definido por las distintas alternativas de las políticas de información, transferencia, localización, y selección, más los valores que pueden tomar los parámetros LBF, GRN, y los umbrales TSD1 y TSD2. Su objetivo es encontrar las configuraciones de alternativas en relación con las políticas y los valores de los parámetros para dichas políticas que proporcionen una mejor distribución de carga para una aplicación paralela. Es decir, de forma que se consiga la mayor ganancia de velocidad y la mejor utilización de la plataforma paralela.

Teniendo en cuenta los pasos de un algoritmo genético y los elementos que lo definen, tal y como se presentaron en el Capítulo 2, realizar la descripción del algoritmo genético implica detallar los aspectos relacionados con la codificación de los individuos, las características de las operaciones de cruce y mutación que se aplican a la población, así como

la forma de llevar a cabo la selección. El algoritmo genético que se ha desarrollado consta de los pasos que se muestran en la Figura 4.1, a continuación describimos la forma de representar las soluciones (Sección 4.2.1), la forma de generar la población inicial (Sección 4.2.2), la función de idoneidad utilizada (Sección 4.2.3), y las características de los operadores de selección, cruce y mutación que se han utilizado (Sección 4.2.4).

1. ***Inicializar parámetros:*** Tamaño de la Población, Probabilidades de Cruce y Mutación.
2. ***Generar una población inicial*** aleatoriamente.
3. ***Evaluar la idoneidad de los individuos de la población.*** La idoneidad de un individuo es la ganancia de velocidad que consigue el procedimiento de distribución de carga que representa.
4. ***Seleccionar una nueva población.*** La selección utiliza el método de la ruleta, pero tomando siempre el mejor individuo para la siguiente iteración.
5. ***Aplicar operadores.*** Se realiza cruce en un punto y mutación mediante el operador de mutación descrito en 4.2.4.
6. ***Ir a 3 si no se ha alcanzado la condición de final.***

Figura 4.1. Pseudocódigo del algoritmo genético desarrollado

4.2.1 Representación de las soluciones

Un algoritmo genético maneja una población de *cromosomas*, cada uno de los cuales representa una solución factible del problema. Un cromosoma está formado por una cadena de genes que usualmente se representan en binario, aunque a veces es más adecuado para las características del problema utilizar una representación entera o real. En nuestro caso se ha utilizado una representación real para los genes que corresponden a los parámetros LBF, GRN y los umbrales, TSD1 y TSD2. Concretamente, y como se muestra en la Tabla 4.1, los valores de GRN están relacionados con los del tamaño del problema a través de un factor que denominamos GRANP, un número real entre 0 y 1, que es el que se utiliza como gen. De la misma forma LBF, TSD1, y TSD2 están relacionados con GRN a través de los factores LBP, TSDP1, y TSDP2, respectivamente, que también son números reales entre 0 y 1, y son los que se utilizan en los genes correspondientes.

En un problema de tamaño fijo, el volumen de trabajo se puede repartir entre GRN tareas, considerándose cada una de esas tareas como una unidad de computación. Por ejemplo, si consideramos la multiplicación de dos matrices de tamaño N×N cada una en un sistema con *P* procesadores, el número total de tareas en que se puede dividir el trabajo, GRN, puede variar desde un valor igual al número de procesadores empleados, P, hasta N (si se toma como tarea más pequeña la que corresponde a la multiplicación de una fila de la primera matriz por la segunda matriz). Por tanto, una granularidad fina sería GRN = N. Como es sabido una granularidad facilita el equilibrado de carga pero puede acarrear una mayor sobrecarga comunicación. Una granularidad gruesa conlleva un menor coste de comunicación pero dificulta el equilibrado de la carga. Así pues, la granularidad que se seleccione representar un compromiso entre coste de comunicación y facilidad de equilibrado de carga. Obviamente, GRN no debe se menor que P, puesto que en este caso no se dispondría de tareas suficientes para asignar una tarea a cada procesador, la utilización de la plataforma no sería adecuada y no se conseguiría un rendimiento óptimo del programa paralelo.

Tabla 4.1. Rango y significado de parámetros de los procedimientos de distribución de carga

Parámetro	Rango	Significado
N	-	***Tamaño del problema.*** Estimación del número máximo de tareas en las que se puede dividir el problema.
P	-	***Número de procesadores***
GRN	⌈GRANP×N⌉ (0<GRANP<=1)	***Granularidad.*** El problema se divide inicialmente en GRN tareas que se introducen en una cola de trabajos
TSD1	⌈TSDP1×GRN⌉ (0<TSDP1<1)	***Umbral inferior.***
TSD2	⌈TSDP2×GRN⌉ (0<TSDP2<1)	***Umbral superior.***
LBF	⌈LBP×GRN⌉ (0<LBP<1)	***Frecuencia de distribución de carga.***

Como se ha indicado, y se observa en la Tabla 4.1, los parámetros LBF, TSD1 y TSD2, están relacionados con el propio valor de GRN. Si para un procedimiento de distribución de carga determinado se encuentran varios valores de granularidad que pueden representar comportamientos óptimos, los valores de LBF, TSD1 y TSD2 también presentarán varios valores óptimos relacionados con los correspondientes valores de GRN. En los experimentos que hemos realizado en el desarrollo de este trabajo hemos encontrado que

una granularidad aceptable para la plataforma de tipo cluster utilizada puede ser igual a P×P siempre que N > P×P. Una vez fijada la granularidad (por ejemplo a P×P) los parámetros LBF, TSD1 y TSD2, serán explorados en el espacio correspondiente (en este caso entre 1 y P×P).

4.2.2 Población Inicial

La población inicial que sirve de partida para el algoritmo genético, por lo general, se crea aleatoriamente. Ciertos estudios empíricos realizados sobre una amplia variedad de problemas de optimización de funciones [DAV91] recomiendan un tamaño de población dependiente de la complejidad del espacio de búsqueda. Como hemos dicho, cada uno de los genes reales que tenemos, GRANP, TSDP1, TSDP2, y LBP toman valores entre 0 y 1. Por otra parte, en nuestros experimentos hemos utilizado un tamaño de población es 10 individuos, aunque también hemos considerado otros tamaños en algunos experimentos. Se ha elegido este tamaño de la población porque aunque, cuando se fijan los parámetros que definen el procedimiento de distribución de carga, el tamaño del espacio de la búsqueda no es excesivamente grande, la evaluación de la idoneidad necesita mucho tiempo. Al reducir el tamaño de la población se consiguen velocidades razonables para la búsqueda que realiza el algoritmo. Además, los resultados obtenidos con el tamaño de población considerado muestran que se trata de un tamaño adecuado.

4.2.3 Evaluación de la idoneidad (*fitness*)

La calidad de las soluciones de la población se evalúa a partir de una *función de coste* o *idoneidad*, o simplemente *idoneidad*. En nuestro caso, la función de idoneidad tiene en cuenta la ganancia de velocidad, *S*, de la aplicación paralela que se ha ejecutado implementando el procedimiento de distribución dinámica de carga correspondiente al individuo de la población que se evalúa y la utilización media de los procesadores, *U*, al ejecutar la aplicación paralela. Así pues, la función de idoneidad, *F*, se define como, $f= S(N,P)\times U(N,P)$, donde $S(N,P)$ es la ganancia de velocidad obtenida con *P* procesadores para un problema de tamaño *N* y viene dada por (Sección 1.2.1):

$$S(N,P) = \frac{T(N,1)}{T(N,P)}$$

y *U(N,P)* es la utilización media de todos los *P* procesadores empleados durante la ejecución del programa paralelo, y se calcula según la expresión:

$$U(N,P) = \frac{U_1(N,P) + U_2(N,P) + ... + U_P(N,P)}{P}$$

donde $U_i(N,P)$ es la utilización del procesador *i*-ésimo, definida como

$$U_i(N,P) = \frac{T_{comp}}{T_{comp} + T_{com} + T_{ocio}}$$

donde T_{comp}, T_{com}, y T_{ocio} son, respectivamente, las componentes del tiempo paralelo que se han dedicado a cálculo, comunicación con otros procesadores, y espera a otros procesadores. Para tener una medida de la idoneidad estadísticamente significativa se determina el tiempo de ejecución del programa paralelo de prueba que se considere y se obtiene la media y la desviación estándar. Si ésta es suficientemente pequeña se considera válido el tiempo de ejecución medio obtenido y se calcula la idoneidad. En los experimentos que hemos realizado son suficientes tres ejecuciones.

La función de idoneidad que se está utilizando es similar a la que se describe en [ZOM01] (Sección 2.1.1). La diferencia es que en [ZOM01] se tiene también en cuenta el tamaño de las colas en los procesadores, apareciendo un factor más en la idoneidad que depende del cociente entre el número de colas que tienen un tamaño aceptable según un umbral (no tienen ni demasiadas tareas, ni demasiado pocas) y el número de procesadores. Por otra parte, en [ZOM01] no se utiliza la ganancia de velocidad sino sólo la inversa del tiempo de ejecución paralelo. Esto realmente no tiene más efecto que el de normalizar la idoneidad que utilizamos aquí, dividiéndola por el tiempo de ejecución secuencial, que es el mismo para todos los individuos de la población.

4.2.4 Operadores de selección, cruce, y mutación

La operación de selección permite generar una nueva población de individuos a los que se aplicarán las operaciones de cruce y mutación. La selección se realiza mediante el método de

la ruleta, descrito en [GOL89], aunque la nueva población siempre incluye al individuo más idóneo. De esta forma se mantiene la mejor solución encontrada hasta el momento. Se trata de una estrategia elitista para la que, al final de esta sección, haremos algunos comentarios relativos a su interés en problemas dinámicos como el que abordamos aquí.

Dada una población de *M* individuos, para aplicar las operaciones de cruce y mutación se van seleccionando parejas de individuos aleatoriamente. A cada pareja se aplica la operación de cruce con una determinada probabilidad, generándose dos nuevos individuos que sustituyen a los individuos de partida (los progenitores). A la pareja de individuos resultantes tras la operación de cruce (si se ha producido) o bien a la pareja de individuos que se han seleccionado (si no se produce cruce) se aplica la operación de mutación. Después los individuos resultantes se incluyen en la nueva población. Así pues, mientras que la mutación siempre se lleva a cabo, el cruce se produce con arreglo a una probabilidad. En el trabajo [DAV91], al que también nos hemos referido en la Sección 4.2.2, se proporcionan resultados experimentales que indican que los mejores resultados se consiguen por una probabilidad de cruce de entre 0.65 y 0.85, que implica que la probabilidad de un cromosoma seleccionado que sobrevive a la siguiente generación sin alterar (aparte del cambio asociado a la mutación) se extiende de 0.35 a 0.15 La transformación de cruce se implementa seleccionando aleatoriamente un punto en el cromosoma, que se divide en dos trozos cada uno con un subconjunto de los genes. A continuación se intercambian los trozos de cromosomas de ambos progenitores, de forma que el trozo izquierdo del primer progenitor y el trozo derecho del segundo progenitor constituyen uno de los descendientes, y el trozo derecho del primer progenitor y el trozo izquierdo del segundo progenitor constituyen el segundo descendiente. La probabilidad para realizar la operación de cruce entre dos progenitores seleccionados se ha fijado a 0.6 en nuestros experimentos.

Si sólo se utilizase el operador de cruce para obtener la descendencia, podría producirse un problema derivado del hecho de que si todos los individuos de la población tienen el mismo valor en una posición particular, todos los descendientes tendrán este mismo valor en esta posición con lo que se reduce la capacidad de exploración del algoritmo genético. Para evitar esta situación se utiliza el operador de mutación, que introduce cambios aleatorios en los genes. El operador de mutación que se ha utilizado está basado en el propuesto en [REN96] y, como hemos dicho se aplica siempre (probabilidad de mutación

igual a 1). Para aplicarlo se seleccionan aleatoriamente dos números, el número *p*, que puede ser +1 ó –1 con igual probabilidad, y el número *r*, que es un número real elegido aleatoriamente en el intervalo (0,1], que determina de forma indirecta la amplitud de la modificación. Dado el gen X_i, el gen mutado X'_i se obtiene a partir de la expresión:

$$X_i^{'} = X_k + (X_{max} - X_i) \times (1 - r^{(1-(t/T))^b}) \text{ si } p=+1$$

$$X_i^{'} = X_k + (X_i - X_{min}) \times (1 - r^{(1-(t/T))^b}) \text{ si } p=-1$$

donde X_{max} y X_{min} son, respectivamente el valor máximo y mínimo que puede tomar X_i. En este operador de mutación, la amplitud de la mutación va reduciéndose a medida que van sucediéndose las generaciones: el parámetro *b* determina la velocidad de decrecimiento de la amplitud de la mutación, *T* es el número de generación a partir de la cual la amplitud de la mutación se hace cero, y *t* es la generación actual del algoritmo. En nuestros experimentos se ha puesto b=5 (el mismo valor que se utiliza en [REN96]) y se ha fijado T a un valor muy elevado como para que la amplitud no se reduzca significativamente en el periodo de ejecución de nuestro algoritmo genético.

Además del operador de mutación que se ha descrito, también hemos definido un nuevo operador de mutación que permite mejorar la convergencia del algoritmo genético teniendo en cuenta las características de convergencia observadas en los parámetros codificados por los genes. Así, en los primeros experimentos realizados durante este trabajo se observaron ciertas tendencias en la evolución de LBF, TSD1 y TSD2, hacia ciertos puntos en su rango de valores. Así, los parámetros LBF y TSD1 convergen siempre a valores próximos a 1, y TSD2 converge en el entorno de GRN/P, es decir, a la carga media (en tareas) que puede tener un procesador, considerando que hay GRN tareas. Si LBF = 1, eso quiere decir que un procesador realiza una llamada al procedimiento de distribución de carga (ver Sección 3.3) cada vez que ejecuta una de las tareas que tiene asignadas. El significado de TSD1 = 1 es que se considera que un procesador está poco cargado cuando tiene una única tarea ya se está quedando sin trabajo, y el de TSD2 = GRN/P que es cuando un procesador tiene más de GRN/P tareas cuando se considera que está sobrecargado.

Para acelerar la convergencia de los parámetros también se ha implementado una mutación especial que se ha comportado muy eficazmente en los experimentos. Como se ha

dicho anteriormente, utilizamos una representación real en la que los genes del algoritmo genético varían entre 0 y 1. Por tanto, para generar los valores de los parámetros correspondientes a cada gen a la hora de presentarlos al programa de prueba, hay que transformarlos según la formula $e = \lceil r \times GRN \rceil$, donde *e* es el valor entero del parámetro que necesita el programa de prueba, *r* es el valor real utilizado en el algoritmo genético, y *GRN* es, como ya se ha dicho, la granularidad. Por ejemplo, si el programa paralelo utiliza una *GRN = 100*, y el algoritmo genético utiliza *LBP=0.2*, tendremos $LBF = \lceil 0.2 \times 100 \rceil = 20$. La operación de mutación que se ha definido se muestra en la Figura 4.2.

```
if  (mutan LBF or TSD1)      Pc = 1/GRN;
if (muta TSD2)               Pc = P/GRN;
V = abs(Xi- Pc)
if ( V > (max- Pc)) OR ( V > (Pc –min))  {* max=1, min=0*}
{
        if (Xi>Pc)      Xi+1 = Xi- V/2;
        else            Xi+1 = Xi+ V/2;
}
else
{
        if (Xi>Pc)      Xi+1 = Xi- 1.5*V;
        else  Xi+1 = Xi+ 1.5*V;
 }
```

Figura 4.2 Operador de mutación definido para el algoritmo genético utilizado

La secuencia de cruce/mutación que se ha descrito se repite hasta que se tienen nuevamente *M* individuos en la población, tras lo cual se evalúan las soluciones y se aplica otra vez la selección de soluciones según el método de la ruleta pero, como se ha indicado, manteniendo al mejor individuo. En la Figura 4.3 se ilustra la secuencia que se repite generación tras generación.

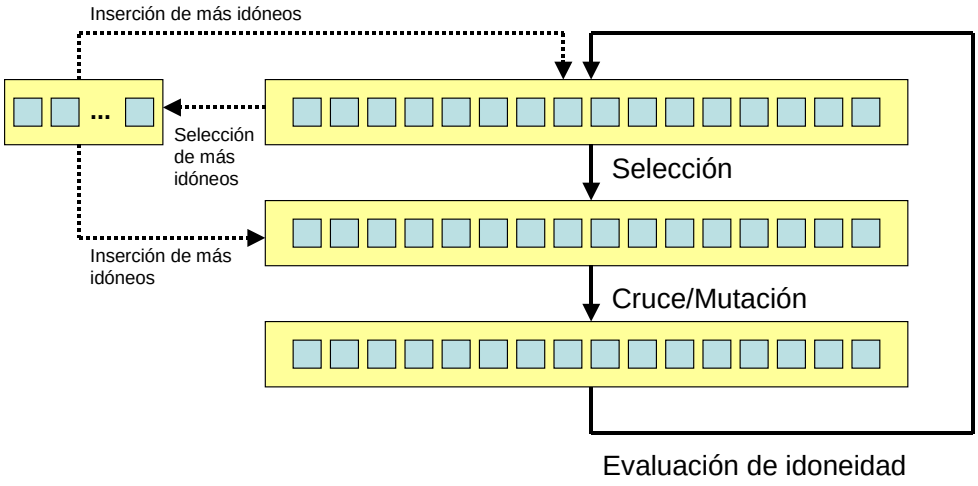


Figura 4.3. Secuencia de operaciones repetidas en cada generación

Como se ha indicado, en la población de soluciones que evoluciona generación a generación se sigue manteniendo la solución más idónea que se ha encontrado en la generación anterior. En la Figura 4.3 se representa esta estrategia *elitista*. Así, antes de aplicar el procedimiento de la ruleta para seleccionar las soluciones a las que aplicar el cruce y la mutación se extraen las *m* mejores soluciones (en nuestro caso *m*=1) que se incluyen directamente en la nueva población. A esa población se le aplica los operadores de cruce y mutación que se han descrito más arriba y, una vez obtenida la nueva población se evalúa la idoneidad de las soluciones obtenidas. Si alguna de las *m* soluciones mejores que se habían almacenado antes de aplicar los operadores sigue siendo mejor que todas las incluidas en la nueva población se introduce en dicha nueva población sustituyendo a alguna de las peores soluciones. El *elitismo* fue propuesto por primera vez en [DEJ75] como una estrategia para mantener al mejor individuo de la población en generaciones sucesivas y evitar el efecto de degradación (*disruption*) que pueden ocasionar los operadores de variación de cruce y mutación. Esta estrategia, como se ha dicho, puede extenderse a copiar los *m* individuos mejores. Los experimentos que se describen en [DEJ75] ponen de manifiesto la mejora del rendimiento de los algoritmos genéticos para funciones de idoneidad unimodales (en funciones de coste multimodales habría que introducir además mecanismos adicionales para mantener la diversidad de la población). Además, el elitismo juega un papel muy importante

en los procedimientos de optimización multiobjetivo, donde se ha comprobado que mejora las características de convergencia [ZIT98, COR00]. Precisamente, la capacidad para recordar información útil relativa a generaciones pasadas es una de las estrategias que se ha venido considerando en la aplicación de los algoritmos evolutivos en problemas de optimización dinámica, en los que las soluciones obtenidas deben tener en cuenta la naturaleza cambiante del problema [BRA01]. Así, dentro de esta aproximación a los problemas de optimización dinámica existen procedimientos están los que proponen disponer de una memoria explícita donde se almacene información (información sobre las soluciones mejores) y desde donde se recupere para las sucesivas generaciones [BRA99, MOR97], y los que utilizan un algoritmo evolutivo con una representación redundante de las soluciones [HAD97,LEW98]. En [BRA99] se ha mostrado, no obstante, que la calidad de este tipo de aproximaciones es muy dependiente de la diversidad y, por lo tanto, debe utilizarse en combinación con técnicas de mantenimiento de la diversidad [SAR98].

4.3 Resultados Experimentales

En esta sección se analizan los resultados obtenidos al aplicar el algoritmo genético que hemos desarrollado al espacio de diseño de los procedimientos distribuidos de equilibrado dinámico de carga, caracterizado por los parámetros que definen las políticas que concurren en el equilibrado de carga. No obstante, antes de pasar a la propia discusión de resultados se describen los programas de prueba que se han utilizado para realizar el trabajo experimental y se indican algunas características de la plataforma donde se ha llevado a cabo la investigación.

4.3.1. Programas de prueba utilizados

Para generar resultados experimentales que pongan de manifiesto las prestaciones de la metodología que se presenta se han considerado cuatro problemas que se abordan con programas paralelos con comportamientos diversos y, creemos, suficientemente representativos a la hora de poner de manifiesto la casuística del problema de la distribución dinámica de la carga. Además, se trata de problemas que suelen utilizarse frecuentemente en la literatura. En cualquier caso, las características del comportamiento de las aplicaciones paralelas son muy diversas, por lo que considerar algunas aplicaciones más la presentación

que se hace aquí, tampoco significaría agotar las posibilidades existentes. De todas formas, las posibilidades de la metodología que describimos quedan suficientemente claras con el conjunto de aplicaciones consideradas y siempre se podría aplicar sobre otros programas paralelos que puedan interesar en un momento dado.

Así pues, las aplicaciones paralelas que se han utilizado en nuestro trabajo experimental son la multiplicación de matrices, la integración numérica mediante el método del trapecio, el cálculo del número *pi* mediante el método de Monte Carlo, y la solución del problema del viajante de comercio mediante el método de ramificación y poda (*branch-and-bound*). A continuación se describen brevemente algunas de sus características.

Multiplicación de Matrices. Ni que decir tiene que la multiplicación de matrices aparece en gran cantidad de algoritmos utilizados para resolver problemas en ciencia e ingeniería. Por ejemplo en campos como la química computacional y en las transformaciones usadas para el procesamiento de señales aparecen multiplicaciones de matrices de gran tamaño [QUI04]. Se han desarrollado bastantes algoritmos paralelos para multiplicar matrices, considerando distintos tipos de plataformas y características de las propias matrices: densas, dispersas, a bandas, etc. [WIL99]. Aquí consideraremos el producto *C=AxB* de dos matrices cuadradas *nxn* densas, *A* y *B*. El algoritmo de multiplicación paralela de matrices que se ha implementado está basado en una descomposición de matrices orientada a filas. Así, para identificar las tareas se tiene en cuenta que cada elemento de la matriz *C* es función de elementos en *A* y *B*, y dado que *A* y *B* no se modifican se pueden calcular los elementos de *C* simultáneamente. Por otra parte, en el cálculo de cada elemento $c_{i,j}$ de *C* la fila *i* de *A* y la columna *j* de *B*:

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

Por lo tanto, con la fila *i* de la matriz *A* y todos los elementos de la matriz B, la tarea puede calcular la fila *i* de la matriz *C*. Por lo tanto, si el trabajo de una tarea se asocia al cálculo de una fila de C, para completarse, cada tarea debe tener acceso a todos los elementos de la matriz *B*. Para ello, simplemente se transmite la matriz *B* a todos los procesos antes de comenzar la multiplicación, con lo que el coste de comunicación de transmitir la matriz *B* se afecta al algoritmo sólo una vez y se puede incluir en el coste de inicialización del algoritmo.

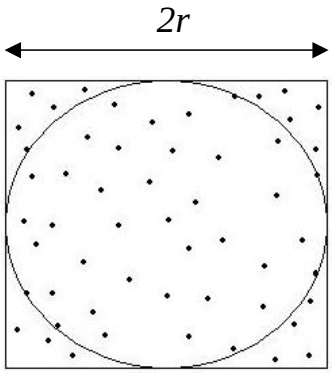
Para simplificar el análisis del algoritmo paralelo, asumimos que *n* es un múltiplo de *P* (el número de procesadores). A cada procesador se asignan las mismas filas de *A* y *C* durante la ejecución del algoritmo según establezca el procedimiento de distribución de carga y, como se ha dicho, las filas de *B* se transmiten a todos los procesadores al principio de la ejecución. Así, existe un proceso padre que distribuye las matrices inicialmente y recibe al final el resultado, además en este proceso se finan LBF, los umbrales TSD y GRN. La matriz se divide en un número de tareas igual a GRN. Durante cada iteración del algoritmo cada proceso multiplica una fila de la matriz *A* por la matriz *B*. Cada LBF iteraciones el algoritmo aplica las políticas que definen el procedimiento del equilibrio de carga, tal y como se ha descrito en el Capítulo 3.

Integración Numérica. El método del trapecio permite encontrar el valor aproximado de una integral definida, que corresponde al área bajo la curva de la función que se integra en el intervalo indicado en la integral. En este método, el área bajo la curva $y = f(x)$ en el intervalo $[a,b]$ se divide en subintervalos de tamaño *h=(b-a)/n*. El área de cada subintervalo *i* se calcula aproximadamente mediante $\frac{h}{2}(f(a+(i-1)h)+f(a+ih))$. La suma de estas áreas trapezoidales proporciona una aproximación para la integral definida $\int_a^b f(x)dx$ que, de esta forma, se define mediante la expresión $\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{h}{2}(f(a+(i-1)h)+f(a+ih))$. Esta estimación de $\int_a^b f(x)dx$ normalmente mejora cuanto mayor es el número de subintervalos n que se consideran.

Calculo del número pi. El procedimiento para calcular el número *pi* mediante el método de Monte Carlo consta de los siguientes pasos:

(1) *Inscribir un círculo en un cuadrado*
(2) *Generar puntos en el cuadrado aleatoriamente.*
(3) *Determinar el número de puntos en el cuadrado que están también en el círculo.*
(4) *Sea* ***r*** *el número de puntos en el círculo dividido por el número de puntos total en el cuadrado.*
(5) *pi ~ 4* ***r****.*

La justificación de este algoritmo se puede entender a partir de la Figura 4.4. Utilizando este algoritmo, cuantos más puntos generados, la aproximación del número es mejor, tal y como ilustra la figura 4.5. El tiempo de este algoritmo será de orden *O(n)*, donde *n* es número de puntos a generar. La paralelización de este algoritmo es bastante sencilla utilizando un modelo SPMD en el que se divide el número de puntos a generar entre los procesadores que participan en la solución del problema.



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4\frac{A_C}{A_S}$$

Figura 4.4: Cálculo del número *pi* con el método de Monte Carlo

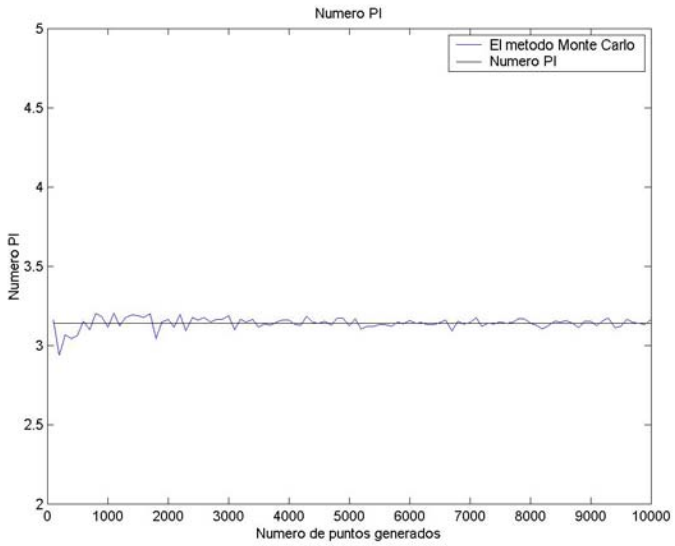


Figura 4.5: cálculo del número PI utilizando el método Monte-Carlo

Algoritmo de Ramificación y Poda para el problema del viajante. Los algoritmos de ramificación y poda (*branch-and-bound*) se pueden incluir dentro de las técnicas clásicas para

resolver problemas de búsqueda y optimización combinatoria para los que una búsqueda exhaustiva resulta imposible. Para aplicar un algoritmo de ramificación y poda, el problema a resolver se divide en una serie de subproblemas que han de resolverse. Así, las acciones del algoritmo se pueden describir mediante un árbol en el que la raíz representa el punto de partida del algoritmo, y a partir de dicho punto, cada nivel en la jerarquía del árbol representa la selección de una alternativa que proporciona una solución para uno de los subproblemas en que se ha dividido el problema original. Al ir profundizando en los niveles del árbol se va construyendo la solución que se busca.

El espacio de estados puede explorarse según el método de *primero-en-profundidad* (*depth-first*) que empieza en un nodo y continúa con los que cuelgan de él empezando primero por el de la derecha (por ejemplo) expandiendo los nodos que cuelgan de él hasta llegar al nodo del último nivel. También puede utilizarse el método de exploración de *primero-a-lo-ancho* (*breadth-first*) en el que se expanden todos los nodos de un nivel antes de empezar a expandir los nodos del siguiente nivel. No obstante, lo usual es utilizar un método de primero-el-mejor (*best-first*) que dirige el árbol podando los caminos poco prometedores. En este caso, el algoritmo no visita todos los nodos del árbol, sino que utiliza una función de acotación (*bounding function*) o de corte (*cut-off function*) que debe diseñarse para el problema al que se aplica el algoritmo, y que proporciona una cota inferior (para los problemas de minimización) y una cota superior (para los problemas de maximización) para decidir si el subárbol que cuelga del nodo en el que se encuentra el algoritmo en un momento dado debe ser explorado o no. Así, si en un problema de minimización, la cota inferior que se asigna a un nodo, *n*, es mayor que el coste de la mejor solución encontrada hasta el momento, se puede concluir que el subárbol que parte del nodo *n* no puede producir una solución óptima. El tiempo de ejecución de un algoritmo de ramificación y poda depende del número de nodos visitados, aunque ese número no se conoce hasta que termina el proceso.

Respecto a la paralelización del algoritmo de ramificación y poda, una aproximación bastante natural sería asignar una parte del espacio del árbol que parte de un nodo a un procesador, que se encargaría de realizar la búsqueda en ese área del espacio. También sería posible aprovechar el paralelismo en la evaluación de la función de acotación y en la selección de los nodos que se expanden. Sin embargo, existen una serie de dificultades que hacen la paralelización más difícil y menos eficiente de lo que pudiera pensarse inicialmente.

Por una parte, se necesita que la cota inferior o superior obtenida por la función de acotación se conozca en todos los procesadores, para que en todo momento puedan realizar una poda óptima de sus subárboles. Además, el tamaño del espacio de estados no se conoce en el momento de realizar la partición de trabajo entre los procesadores. Por esta razón, la distribución dinámica de carga constituye un elemento esencial en la paralelización de este tipo de algoritmos, y por esta razón los hemos incluido entre los problemas de prueba que utilizaremos en nuestra investigación. Para un estudio más detenido de la paralelización de los algoritmos de ramificación y poda se puede consultar [WIL05].

El algoritmo de ramificación y poda se ha aplicado al problema del viajante de comercio. Este problema consiste en encontrar un camino de longitud mínima que, partiendo de una ciudad pase por todas las ciudades de un conjunto dado, volviendo al punto de partida, y supuesto que se conocen las distancias entre cada par de ciudades. Por tanto, se trata de que, dado un grafo dirigido con arcos de longitud no negativa, se encuentre un circuito de longitud mínima que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes: lo que se llama circuito *hamiltoniano*. Este problema es un problema NP-completo que se utiliza frecuentemente para evaluar la calidad de procedimientos y heurísticas de optimización combinatoria. Para una discusión bastante completa del problema del viajante de comercio y de diferentes procedimientos para abordarlo se puede consultar [MIC00]. A continuación se muestra el pseudocódigo correspondiente a la implementación paralela de un algoritmo de ramificación y poda para el problema del viajante de comercio que utilizaremos en nuestro trabajo experimental.

Pseudocódigo del algoritmo paralelo para el problema del viajante:

1. *Enviar la matriz de distancia a todos los procesos.*
2. *Empezar en una ciudad aleatoria.*
3. *Inicializar la cola de trabajo*
4. *Tomar un nodo de la cola y expandirlo.*
5. *Comprobar los nuevos nodos:*
   - *Si se trata de una solución, comparar el coste con la cota inferior*
     - *Si es menor, reemplazar la cota.*
     - *Si es mayor, ir a 6.*
   - *Si no es una solución, introducir el nodo en la cola de trabajo.*
6. *Distribución de carga: evaluar la carga y tomar decisiones*
   (Migración de sub-árboles si hace falta, y actualización de la cota mínima que se ha obtenido en todos los procesadores)
7. *Si hay un nueva cota, realizar la poda (comprobar el coste de todos los nodos en la cola de trabajo, y si un nodo tiene coste superior a la cota actual, se elimina de la cola.*
8. *Si la exploración no han terminado en todos los procesos, ir a 4.*

Características de la plataforma utilizada. Para realizar el trabajo experimental se ha utilizado un cluster constituido por ocho nodos de tipo SMP, cada uno de los cuales incluye dos procesadores *AMD Athlon MP 2400+*. Por lo tanto, el cluster cuenta con dieciséis procesadores. La interconexión se realiza a través de un conmutador de Gigabit Ethernet, proporcionando anchos de banda de comunicación del orden de los Gigabytes/s. Cada nodo del cluster cuenta con memoria RAM de dos *GBytes*, y disco duro propio con sistema operativo Linux.

Los programas se han desarrollado utilizando el lenguaje de programación C bajo Linux, y las primitivas de paso de mensajes de la biblioteca *LAM/MPI*, que es una implementación del estándar de paso de mensajes MPI realizada por la Universidad de Indiana (http://www.lam-mpi.org/).

4.3.2. Descripción y análisis de resultados experimentales

En esta sección se presentan los resultados obtenidos a través del trabajo experimental resultado y, a partir de los datos obtenidos, se extraen una serie de conclusiones relativas, tanto a las propiedades de los distintos procedimientos de distribución de carga considerados, como a las características de la metodología que proponemos en esta Tesis.

Se van a mostrar tres tipos de estudios. En primer lugar (Sección 4.3.2.1) se determinan los valores más adecuados para los parámetros del algoritmo genético que se utilizará en el resto de las experiencias. Después, en la Sección 4.3.2.2 se comparan los distintos tipos de procedimientos distribuidos de equilibrado dinámico de carga considerando los distintos programas paralelos de prueba que se han enumerado más arriba. En esta sección se considerará la optimización de los valores de los parámetros LBF, TSD1, y TSD2 para cada tipo de procedimiento. Por último, la Sección 4.3.2.3 analiza los resultados obtenidos cuando se emplean los algoritmos genéticos para explorar todo el espacio definido por todas las alternativas para las distintas políticas y los parámetros que, al ser fijados, particularizan el procedimiento de distribución de carga.

Puesto que los experimentos se realizan en un cluster de computadores que utiliza un conmutador para Gigabit Ethernet, la topología de conexión es de todos-con-todos y, por tanto, no consideraremos alternativas que se refieran a aspectos de la topología de

interconexión en las políticas. Siempre se considera la topología de interconexión que existe físicamente.

4.3.2.1. Parámetros del algoritmo genético

Para definir el algoritmo genético que se va a utilizar deben fijarse dos parámetros: el tamaño de la población y la probabilidad de cruce. Hay que tener en cuenta que la operación de mutación se aplica siempre. Para hacer esto se han realizado diversos experimentos con distintas asignaciones de estos parámetros y considerando diversos programas de prueba para diferentes conjuntos de datos a los que están asociadas complejidades diferentes.

En primer lugar, las Figuras 4.6, 4.7, y 4.8 muestran las diez primeras generaciones de la evolución de un algoritmo genético en el espacio de búsqueda definido por los parámetros GRN, LBF y TSD y las distintas alternativas para las políticas de información, transferencia, localización y selección. Las figuras corresponden a un programa de prueba de multiplicación paralela de matrices de tamaño 1000x1000 que utiliza 12 procesadores (la máxima ganancia que se podría alcanzar estaría alrededor de doce). Cada una de esas figuras corresponde a una población de distinto tamaño. Concretamente, se muestra la evolución para 10 (Figura 4.6), 20 (Figura 4.7) y 30 individuos (Figura 4.8).

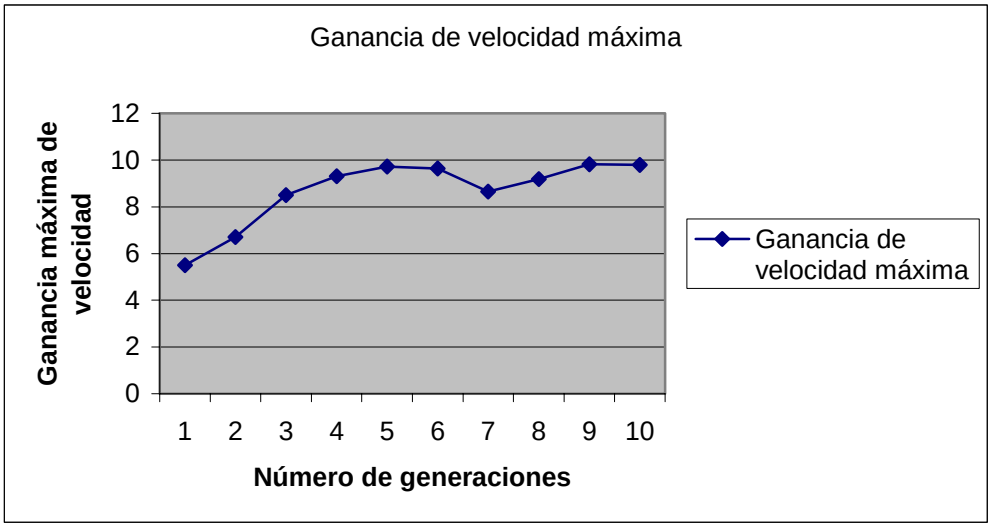


Figura 4.6: Ganancia de velocidad máxima con una población de diez individuos

La diferencia entre el mejor y el peor valor al que converge el algoritmo genético que se muestra en las Figuras 4.6-4.8 para distintas poblaciones es menor de un 3.3%. Los resultados que se han obtenido para distintos programas de prueba, con distintos niveles de complejidad en cuanto a los problemas que abordan, muestran un comportamiento similar al de las Figuras 4.6-4.8 en cuanto a las diferencias entre los valores finales que se alcanzan en la ganancia de velocidad. Es decir, se observa una variación muy pequeña el valor de idoneidad al que convergen los algoritmos genéticos para tamaños de población comprendidos entre 10 y 30 individuos. Por esta razón, la mayoría de los experimentos (y por supuesto todos aquellos que requieren un mayor número de repeticiones) se han realizado con una población igual a 10.

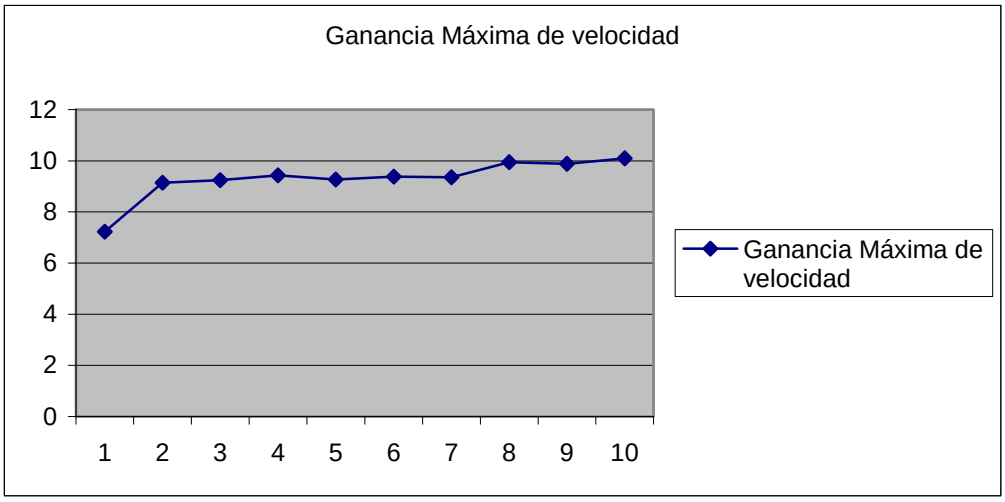


Figura 4.7: Ganancia de velocidad máxima con una población de veinte individuos

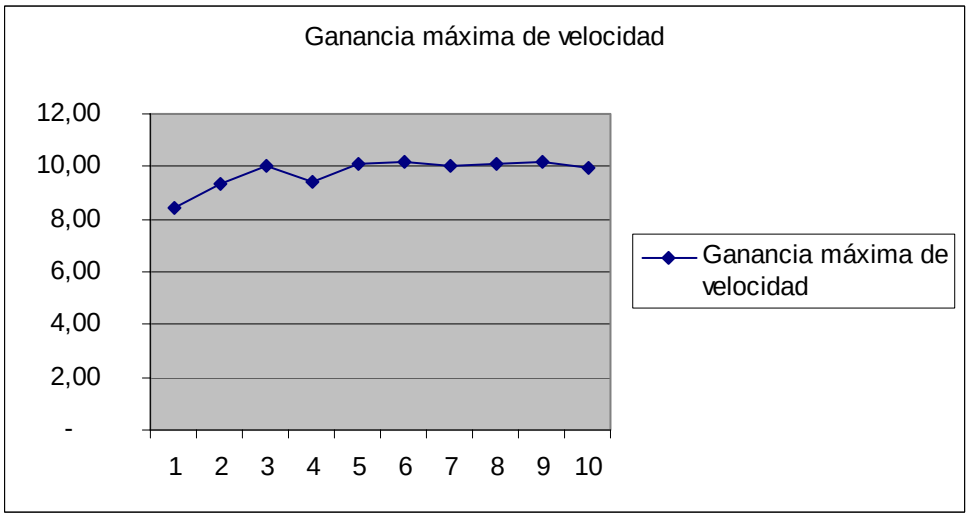
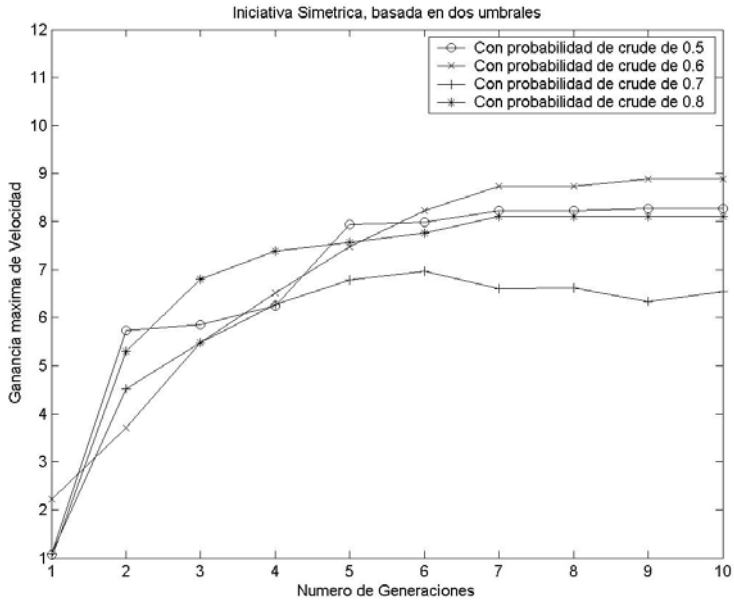


Figura 4.8: Ganancia de velocidad máxima con una población de treinta individuos

Probabilidad del cruce. Para determinar el valor de la probabilidad de cruce también se han hecho distintas pruebas con valores diferentes de dicho parámetro en distintos procedimientos de distribución de carga y con distintos programas paralelos de prueba aplicados a entradas de diferente complejidad.

En la Figura 4.9 se muestra un ejemplo característico de los resultados obtenidos. En este caso se ha considerado un procedimiento de equilibrio dinámico distribuido con política de transferencia de iniciativa simétrica y política de localización de dos umbrales. El procedimiento se ha utilizado para realizar una multiplicación paralela de matrices 1000x1000 en doce procesadores. Como se puede observar los mejores resultados de convergencia se producen para una probabilidad de cruce de 0.6, para probabilidades menores (0.5) y mayores (0.7 y 0.8) se obtienen resultados peores en la ganancia de velocidad del procedimiento. Como se puede ver, los valores para 0.5 y 0.7 son casi iguales.



Probabilidad de cruce	LBF	GRN	TSD1	TSD2	S Máxima	Tiempo total del algoritmo (s)
0,5	13	1000	11	83	8,27	857,35
0,6	13	1000	12	85	8,89	578,17
0,7	19	1000	26	95	6,96	873,15
0,8	7	1000	51	133	8,11	830,59

Figura 4.9: Convergencia del algoritmo genético para distintas probabilidades de cruce

Teniendo en cuenta los resultados obtenidos, en la mayoría del trabajo experimental realizado se utilizará una probabilidad de cruce de 0.6. En la Figura 4.9 también se muestran los valores a los que convergen los parámetros para cada probabilidad de cruce. Como se puede ver, pueden obtenerse valores diferentes para diferentes probabilidades de cruce. En la siguiente sección se considerarán estos aspectos.

4.3.2.2. Comparación de procedimientos y búsqueda de parámetros óptimos

A continuación se muestran los resultados obtenidos al explorar el espacio de diseño de los procedimientos distribuidos de equilibrado dinámico de carga. Ese espacio queda definido a partir de la taxonomía que presentamos en el Capítulo 3 por las alternativas de las políticas de información, transferencia, localización, distribución, y selección, y por sus parámetros correspondientes. Para organizar la exploración de ese espacio, se analizarán por separado los procedimientos de distribución de carga correspondientes a las tres alternativas de la política de transferencia: iniciada por el emisor, iniciada por el receptor, y simétrica. Además, se considerarán los resultados obtenidos para cada uno de los cuatro tipos de problemas de prueba considerados. En cada uno de los casos analizaremos la ganancia de velocidad y la convergencia de los parámetros en cada uno de los procedimientos alternativos para cada política de transferencia.

Tabla 4.2. Convergencia de los parámetros para las seis alternativas con política de transferencia iniciada por el emisor y el *benckmark* multiplicación de matrices (100x1000)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S máxima	Desv.	Util. Máxima
Umbral (*Preemption*)	1	1	144	3	1	12	1	6,4	0,11	0,52
Umbral (*No-Preemption*)	1	1	144	2	2	12	1	5,23	0,01	0,46
Aleatoria (*Preemption*)	1	2	144	1	2	13	3	5,47	0,08	0,47
Aleatoria (*No-Preemption*)	1	1	144	1	2	12	2	5,56	0,02	0,48
Mínimo/máximo (*Preemption*)	1	2	144	2	1	12	1	5,6	0,04	0,49
Mínimo/máximo (*No-Preemption*)	1	2	144	1	2	13	2	6,01	0,04	0,48

Multiplicación de matrices. La Tabla 4.2 muestra datos de convergencia de los parámetros LBF, TSD1, TSD2, considerando un valor fijo de granularidad (GRN=P^2=12^2=144) para las alternativas de los procedimientos de distribución con política de transferencia iniciada por el emisor. Por otra parte también se proporcionan los valores de la ganancia de velocidad y la utilidad media obtenidas al final de la evolución del algoritmo genético. Los resultados que se proporcionan corresponden a los valores medios de tres ejecuciones del programa paralelo con el procedimiento de distribución de carga correspondiente. Junto con ellos se proporciona la correspondiente desviación típica. La Figura 4.10 muestra la evolución de la ganancia de velocidad para 10 generaciones del algoritmo genético y la Figura 4.11 la evolución de la utilización media de los procesadores. Cada uno de los puntos de las gráficas corresponde al valor medio de tres ejecuciones del algoritmo con la distribución de carga correspondiente, la desviación típica para cada punto es muy pequeña para que pueda ser representada en las figuras.

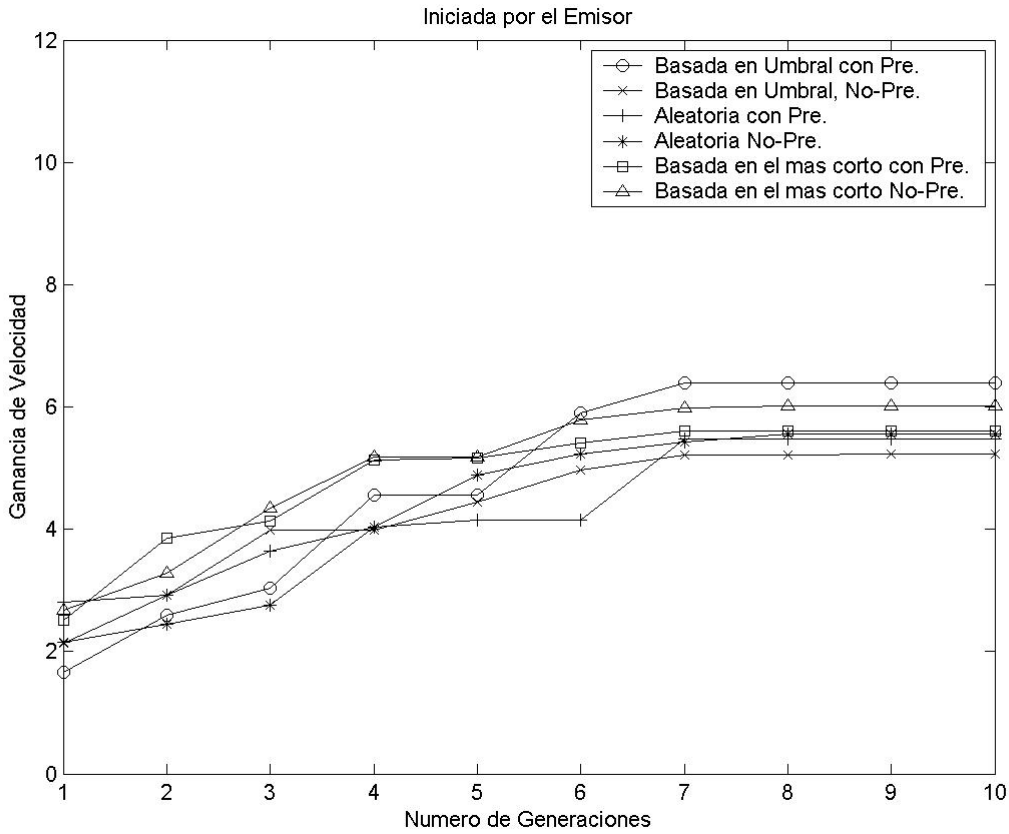


Figura 4.10 : Ganancia de velocidad para las seis alternativas con política de transferencia iniciada por el emisor y el *benckmark* multiplicación de matrices (1000x1000)

Como conclusiones de los resultados proporcionados en la tabla y en las figuras se pueden extraer las siguientes. En primer lugar, parece bastante clara la convergencia a ciertos valores en los parámetros. Por ejemplo, LBF parece tender a 1 en todos los procedimientos, TSD1 converge a 1, 2, o 3 según los procedimientos, y TSD2 a valores entre 12 y 13 también según los procedimientos. Las desviaciones típicas para estos valores son suficientemente pequeñas en comparación con los rangos entre los que pueden variar los parámetros. En cuanto a los valores de la ganancia y la utilización media, los valores obtenidos para todos los procedimientos son relativamente bajos, y se observan tendencias parecidas en ambas magnitudes para todos los procedimientos. Quizá los valores de las utilizaciones medias son más parecidos en todos los procesadores. Parece que los procedimientos mejores tanto en ganancia de velocidad como en utilización son el procedimiento basado en umbral con corte forzado y las basadas en la mínima/máxima carga con o sin corte forzado. Las peores alternativas, como era de esperar, corresponden a los procedimientos aleatorios.

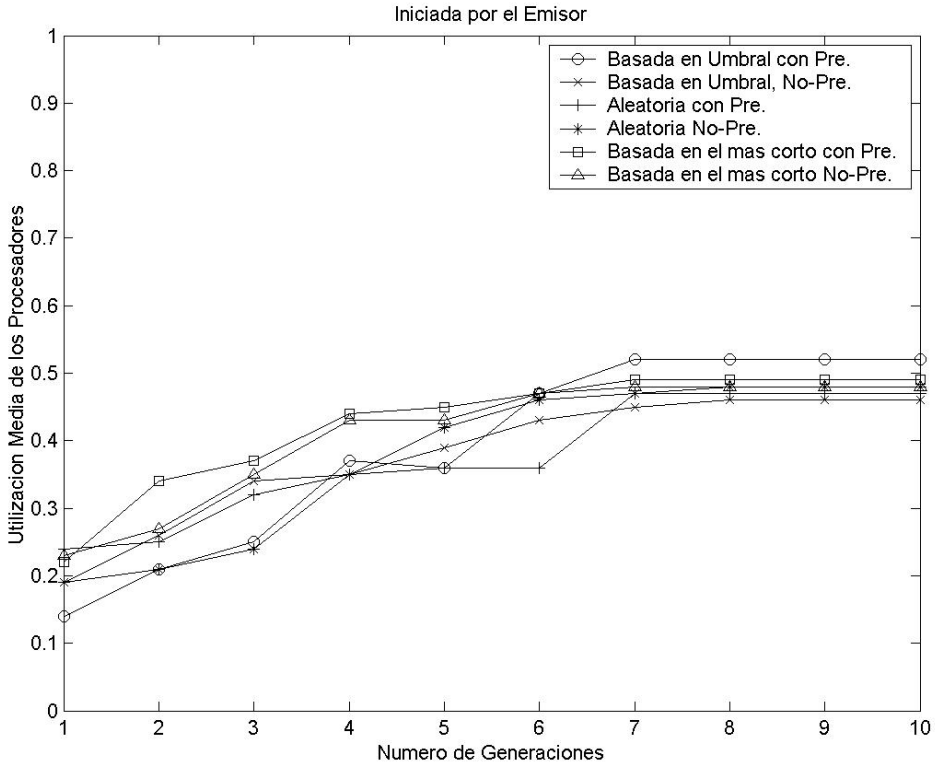


Figura 4.11. Utilización media de los procesadores para las seis alternativas con política de transferencia iniciada por el emisor y multiplicación de matrices (1000x1000)

Tabla 4.3 Convergencia de los parámetros para las alternativas con política de transferencia iniciada por el receptor y el *benckmark* multiplicación de matrices (1000x1000)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S máxima	Desv.	Util. Máxima
Umbral (*Preemption*)	3	2	144	3	1	-	-	6.27	0,03	0.55
Umbral (*No-Preemption*).	1	1	144	2	1	-	-	6.72	0,21	0,57

La Tabla 4.3 y las Figuras 4.12 y 4.13 proporcionan los mismos datos que la Tabla 4.2 y las Figuras 4.10 y 4.11, pero ahora para los procedimientos con política de transferencia iniciada por el receptor. Se puede observar que LBF y TSD1 convergen, aunque a valores ligeramente diferentes (si se tienen en cuenta sus desviaciones puede haber un cierto solapamiento). Los valores de las ganancias y las utilizaciones medias son también bajos y del mismo orden que los que se observan para los procedimientos iniciados pro el emisor. El procedimiento basado en un umbral sin corte forzado parece ser el que proporciona una mejor ganancia de velocidad y una mejor utilización, aunque sus valores son relativamente próximos.

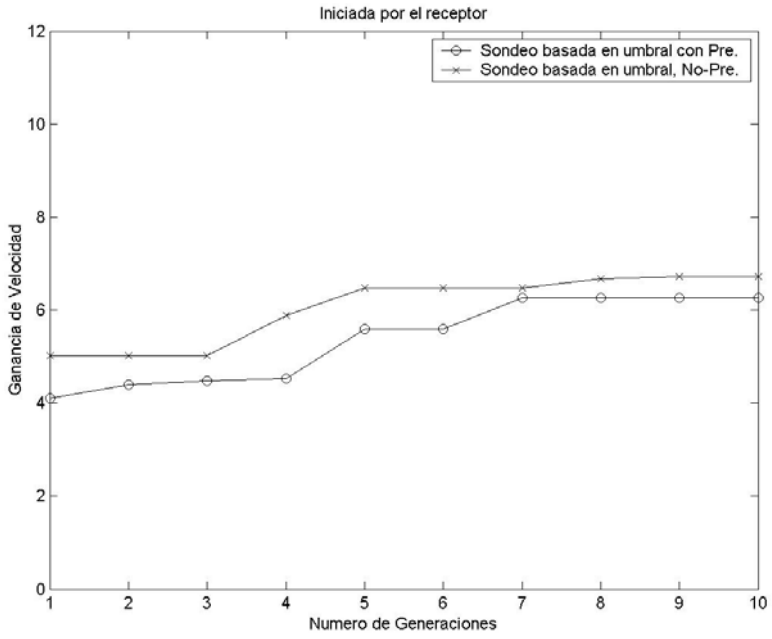


Figura 4.12. Ganancia de velocidad para las alternativas con política de transferencia iniciada por el receptor y el *benckmark* multiplicación de matrices (1000x1000)

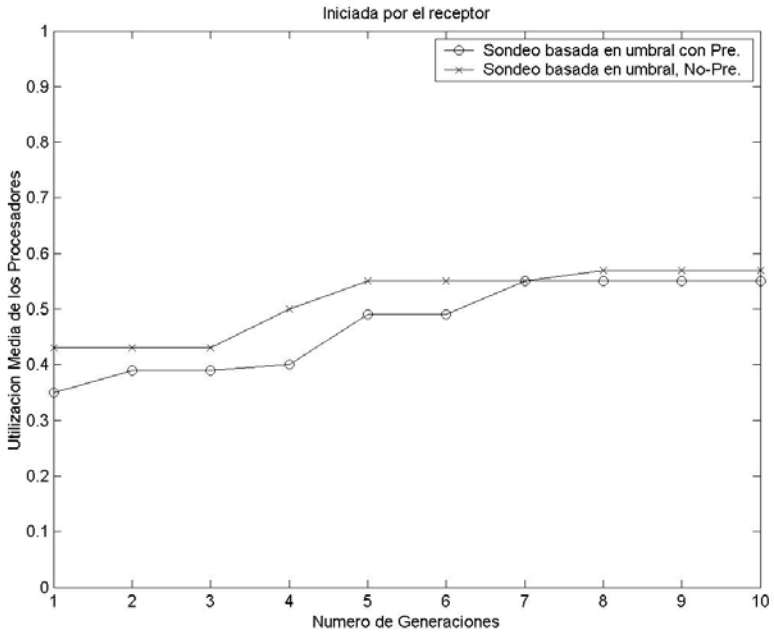


Figura 4.13. Utilización media de los procesadores para las alternativas con política de transferencia iniciada por el receptor y multiplicación de matrices (1000x1000)

Tabla 4.4. Convergencia de los parámetros para las seis alternativas con política de transferencia simétrica y el *benckmark* multiplicación de matrices (1000x1000)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S máxima	Desv.	Util. máxima
Umbral (*Preemption*)	2	1	144	1	2	11	2	9.10	0,01	0,74
Umbral (*No-preemption*)	2	2	144	3	1	12	1	8.28	0,04	0,71
Dos Umbrales (*Preemption*)	2	1	144	2	2	11	2	8.46	0,02	0,73
Dos Umbrales (*No-Preemption*)	2	2	144	4	2	14	3	8.68	0,01	0,75
Mínimo/máximo (*No preemption*)	3	2	144	-	-	-	-	8,15	0,01	0,72
Mínimo/máximo (*No preemption*)	2	2	144	-	-	-	-	7,94	0,01	0,70

La Tabla 4.4 y las Figuras 4.14 y 4.15 proporcionan, para los procedimientos con política de transferencia simétrica y para la multiplicación de matrices, los datos de convergencia de los parámetros, y la evolución del algoritmo genético en cuanto a ganancia de velocidad y utilización media. En relación con la convergencia de los parámetros (se recuerda, para GRN=P^2=144), se observa que, según el procedimiento, LBF converge a

valores entre 2 y 3, TSD1 a valores entre 1 y 4, y TSD2 a valores entre 11 y 14 (aunque teniendo en cuenta la desviación típica esos valores se solaparían en todos los parámetros).

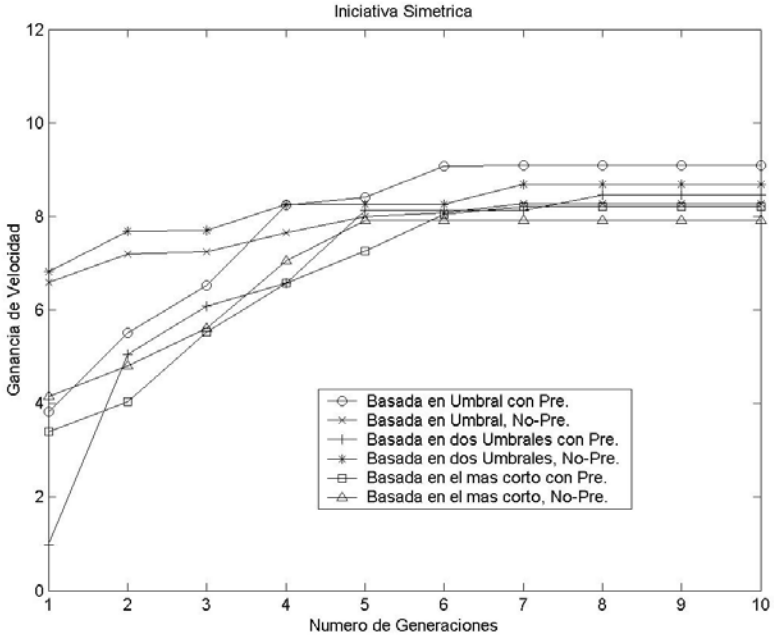


Figura 4.14. Ganancia de velocidad para las seis alternativas con política de transferencia simétrica y el *benckmark* multiplicación de matrices (1000x1000)

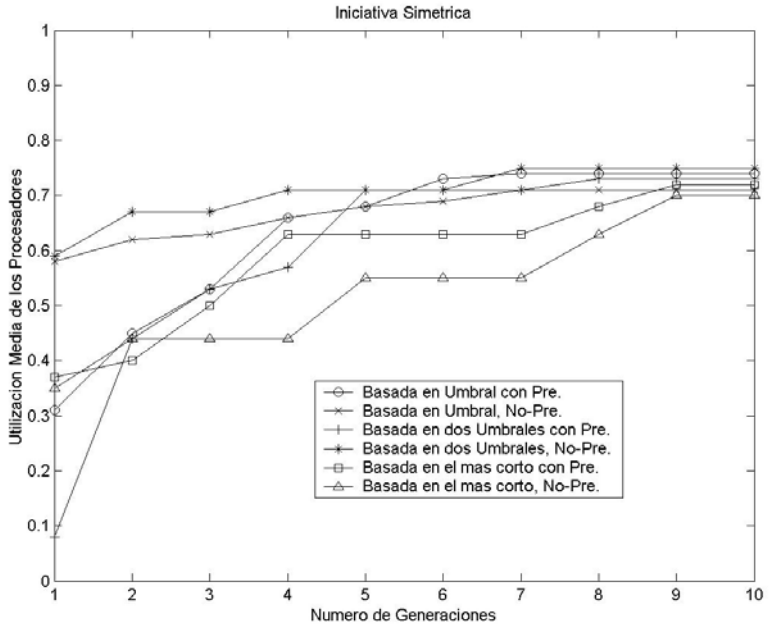


Figura 4.15. Utilización media para las seis alternativas con política de transferencia simétrica y el *benckmark* multiplicación de matrices (1000x1000)

En cuanto a la ganancia de velocidad y a la utilización media, se observa que los valores obtenidos para ambas en los procedimientos con transferencia simétrica superan a los de los otros procedimientos en el caso de la multiplicación paralela de matrices (se obtienen ganancias superiores a 9 y utilizaciones medias de más de un 70%). Además, se puede ver que la evolución de las ganancias y las utilizaciones medias tiene un comportamiento muy parecido en las alternativas para los procedimientos con transferencia simétrica. No obstante, se sigue manteniendo la alternativa correspondiente a la política de localización basada en un umbral con corte forzado como mejor alternativa.

También se puede concluir que los parámetros LBF, TSD1, y TSD2 tienden a valores situados en rangos parecidos en todos los procedimientos, para el caso de la multiplicación paralela de matrices. Así, LBF está entre 1 y 3; TSD1 entre 1 y 4; y TSD2 entre 11 y 14, por supuesto, considerando GRN=P^2=12^2=144.

A continuación se proporcionan los datos correspondientes a los otros tres programas paralelos que se han descrito en la Sección 4.3.1: el cálculo de pi con el método de Monte Carlo, el cálculo de la integral por el método del paralelogramo, y un procedimiento de ramificación y poda (*branch-and-bound*) para el problema del viajante de comercio. Para cada uno de estos casos los procedimientos se agrupan teniendo en cuenta las tres políticas de transferencia: iniciada por el emisor, iniciada por el receptor, y simétrica. Como hasta ahora, para cada caso se proporciona una tabla con información de convergencia de los parámetros y dos figuras que muestran la evolución en cuanto a ganancia y utilización media, respectivamente, para las alternativas dentro de una misma política de transferencia. De esta forma se pueden comparar esas alternativas. Para no ser excesivamente prolijo, no se explicarán los datos tabla a tabla o figura a figura, sino que se indicarán las principales conclusiones que se puedan extraer para cada aplicación paralela considerada.

Cálculo del número pi. Las Tablas 4.5, 4.6, y 4.7 y las Figuras 4.16 – 4.21 proporcionan datos relativos a los procedimientos de distribución de carga para las tres políticas de transferencia (iniciada por el emisor, iniciada por el receptor, y simétrica) utilizando como programa de prueba el cálculo paralelo de pi mediante el Método de Monte Carlo.

Tabla 4.5. Convergencia de los parámetros para las seis alternativas con política de transferencia iniciada por el emisor y el *benckmark* de cálculo del número pi (10^8 iteraciones)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima
Umbral (*Preemption*)	2	1	144	2	2	10	2	6,02	0,07	0,39
Umbral (*No-Preemption*).	1	1	144	3	2	11	2	5,93	0,06	0,35
Aleatoria (*Preemption*).	1	2	144	2	2	12	2	6,28	0,17	0,36
Aleatoria (*No-Preemption*)	1	2	144	6	2	13	2	6,28	0,01	0,36
Mínimo/máximo (*Preemption*)	3	1	144	2	1	13	1	5,28	0,08	0,29
Mínimo/máximo (*No-Preemption*)	1	2	144	2	1	13	2	6,15	0,05	0,38

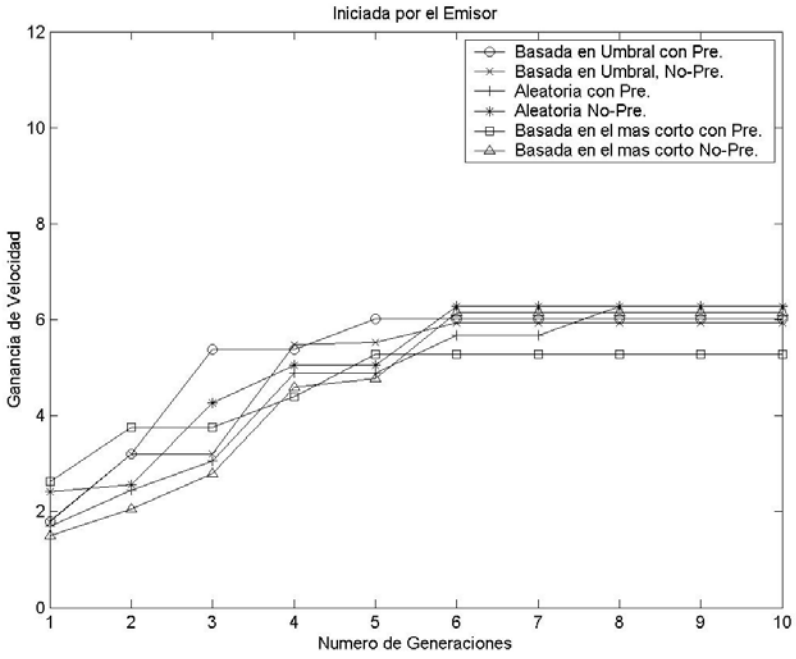


Figura 4.16. Ganancia de velocidad para las seis alternativas con política de transferencia iniciada por el emisor y el *benckmark* de cálculo de pi (10^8 iteraciones)

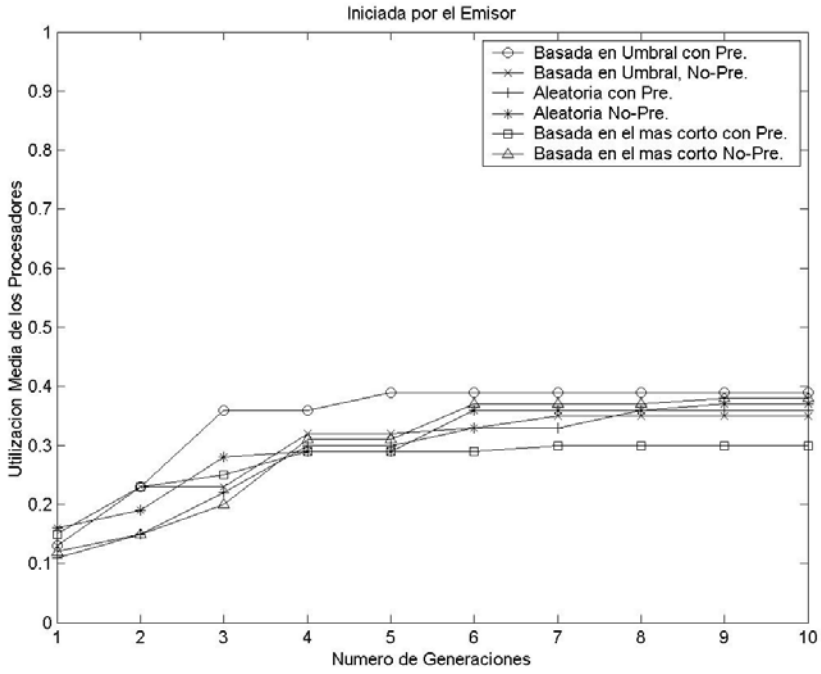


Figura 4.17. Utilización media para las seis alternativas con política de transferencia iniciada por el emisor y el *benckmark* de cálculo de pi (10^8 iteraciones)

Tabla 4.6. Convergencia de los parámetros para las dos alternativas con política de transferencia iniciada por el receptor y el *benckmark* de cálculo del número pi (10^8 iteraciones)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima
Umbral (Preemption)	2	1	144	6	3	-	-	7,91	0,01	0,47
Umbral (No-Preemption)	2	1	144	2	1	-	-	8,21	0,01	0,48

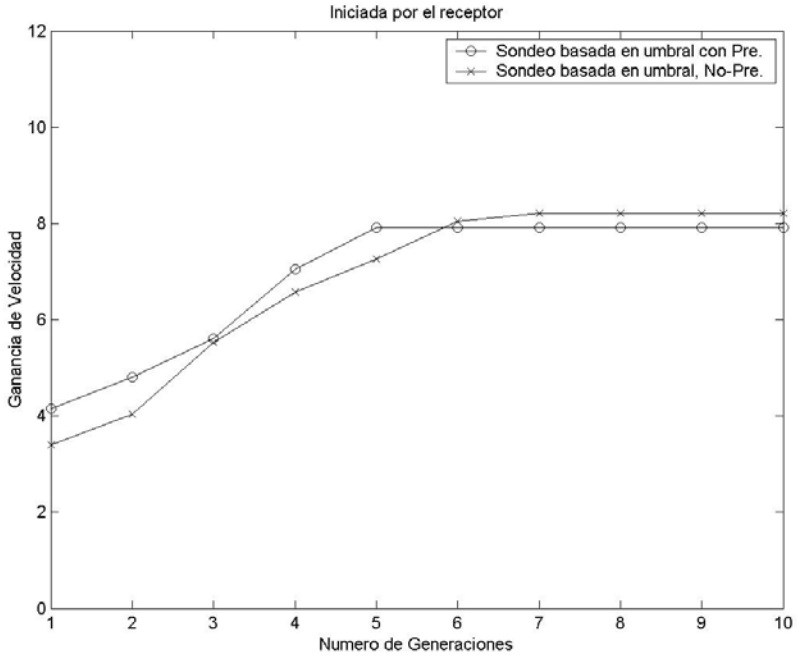


Figura 4.18: Ganancia de velocidad para las dos alternativas con política de transferencia iniciada por el receptor y el *benckmark* de cálculo de pi (10^8 iteraciones)

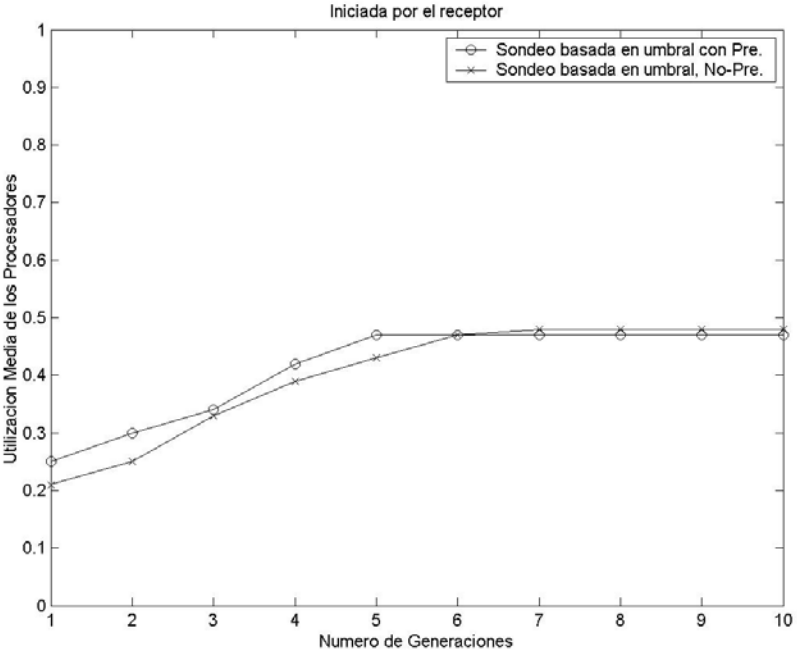


Figura 4.19: Utilización media para las dos alternativas con política de transferencia iniciada por el receptor y el *benckmark* de cálculo de pi (10^8 iteraciones)

Tabla 4.7. Convergencia de los parámetros para las seis alternativas con política de transferencia simétrica y el *benckmark* de cálculo del número pi (10^8 iteraciones)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima
Umbral *(Preemption)*	2	1	144	1	2	13	1	10,35	0,03	0,61
Umbral *(No-Preemption)*	2	2	144	3	2	11	2	9,93	0,01	0,59
Dos Umbrales *(Preemption)*	2	1	144	1	1	13	1	9,54	0,01	0,56
Dos Umbrales *(No Preemption)*	2	1	144	2	1	13	2	10,34	0,03	0,6
Mínimo/máximo *(Preemption)*	1	1	144	-	-	-	-	9,20	0,12	0,58
Mínimo/máximo *(No Preemption)*	2	1	144	-	-	-	-	9,10	0,06	0,57

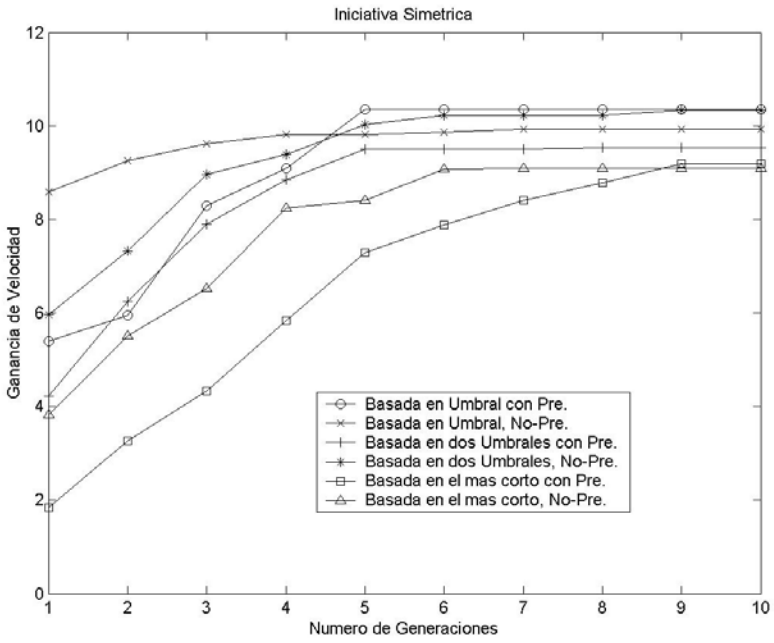


Figura 4.20. Ganancia de velocidad para las seis alternativas con política de transferencia simétrica y el *benckmark* de cálculo de pi (10^8 iteraciones)

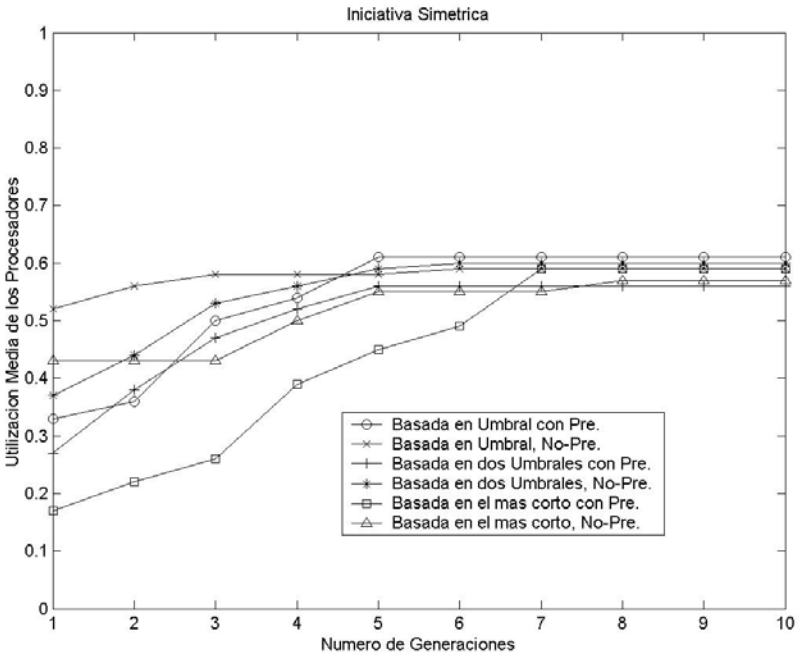


Figura 4.21. Utilización media para las seis alternativas con política de transferencia simétrica y el *benckmark* de cálculo de pi (10^8 iteraciones)

Las conclusiones respecto a los resultados experimentales obtenidos para el cálculo paralelo de *pi* son muy parecidas a las que se han extraído para la multiplicación de matrices. En primer lugar tenemos que los procedimientos con política de transferencia simétrica permiten alcanzar mejores valores para la ganancia de velocidad y la utilización media del programa paralelo. Según la alternativa, para estos procedimientos con transferencia simétrica, y utilizando 12 procesadores, se obtienen ganancias de entre 9.1 y 10.3, y utilizaciones medias entre 0.56 y 0.61. Después estarían los procedimientos con transferencia iniciada por el receptor, y finalmente los iniciados por el emisor. En cuanto a la convergencia en los parámetros, utilizando también GRN=P^2=144 se observa que LBF tiende a valores entre 1 y 2, TSD1 a valores entre 1 y 3, y TSD2 a valores entre 10 y 13, dependiendo de las alternativas que hay para cada política de transferencia.

En cuanto a las diferencias entre las políticas de localización y selección, en el caso de los procedimientos con política de transferencia iniciada por el emisor todos, salvo la localización basada en el máximo/mínimo nivel de carga con corte forzado, todas proporcionan resultados de ganancia de velocidad y utilización media muy similares. Las dos

alternativas consideradas para la política de transferencia iniciada por el receptor son bastante similares, aunque la basada en un umbral sin corte forzado es algo mejor. Finalmente, en el caso de la política de transferencia simétrica, todas las alternativas tienen una tendencia similar en la convergencia. Las que mejores prestaciones ofrecen en cuanto a la ganancia son las basadas en uno o dos umbrales, y dentro de éstas la basada en un umbral con corte forzoso y la basada en dos umbrales sin corte forzoso son las que proporcionan un nivel de prestaciones algo mejor.

Cálculo de una integral definida. A continuación se analizan los resultados obtenidos para el programa paralelo de prueba correspondiente al cálculo de una integral definida mediante el método del trapecio. Esos resultados, para cada uno de los procedimientos de distribución de carga según las tres políticas de transferencia (iniciada por el emisor, iniciada por el receptor, y simétrica), se muestran en las Tablas 4.8, 4.9, y 4.10 y las Figuras 4.22 – 4.27.

Tabla 4.8 Convergencia de los parámetros para las seis alternativas con política de transferencia iniciada por el emisor y el cálculo de la integral (5×10^8 iteraciones)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima
Umbral *(Preemption)*	1	1	144	2	1	14	2	6,19	0,01	0,53
Umbral *(No-Preemption).*	1	1	144	1	1	12	1	6,39	0,05	0,54
Aleatoria *(Preemption).*	1	2	144	1	1	13	1	7,16	0,1	0,61
Aleatoria *(No-Preemption)*	1	1	144	1	2	12	1	5,97	0,02	0,51
Mínimo/máximo *(Preemption)*	1	2	144	3	3	11	1	7,04	0,01	0,59
Mínimo/máximo *(No-Preemption)*	1	1	144	2	2	12	1	6,5	0,09	0,56

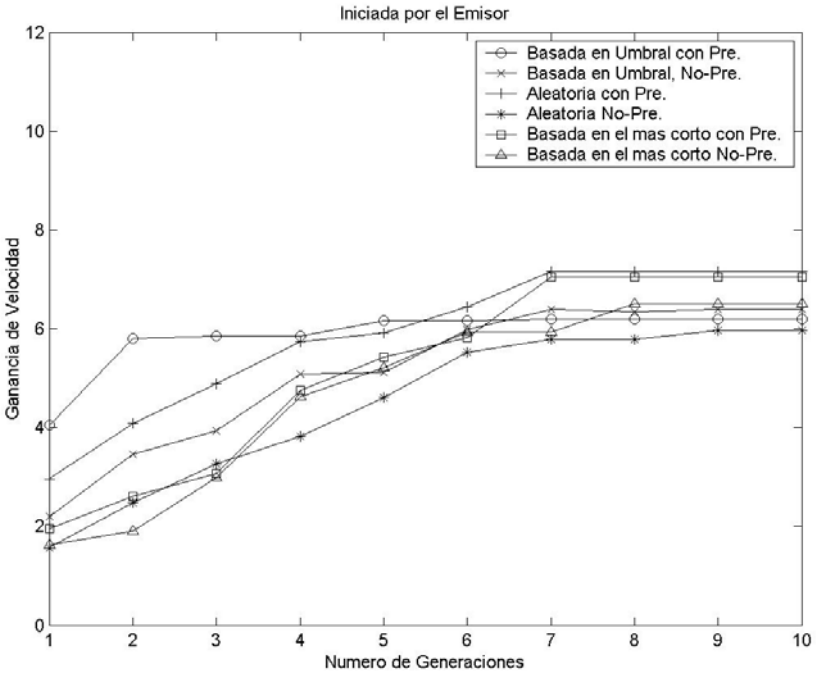


Figura 4.22. Ganancia de velocidad para las seis alternativas con política de transferencia iniciada por el emisor y el cálculo de la integral (5×10^8 iteraciones)

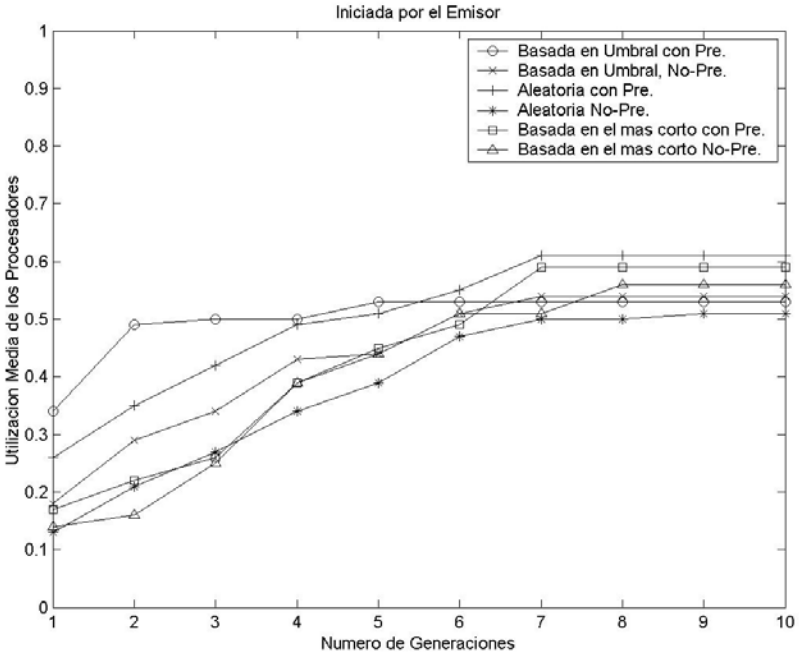


Figura 4.23. Utilización media para las seis alternativas con política de transferencia iniciada por el emisor y el cálculo de la integral (5×10^8 iteraciones)

Tabla 4.9 Convergencia de los parámetros para las dos alternativas con política de transferencia iniciada por el receptor y el cálculo de la integral (5×10^8 iteraciones)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima
Umbral *(Preemption)*	2	2	144	1	1	-	-	7,45	0,04	0,63
Umbral *(No-Preemption)*	1	2	144	2	1	-	-	7,87	0,1	0,66

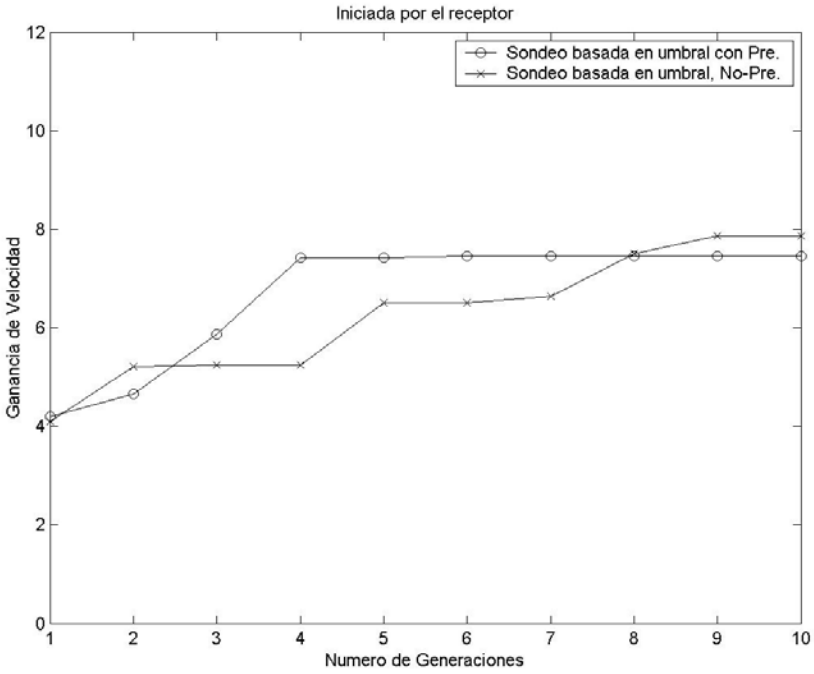


Figura 4.24: Ganancia de velocidad para las dos alternativas con política de transferencia iniciada por el receptor y el cálculo de la integral (5×10^8 iteraciones)

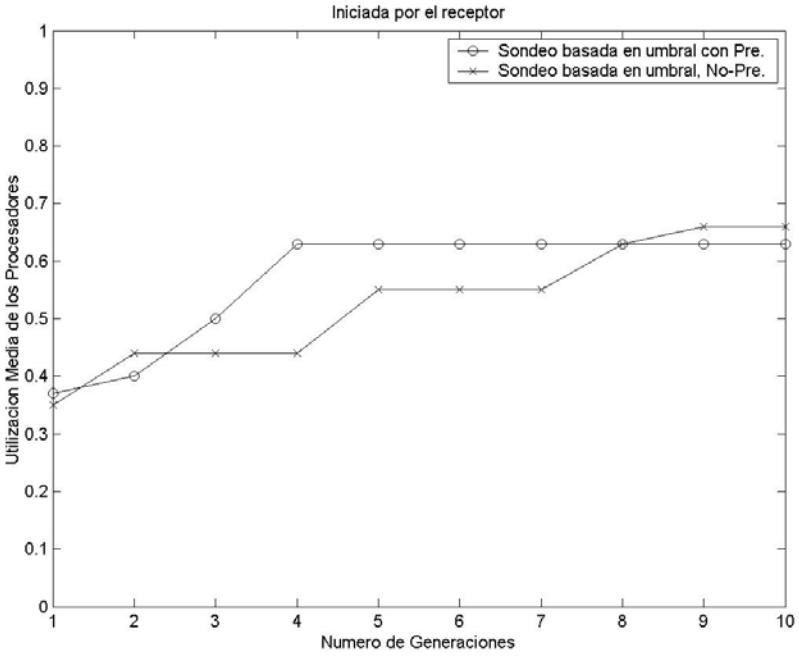


Figura 4.25: Utilización media para las dos alternativas con política de transferencia iniciada por el receptor y el cálculo de la integral (5×10^8 iteraciones)

Tabla 4.10 Convergencia de los parámetros para las seis alternativas con política de transferencia simétrica y el cálculo de la integral (5×10^8 iteraciones)

Procedimiento	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima
Umbral *(Preemption)*	2	2	144	1	2	12	2	9,65	0,07	0,82
Umbral *(No-Preemption)*	2	2	144	3	3	11	1	9,81	0,02	0,83
Dos Umbrales *(Preemption)*	2	1	144	2	2	12	1	10,05	0,02	0,84
Dos Umbrales *(No Preemption)*	2	2	144	2	1	13	1	9,2	0,02	0,79
Mínimo/máximo *(Preemption)*	2	2	144	-	-	-	-	5,7	0,03	0,51
Mínimo/máximo *(No Preemption)*	1	2	144	-	-	-	-	5,7	0,01	0,52

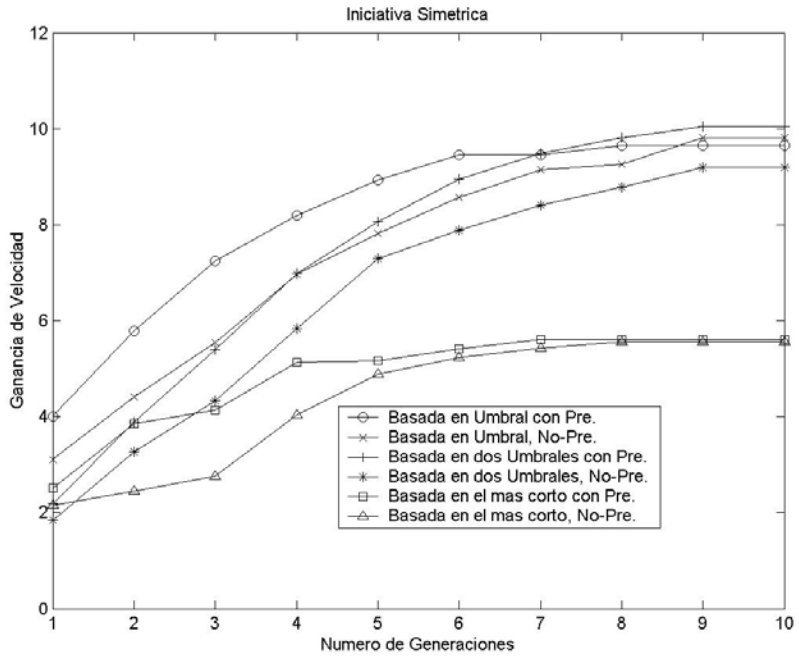


Figura 4.26: Ganancia de velocidad para las seis alternativas con política de transferencia simétrica y el cálculo de la integral (5×10^8 iteraciones)

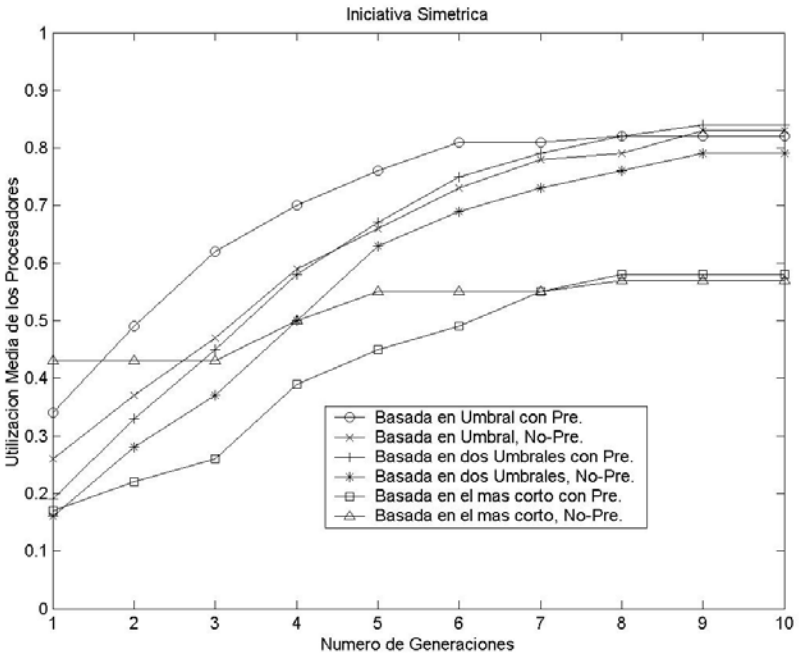


Figura 4.27: Utilización media para las seis alternativas con política de transferencia simétrica y el cálculo de la integral (5×10^8 iteraciones)

Las conclusiones respecto a los resultados experimentales obtenidos para el cálculo de la integral son similares a las que se han podido establecer para los programas de prueba que se han mostrado antes. Aquí también ocurre que los procedimientos con política de transferencia simétrica son los que alcanzan los mejores valores de ganancia de velocidad y la utilización media de procesadores por parte del programa paralelo. Así, para los procedimientos con transferencia simétrica se obtienen ganancias que llegan hasta 10.05 para 12 procesadores y utilizaciones medias de hasta 0.84. Después estarían los procedimientos con transferencia iniciada por el receptor, y finalmente los iniciados por el emisor, que proporciona en sus mejores alternativas valores de ganancia similares a los procedimientos con transferencia iniciada por el receptor. En cuanto a la convergencia en los parámetros, utilizando también GRN=P^2=144, se tiene que LBF tiende a valores entre 1 y 2, TSD1 a valores entre 1 y 3, y TSD2 a valores entre 11 y 14, dependiendo de las alternativas que hay para cada política de transferencia.

En cuanto a las diferencias entre las políticas de localización y selección, en el caso de los procedimientos con política de transferencia iniciada por el emisor todos tienen un comportamiento similar en cuanto a ganancia de velocidad y utilización media. Las diferencias se pueden observar en las Figuras 4.22 y 4.23. Las dos alternativas consideradas para la política de transferencia iniciada por el receptor son bastante similares, aunque la basada en un umbral sin corte forzado es algo mejor, como ya ocurría con el programa de prueba del cálculo paralelo de pi. Para terminar con este programa de prueba, en el caso de la política de transferencia simétrica, todas las alternativas tienen una tendencia similar en cuanto a la utilización media y a la ganancia de velocidad excepto dos para las que se observan valores más bajos tanto en la ganancia de velocidad (valores de 5.7 frente a valores superiores a 9.0) como en la utilización (valores de 0.51 frente a valores por encima de 0.79). Estas dos alternativas que tienen peores prestaciones son las dos basadas en el máximo/mínimo nivel de carga, independiente del algoritmo de selección utilizado.

Algoritmo de ramificación y poda (branch-and-bound). Para concluir esta sección se muestran los resultados de un algoritmo paralelo de ramificación y poda en la resolución del problema del viajante de comercio. Este problema tiene unas características distintas de los otros tres problemas considerados. Por una parte, resulta más complicado asignar un coste

computacional a las tareas entre las que se distribuye el trabajo. Precisamente, el volumen de cómputo del trabajo no se conoce de antemano, y depende del propio proceso de búsqueda implícita que implementa el algoritmo en el problema sobre el que se aplica. Este carácter hace imprescindible utilizar un procedimiento de distribución de carga dinámico en este tipo de problemas.

El algoritmo de ramificación y poda se ha aplicado al problema del viajante de comercio, que es uno de los problemas de optimización más utilizados para evaluar las prestaciones de los algoritmos de optimización. Se han considerado dos tamaños de problema correspondientes a 8 y 9 ciudades, respectivamente.

Los resultados para el caso de 8 ciudades se muestran en las Tablas 4.11 – 4.13 y en las Figuras 4.28 – 4.30. Las tablas proporcionan, para los procedimientos alternativos de cada una de las políticas de transferencia (iniciada por el emisor, iniciada por el receptor, y simétrica), los valores de los parámetros a los que se converge el algoritmo genético y los valores de la idoneidad y de la utilización media obtenidos. Como valor de idoneidad se representa la inversa del tiempo de ejecución paralela, con lo que evitamos el tiempo de ejecución secuencial del algoritmo, que puede llegar a ser extremadamente elevado. Las figuras representan la evolución de la idoneidad a través de las sucesivas generaciones del algoritmo genético en los procedimientos alternativos de cada política de transferencia.

Tabla 4.11 Convergencia de los parámetros para las seis alternativas con política de transferencia iniciada por el emisor y el algoritmo de ramificación y poda (tamaño 8!)

Procedimiento	LBF	Desv.	GRN	Desv.	TSD1	Desv.	TSD2	Desv.	1/(Tiempo Paralelo)	Desv.	Util. Máxima
Umbral *(Preemption)*	121	10	2	1	90	8	290	10	0.76	0,04	0.04
Umbral *(No-Preemption)*	125	15	1	1	103	12	297	15	0.68	0,03	0.03
Dos Umbrales *(Preemption)*	105	7	1	1	94	6	273	20	0.85	0,01	0.03
Dos Umbrales *(No Preemption)*	120	10	1	1	117	20	290	8	0.78	0,03	0.03
Mínimo/máximo *(Preemption)*	115	8	2	1	105	14	290	9	0.80	0,02	0.04
Mínimo/máximo *(No Preemption)*	117	9	2	1	98	7	328	30	0.83	0,01	0.03

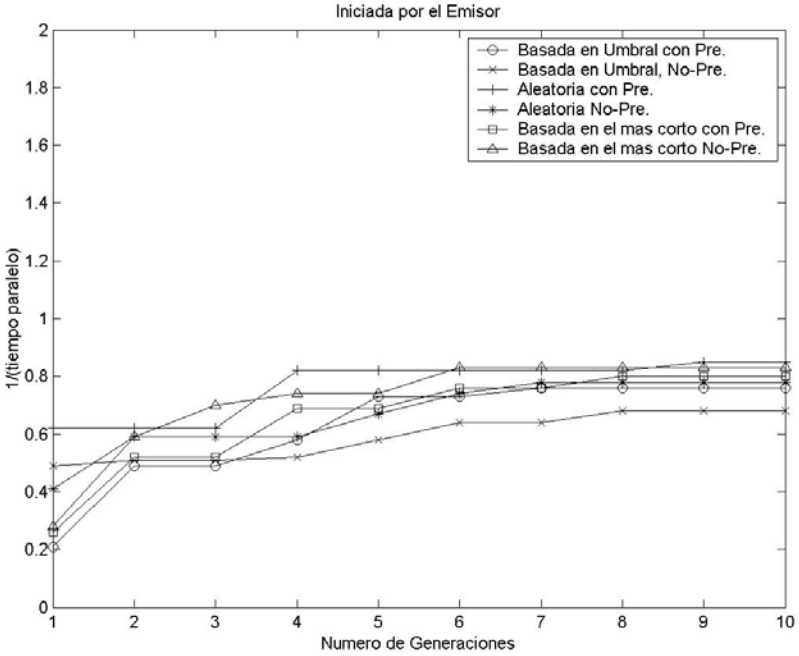


Figura 4.28 : Idoneidad (1/Tiempo paralelo) para las seis alternativas con política de transferencia iniciada por el emisor y algoritmo de ramificación y poda (tamaño 8!)

Tabla 4.12 Convergencia de los parámetros para las dos alternativas con política de transferencia iniciada por el receptor y algoritmo de ramificación y poda (tamaño 8!)

Procedimiento	LBF	Desv.	GRN	Desv.	TSD1	Desv.	TSD2	Desv.	1/(Tiempo Paralelo)	Desv.	Util. Máxima
Umbral *(Preemption)*	85	7	1	1	97	7	-	-	0.57	0,03	0.19
Umbral *(No-Preemption)*	75	8	1	1	108	11	-	-	0.55	0,01	0.05

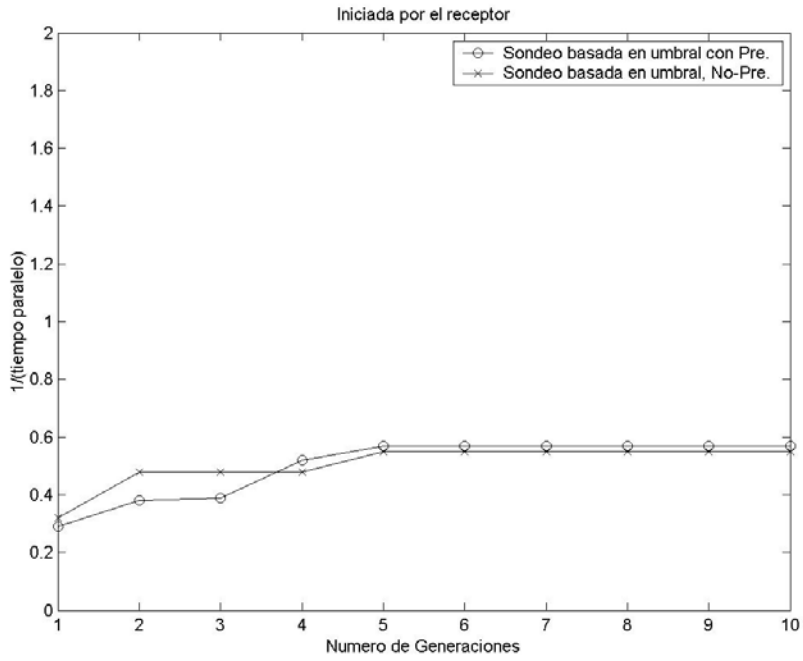


Figura 4.29: Idoneidad (1/Tiempo paralelo) para las dos alternativas con política de transferencia iniciada por el receptor y el algoritmo de ramificación y poda (tamaño 8!)

Tabla 4.13 Convergencia de los parámetros para las seis alternativas con política de transferencia simétrica y el algoritmo de ramificación y poda (tamaño 8!)

Procedimiento	LBF	Desv.	GRN	Desv.	TSD1	Desv.	TSD2	Desv.	1/(Tiempo Paralelo)	Desv.	Util. Máxima
Umbral *(Preemption)*	107	12	1	1	101	7	279	18	1.87	0,03	0.09
Umbral *(No-Preemption)*	103	8	2	1	106	8	279	16	1.88	0,02	0.11
Dos Umbrales *(Preemption)*	111	11	2	1	109	12	280	12	1.58	0,02	0.17
Dos Umbrales *(No Preemption)*	116	14	2	1	94	6	297	8	1.80	0,03	0.13
Mínimo/máximo *(Preemption)*	105	7	2	1	-	-	-	-	1.00	0,01	0.05
Mínimo/máximo *(No Preemption)*	119	14	5	3	-	-	-	-	1.24	0,03	0.03

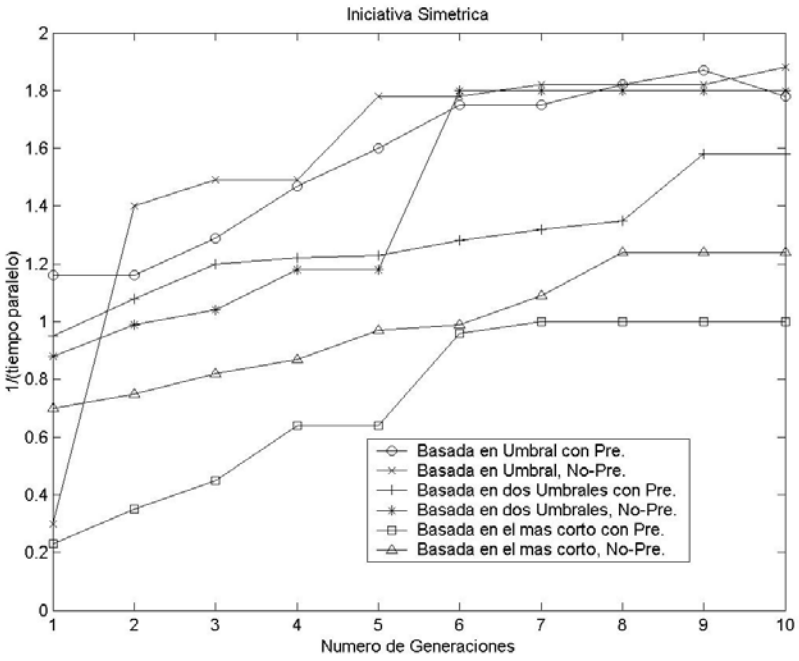


Figura 4.30: Idoneidad (1/Tiempo paralelo) para las seis alternativas con política de transferencia simétrica y el algoritmo de ramificación y poda (tamaño 8!)

Los resultados obtenidos para el algoritmo de ramificación y poda muestran que los procedimientos con política de transferencia simétrica son los que proporcionan unos mejores valores para la ganancia de velocidad. En este caso los procedimientos con política de transferencia iniciada por el emisor son mejores que los de transferencia iniciada por el receptor, que se encontraban en el segundo lugar para los otros tres programas de prueba que hemos analizado antes. Una característica importante de todas las versiones del algoritmo de ramificación y poda con los distintos procedimientos de distribución de carga es la baja utilización que se observa. En este caso, las utilizaciones de los procedimientos simétricos son similares en algunos casos a las de los procedimientos iniciados por el receptor, y son los procedimientos con transferencia iniciada por el emisor los que tienen utilizaciones más bajas.

En cuanto a las distintas alternativas para cada política de transferencia tenemos que en el caso de los procedimientos con transferencia iniciada por el emisor (Figura 4.28) y por el receptor (Figura 4.29) todas las alternativas convergen hacia valores bastante próximos, indicando que puede considerarse que son casi equivalentes en cuanto a la ganancia de velocidad que permiten alcanzar. En cuanto a los procedimientos con transferencia simétrica,

la Figura 4.30 pone de manifiesto comportamientos relativamente diferentes para distintas políticas de localización y selección. En este caso se observa que los procedimientos basados en un umbral (con o sin corte forzado) y el procedimiento basado en dos umbrales sin corte forzado son los que proporcionan mejores prestaciones.

La convergencia a los parámetros GRN, LBF, TSD1, y TSD2 muestra mayores desviaciones que en los tres programas de prueba que se han analizado antes. Además existe una mayor variación entre los parámetros a los que se converge según los procedimientos. Por ejemplo GRN está entre 1 y 2, aunque en algunos casos es igual a 5. Los valores de LBF a los que se converge están entre 103 y 121 para los procedimientos con transferencia iniciada por el emisor y los simétricos, pero en el caso de la transferencia iniciada por el receptor son 85 y 75, respectivamente, para las dos alternativas consideradas. Los valores de TSD1 están entre 90 y 117, y los valores de TSD2 están entre 273 y 297, aunque en un caso llega a 328 (transferencia iniciada por el emisor con localización basada en el máximo/mínimo nivel de carga sin corte forzoso). El mejor procedimiento para el algoritmo de ramificación y poda aplicada al problema del viajante de comercio con 8 ciudades utiliza política de transferencia simétrica, política de localización de un umbral, y política de selección sin corte forzoso con GRN=2, LBF=103, y TSD1=106.

Para este programa de prueba también hemos analizado los resultados para un tamaño de problema diferente. Concretamente, se ha considerado el problema del viajante de comercio para 9 ciudades.

Tabla 4.14 Convergencia de los parámetros para las dos mejores alternativas con política de transferencia simétrica y el algoritmo de ramificación y poda (tamaño 9!)

Procedimiento	LBF	Desv.	GRN	Desv.	TSD1	Desv.	TSD2	Desv.	1000/(Tiempo Paralelo)	Desv.	Util. Máxima
Dos Umbrales *(Preemption)*	673	10	1	1	1197	43	1857	111	36,24	1,29	0.67
Dos Umbrales *(No Preemption)*	687	11	2	1	1287	55	2156	121	34,41	1,29	0.67

En la Tabla 4.14 se muestran los resultados para dos procedimientos con política de transferencia simétrica (que son los que ofrecen mejores prestaciones) basados en dos umbrales. Como se puede ver, GRN sigue convergiendo a valores iguales a 1 ó 2, pero el

resto de parámetros, al aumentar el tamaño del problema también aumentan. Así, aproximadamente, LBF está entre 670 y 690, TSD1 entre 1200 y 1300, y TSD2 entre 1800 y 2200. Un dato importante que muestra la Tabla 4.14 es el de la utilización media, que para este problema es bastante más elevada que en el problema del viajante de comercio para 8 ciudades. Como en el caso de 8 ciudades, para 9 ciudades el mejor procedimiento también utiliza una política de transferencia simétrica, pero una política de localización de dos umbrales, y una política de selección con corte forzoso con GRN=1, LBF=673, TSD1=1197, y TSD2=1857.

En la Figura 4.31 se muestra la evolución de la idoneidad para las generaciones sucesivas del algoritmo genético. Se puede comprobar que el proceso de optimización de parámetros que realiza el algoritmo genético permite mejorar considerablemente las prestaciones de los procedimientos de distribución de carga.

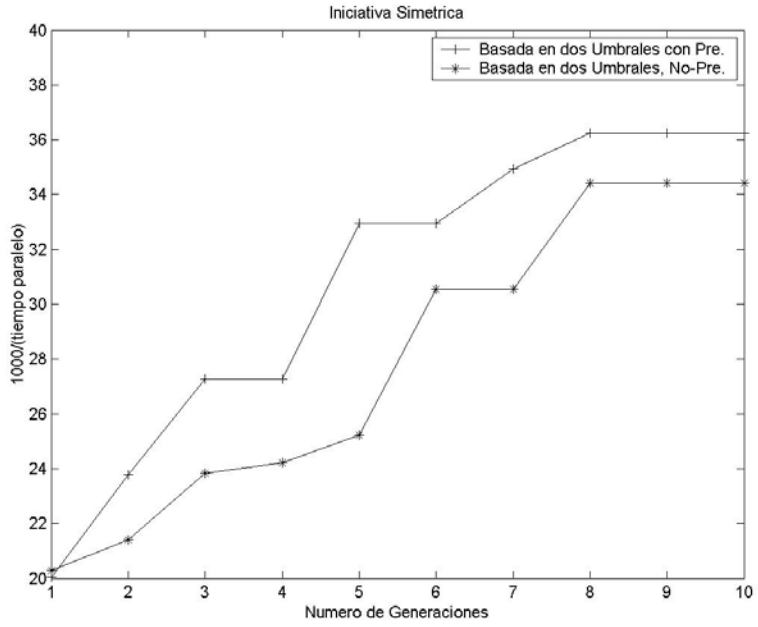


Figura 4.31: Idoneidad (1000/Tiempo paralelo) para las dos mejores alternativas con política de transferencia simétrica y el algoritmo de ramificación y poda (tamaño 9!)

4.3.2.3. Resultados obtenidos buscando en el espacio de todos los algoritmos con los tres parámetros LBF, TS1 y GRN

En la sección anterior se ha utilizado el algoritmo genético que hemos implementado para explorar el espacio de parámetros de las alternativas más comúnmente consideradas en la literatura para los procedimientos distribuidos de equilibrado de carga dinámica. Así, para proporcionar los datos que nos han permitido comparar los distintos procedimientos en problemas diferentes se han ido analizando por separado cada uno de dichos procedimientos fijando convenientemente los parámetros del procedimiento genérico de distribución de carga que se describe en el Capítulo 3.

En esta sección, el propósito es mostrar la capacidad de la metodología que proponemos para encontrar procedimientos de distribución de carga óptima. Para ello, hemos realizado dos tipos de experimentos para la multiplicación de matrices, el cálculo del número pi, y el cálculo de la integral determinada. Se han considerado los mismos tamaños para los que se hizo el trabajo experimental en la sección anterior. Se han considerado estos tres problemas de prueba debido a que presentan un comportamiento similar en cuanto a la convergencia a los parámetros LBF, TSD1 y TSD2 (para todos ellos se ha fijado GRN=P^2).

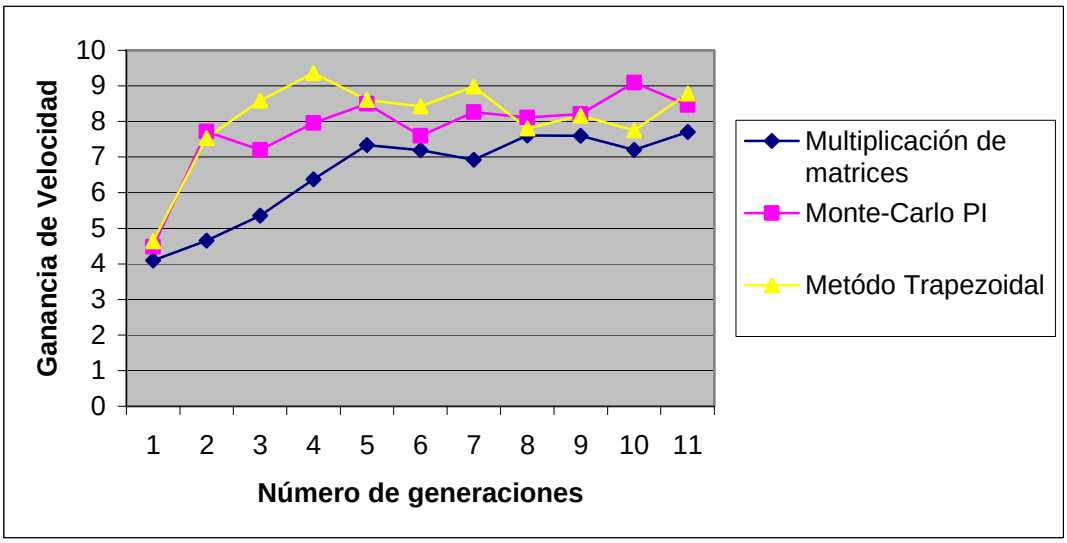


Figura 4.32. Ganancia de velocidad media obtenida en la exploración del espacio de búsqueda de todos los procedimientos de distribución de carga para cada *benchmark*

Los dos tipos de experimentos a los que nos hemos referido más arriba son los siguientes. En el primero de ellos se ha considerado cada uno de los problemas de prueba y se ha aplicado el algoritmo genético permitiendo que varíen no sólo GRN, LBF, TSD1, y TSD2, sino también los parámetros que conforman el resto del espacio de diseño y seleccionan entre las distintas alternativas existentes para las políticas que intervienen en los procedimientos de distribución de carga. Las Figuras 4.32 y 4.33 se refieren a este tipo de experimento. En la Figura 4.32 se muestra la evolución de la ganancia media obtenida a través de las sucesivas generaciones. Esta ganancia media se obtiene como la media de las ganancias de todos los individuos de la población. Como ya se ha dicho, la ganancia de un individuo se obtiene como la media de la ganancia obtenida para tres ejecuciones del programa paralelo con el correspondiente procedimiento de distribución de carga. En la Figura 4.33 se muestran los valores de la ganancia máxima, es decir la ganancia (ganancia media para las tres ejecuciones) del mejor individuo. Como se puede ver, en las Figuras 4.32 y 4.33 existe una curva para cada uno de los tres problemas de prueba considerados. En los tres casos se pone de manifiesto la mejora de calidad de la solución encontrada a medida que se suceden las generaciones. Al concluir el proceso de evolución del algoritmo genético, para cada problema tendremos una población de soluciones, que corresponden a distintos procedimientos de distribución de carga. A partir de esa población podremos determinar el mejor procedimiento (el asociado al mejor individuo). Los procedimientos que se han encontrado utilizan todos la política de transferencia simétrica. El resto de características y parámetros se muestran en la Tabla 4.15.

Tabla 4.15: Mejores procedimientos obtenidos tras la búsqueda en el espacio de diseño completo para cada benchmark (con problema fijo)

Programa de Prueba	Mejor Procedimiento encontrado	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima	Fitness = S. Máxima X Util. Máxima
Matrices (1000X1000)	Umbral (Preemption)	2	1	144	1	2	11	2	9,01	0,01	0,74	6,734
Cálculo de PI (10^8)	Umbral *(Preemption)*	2	1	144	1	2	13	1	9,81	0,03	0,61	6,3135
Integración numérica ($5X10^8$)	Dos Umbrales (Preemption)	2	1	144	2	2	12	1	9,83	0,02	0,84	8,442

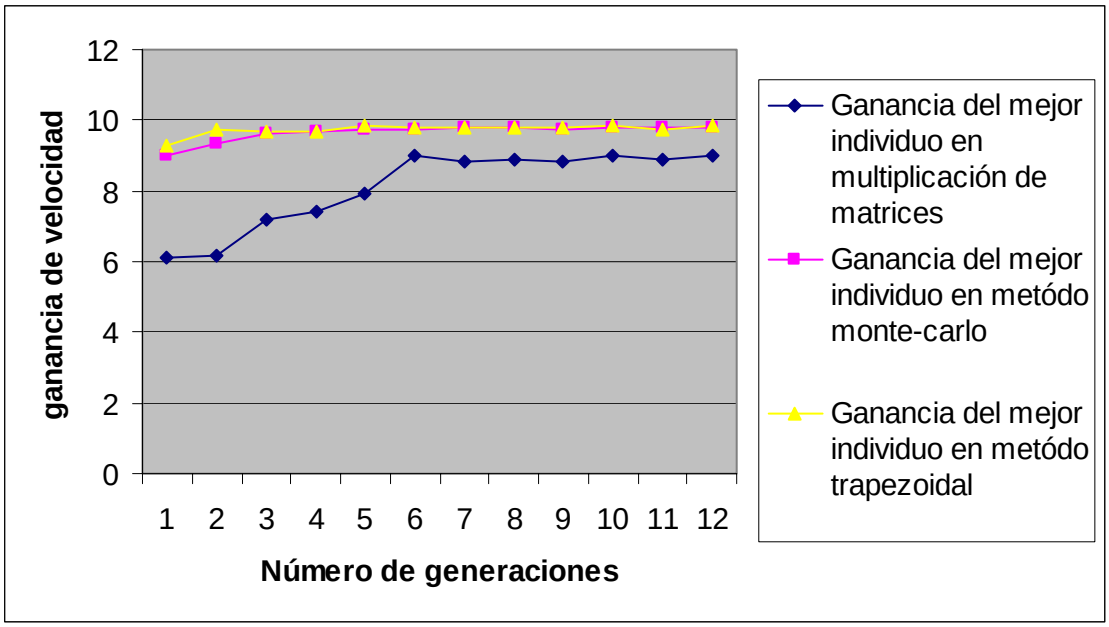


Figura 4.33. Ganancia de velocidad máxima obtenida en la exploración del espacio de búsqueda de todos los procedimientos de distribución de carga para cada *benchmark*

El otro experimento que hemos realizado consiste en ejecutar el procedimiento genético sin distinguir entre aplicaciones diferentes. En este caso, la ganancia que se asigna a cada individuo es la media de las obtenidas en tres ejecuciones de cada uno de los programas de prueba considerados. Las Figuras 4.34 y 4.35 muestran, respectivamente, la evolución de la ganancia media de los individuos de la población a medida que se suceden las generaciones del algoritmo genético, y la evolución de la ganancia máxima (la ganancia del mejor individuo de la población). Al final de la ejecución del algoritmo genético se puede obtener información de los procedimientos asociados a los individuos que se encuentren en la población final. En particular, el procedimiento de distribución de carga asociado al individuo con más idoneidad es el mejor procedimiento encontrado para el conjunto de programas de prueba considerado.

En nuestros experimentos, hemos encontrado que el procedimiento mejor es utiliza política de transferencia simétrica con las siguientes características:

Política de localización *(política de selección)*	LBF	Desv.	GRN	TSD1	Desv.	TSD2	Desv.	S Máxima	Desv.	Util. Máxima
Dos Umbrales *(Preemption)*	2	1	144	2	2	12	1	9,83	0,02	0,84

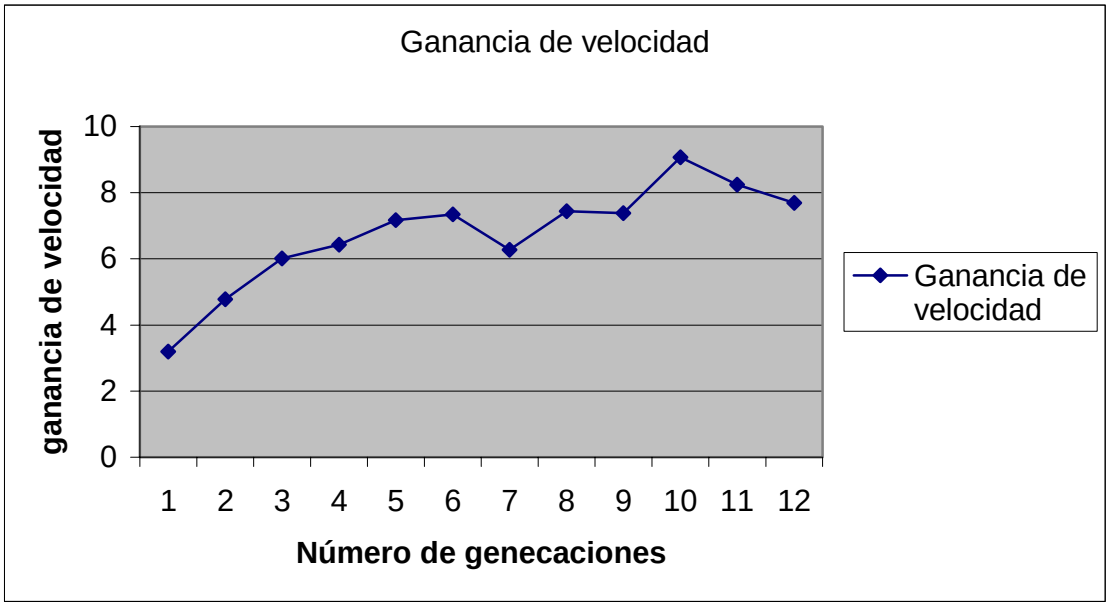


Figura 4.34. Ganancia de velocidad media obtenida en la exploración del espacio de búsqueda de todos los procedimientos de distribución de carga para todos los *benchmarks*

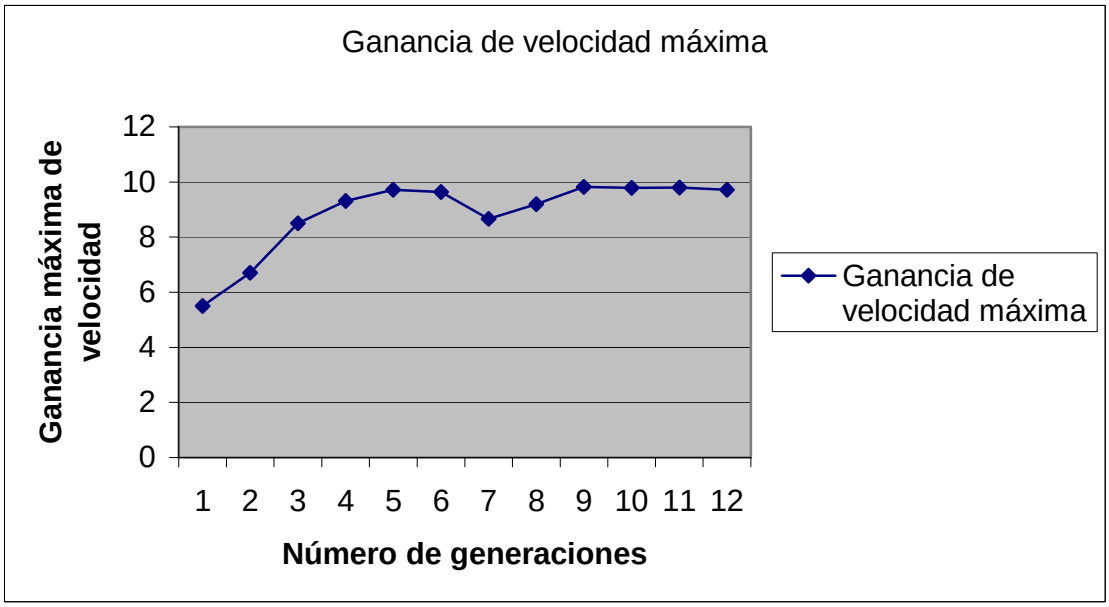


Figura 4.35. Ganancia de velocidad máxima obtenida en la exploración del espacio de búsqueda de todos los procedimientos de distribución de carga para todos los *benchmarks*

4.4 Conclusiones del capítulo

En este capítulo se ha utilizado un algoritmo genético que hemos desarrollado para explorar el espacio de diseño de los procedimientos distribuidos de equilibrado dinámico de carga. Dicho espacio de diseño queda definido a partir de las alternativas de las políticas que intervienen en un procedimiento de equilibrado de carga y de los valores de los parámetros que las particularizan, tal y como se describe en el Capítulo 3, que también proporciona un procedimiento genérico de equilibrado dinámico y distribuido de carga. La posibilidad de explorar el espacio de diseño mediante un algoritmo genético, con su capacidad de exploración y explotación de las soluciones encontradas, permitirá evaluar de forma automática la calidad de las distintas alternativas de distribución de carga para las diferentes aplicaciones paralelas. Además, también hará posible la determinación de los valores óptimos para los parámetros de que dependen los procedimientos de distribución de carga.

Tras situar, en la Sección 3.1, el trabajo que hemos realizado en el contexto de la investigación realizada acerca del uso de procedimientos basados en la computación evolutiva para el equilibrado dinámico y distribuido de carga, la Sección 3.2 se ha dedicado a describir el algoritmo genético que proponemos. Se trata de un algoritmo que utiliza elitismo como estrategia útil, no sólo para evitar la posible degradación de soluciones al aplicar los operadores de mutación y cruce, sino también para conseguir una mejor adaptación a la naturaleza dinámica del problema que se aborda, en la línea de los algoritmos evolutivos de memoria explícita para problemas de optimización dinámica. El procedimiento genético puede utilizar dos operadores de mutación. Uno de ellos ha sido introducido para mejorar la explotación de determinadas zonas del espacio de diseño con un menor número de generaciones.

La descripción y el análisis de los resultados experimentales obtenidos se ha llevado a cabo en la Sección 3.3, tras describir los cuatro problemas de prueba que se van a utilizar y las características de la plataforma donde se van a realizar los experimentos.

En primer lugar, se ha utilizado el algoritmo genético para explorar por separado el espacio de diseño de cada uno de los procedimientos de distribución de carga clasificados según su política de transferencia y para cada uno de los problemas de prueba considerados (con un tamaño fijo). En todos los casos se ha observado la superioridad de la política de

transferencia simétrica frente a la iniciada por el emisor y el receptor. Estos resultados coinciden con las conclusiones obtenidas en otros trabajos [ANT04]. En cuanto a las políticas de localización, las más eficientes suelen ser las basadas en uno o dos umbrales, y en cuanto a las políticas de selección, en algunos casos las mejores son las de corte forzoso (*preemption*) y en otros las que no utilizan corte forzoso (*no-preemption*). Los procedimientos con política de localización aleatoria son los que proporcionan las peores prestaciones. En estos experimentos se ha puesto de manifiesto la buena convergencia de los parámetros GRN, LBF, TSD1, y TSD2. En el caso de los problemas de prueba correspondientes a la multiplicación de matrices, el cálculo de la integral, y el cálculo del número pi, las mejores prestaciones se obtienen cuando GRN se fija entorno a P^2, y para ese valor, se obtienen convergencias de LBF, TSD1, y TSD2 a valores en entornos bastante estrechos. En el caso del algoritmo de ramificación y poda también se observa la convergencia para LBF, TSD1, y TDS2, aunque en este caso es más amplio el entorno al que convergen, dependiendo de las políticas que utilice el procedimiento de distribución de carga.

También se han hecho experimentos en los que el algoritmo genético ha explorado todo el espacio de diseño generado por la taxonomía que hemos propuesto, tanto si se fija la aplicación paralela (y el tamaño del problema), como si se consideran también varias aplicaciones al mismo tiempo. Los resultados obtenidos ponen de manifiesto la convergencia del algoritmo genético hacia soluciones coherentes con lo que se ha obtenido en los anteriores experimentos en cuanto a las políticas mejores (la política de transferencia simétrica) y a los valores de los parámetros LBF, GRN, TSD1, y TSD2.

En los experimentos que se han mostrado se ha considerado un número de procesadores fijo (igual a 12) y programas de prueba en los que no hemos variado el tamaño del problema (salvo para el algoritmo de ramificación y poda, para el que se ha considerado el problema del viajante de 8 y 9 ciudades). No obstante, es posible utilizar la metodología propuesta para explorar los cambios en los parámetros que, como es lógico, se producen al variar las características de las aplicaciones. Por ejemplo, en la Tabla 4.16 se muestra la convergencia de los parámetros en el caso de la multiplicación con distintos tamaños de matriz. Como se ve, fijada la granularidad a GRN=P^2=144, se tiene que los demás parámetros convergen a los valores que obtuvimos para este problema en la Sección 4.3.2.2. La Tabla 4.17 muestra los resultados de convergencia al variar el número de procesadores que

intervienen. Como se ve, si se fija igualmente GRN=P^2, el valor de TDS2 converge aproximadamente a *P*. Los valores de LBF y TSD1 no cambian con respecto a lo que se obtiene en la Sección 4.3.2.2 para la multiplicación de matrices.

Tabla 4.16: Convergencia de los parámetros para transferencia simétrica y dos umbrales en la multiplicación de matrices, con varios tamaños de matrices

Tamaño de Matrices	LBF	GRN	TSD1	TSD2	S. Mejor solución	Util. Mejor solución
750X750	2	144	3	14	7,56	0,62
1000X1000	2	144	1	13	8,93	0,72
1250X1250	2	144	1	12	9,15	0,78
1500X1500	2	144	2	11	9,36	0,76

Tabla 4.17: Convergencia de los parámetros para transferencia simétrica y dos umbrales en la multiplicación de matrices (1000x1000) para varios números de procesadores

Número de procesadores	LBF	GRN	TSD1	TSD2	S. Mejor solución	Util. Mejor solución
6	2	36	2	6	4,83	0,80
9	2	81	1	8	7,37	0,77
12	2	144	1	13	8,93	0,72

Los resultados obtenidos ponen de manifiesto la utilidad de la metodología que proponemos. Gracias al procedimiento genérico de distribución de carga que se ha construido a partir de la taxonomía definida y al algoritmo genético que permite explorar las posibilidades que encierra dicho procedimiento genético es posible analizar las distintas alternativas y realizar el diseño óptimo de procedimientos distribuidos de equilibrado dinámico de carga para una aplicación paralela (o conjunto de ellas) y para una plataforma dadas.

Capítulo 5

Conclusiones y principales aportaciones

La elección de un algoritmo de equilibrio de carga para una aplicación paralela no es una tarea fácil y, por otra parte, constituye una condición imprescindible para aprovechar eficientemente la capacidad del computador paralelo. En la literatura se han propuesto una gran cantidad de procedimientos, en su mayoría dentro del contexto de un dominio específico de aplicaciones. Las características de esas aplicaciones determinan la forma más eficaz para conseguir una distribución equilibrada de la carga entre los procesadores. Así, en ciertos casos es posible realizar una distribución equilibrada del trabajo a realizar en el momento de diseñar el algoritmo paralelo o en tiempo de compilación. La posibilidad de llevar a cabo esta distribución de carga estática de forma eficiente resulta beneficiosa por cuanto no va a suponer ninguna sobrecarga en el tiempo de ejecución del algoritmo paralelo. Sin embargo, no siempre es posible alcanzar una distribución de carga estática eficaz. Esto se debe, tanto a las características propias de la aplicación, como a las condiciones operativas del computador paralelo donde se va a ejecutar. Así, existen aplicaciones, como por ejemplo las basadas en algoritmos de ramificación y poda, en las que no se puede conocer (y tampoco especificar) con la suficiente precisión el volumen de trabajo a realizar. Por otra parte, las características de la plataforma paralela en las que se ejecuta la aplicación pueden cambiar dinámicamente. Por ejemplo, pueden producirse fallos en el hardware que afecten a las capacidades de los nodos de la plataforma paralela o, en servidores compartidos por distintas aplicaciones y usuario pueden cambiar las

condiciones de carga de los procesadores. Esta situación es usual, y debe tenerse muy presente en el contexto de la investigación en computación en GRID, que pretende crear plataformas paralelas virtuales de gran capacidad aprovechando la interconexión a través de las distintas infraestructuras de red. Por todo esto, la distribución dinámica de carga constituye una línea de investigación de gran relevancia en el ámbito de los sistemas paralelos y distribuidos [THE01].

Aunque, como hemos dicho, se han propuesto a muchos procedimientos de distribución dinámica de carga diferentes, todos tienen en común una serie de tareas [ZAK95]:

- La supervisión de la información de carga en los nodos.
- El intercambio de esta información entre los nodos (lo que supone alguna forma de sincronización).
- El cálculo de una nueva distribución de carga y la decisión acerca del movimiento de tareas para alcanzar un nuevo equilibrado de la carga.
- La migración de tareas, incluyendo código y/o datos.

De estas tareas comunes se deduce que entre la sobrecarga (*overhead*) que distribución dinámica de carga está, junto con la sobrecarga ocasionada por el cálculo de la nueva distribución, la sobrecarga de comunicación debido a la necesidad de intercambiar tanto información del estado de carga de los procesadores y como tareas. Esto hace que los procedimientos dinámicos centralizados sean poco adecuados en plataformas con un gran número de procesadores distribuidos, fundamentalmente porque el procesador central se convertirá en un cuello de botella que limitará la escalabilidad de la aplicación paralela haciendo que utilice de forma poco eficiente la propia plataforma. Además, si hay diferencias importantes entre los retardos de comunicación con el procesador central, puede ser difícil mantener una visión uniforme del estado de carga de los procesadores, fundamental para el correcto comportamiento de muchos de estos procedimientos centralizados.

Este trabajo se ha dedicado al estudio de las posibilidades de la computación evolutiva en el ámbito de los procedimientos de distribución dinámica de carga en computadores paralelos. Así, se han analizado dos posibles aproximaciones de los algoritmos genéticos al diseño de procedimientos de distribución de carga. Una primera

alternativa está en la utilización de un algoritmo genético como base para decidir la distribución óptima de carga entre los procesadores. La segunda consiste en encontrar los parámetros óptimos de los procedimientos de distribución de carga mediante los algoritmos genéticos.

En cuanto a la determinación de la distribución óptima de la carga entre los procesadores, los algoritmos evolutivos se han aplicado (en multitud de variantes e hibridaciones con otras técnicas) sobre todo al problema de la distribución de carga estática, problema de tipo NP-completo, en cuya resolución han demostrado una eficacia y robustez considerable. También han aparecido algunos procedimientos, como el descrito en [ZOM01] que abordan el problema de la distribución dinámica de la carga desde una perspectiva centralizada, donde el procesador central implementa un algoritmo genético que decide la forma de distribuir el trabajo entre los procesadores. Las aportaciones y conclusiones de esta Tesis en este ámbito se presentan en el Capítulo 2, donde se analizan los problemas de la distribución dinámica centralizada, y son las siguientes:

- Se ha realizado una implementación de un procedimiento centralizado de distribución de carga centralizado basado en un algoritmo genético. Este procedimiento está basado en el que se describe en [ZOM01] y nos ha permitido evaluar experimentalmente sus prestaciones y analizar las posibilidades de este tipo de aproximaciones para la distribución dinámica de carga.
- También se ha desarrollado un procedimiento de distribución de carga centralizado basado en la estrategia de ofertas. Este procedimiento, que hemos denominado EFP, permite aprovechar la potencia de sistemas con procesadores heterogéneos asignando cantidades de trabajo proporcionales a la potencia de cómputo de cada procesador.
- El procedimiento EFP se ha comparado con el procedimiento genético, utilizando como programa de prueba la multiplicación de matrices. Los resultados obtenidos muestran un comportamiento mejor del procedimiento EFP, que consigue una distribución más equilibrada de la carga y menores tiempos de ejecución (mayores ganancias de velocidad).

- Para el procedimiento genético se ha puesto de manifiesto experimentalmente el compromiso que debe establecerse entre la calidad de la distribución que se consigue y el número de generaciones que se ejecutan para determinar esa distribución de carga. Llega un punto en que la reducción del tiempo de procesamiento de las tareas asociada a una mejora en la distribución de la carga no compensa la sobrecarga asociada al tiempo necesario para ejecutar más generaciones del algoritmo genético. En los experimentos que hemos realizado, el menor tiempo de ejecución conseguido sigue siendo mayor que el que se consigue mediante el procedimiento EFP.
- Por supuesto, esta conclusión es válida para aplicaciones en las que, si bien las condiciones operativas de la carga pueden cambiar, existe una cierta capacidad para prever el volumen de trabajo de las tareas (como es el caso de la aplicación de matrices). El ámbito de estos procedimientos de distribución de carga genéticos los constituyen aplicaciones en las que existe una mayor variabilidad e imprevisión, como es el caso de los resultados que se proporcionan en [ZOM01] donde se considera una distribución aleatoria del volumen de trabajo asociado a cada tarea. En ese contexto, la robustez de un algoritmo genético como base para decidir la distribución óptima es fundamental.

La segunda aproximación que hemos indicado de los algoritmos genéticos a los problemas de la distribución dinámica de carga, no se ha planteado en muchos trabajos. Quizá quepa citar, en cierto sentido, el trabajo descrito recientemente en [HOV03] donde se intenta que cada nodo ajuste su vecindad en un procedimiento distribuido de equilibrado dinámico de carga de forma que se optimice la sobrecarga de comunicación. La dificultad de esta alternativa radica en el carácter dinámico del problema de optimización que se plantea. En este contexto, el trabajo realizado en esta Tesis se ha centrado en el estudio de los procedimientos distribuidos de equilibrado dinámico de carga que, como se ha indicado anteriormente, presentan un enorme interés dada la tendencia en el desarrollo de nuevas plataformas de cómputo paralelas basadas en el paradigma de la computación en GRID. Las aportaciones que se han realizado y las conclusiones a las que se ha llegado son las siguientes:

- A partir de un análisis de las etapas de los procedimientos de equilibrado dinámico de carga y de las políticas que intervienen en la especificación de las funciones que deben realizarse en dichos procedimientos se ha planteado una taxonomía para los procedimientos distribuidos de equilibrado de carga. Esta taxonomía queda definida a partir de las alternativas que se distinguen en las políticas de información, transferencia, localización, distribución, y selección, que deben especificarse en un procedimiento de distribución de carga. Esta taxonomía permite parametrizar los diferentes procedimientos de distribución de carga para luego poder aplicar la búsqueda genética en el espacio de diseño definido por la propia taxonomía y poder llevar a cabo la optimización de los parámetros que determinan el comportamiento de distribución de carga.
- Se ha diseñado un procedimiento general de equilibrado de carga dinámico y distribuido que puede incorporarse en una aplicación paralela, y que puede particularizarse al fijar el conjunto de parámetros que especifican a cada procedimiento de equilibrado de carga según la taxonomía que hemos propuesto. De esta forma, el problema de optimización del equilibrado dinámico de carga se aborda mediante una aproximación análoga a la que plantea la programación genética.
- Se ha implementado un algoritmo genético que permite explorar el espacio de diseño de los procedimientos de equilibrado dinámico y distribuido de la carga. El algoritmo genético utiliza elitismo para evitar los efectos disruptivos de los operadores de cruce y mutación y para mantener un enfoque de memoria externa en el proceso de optimización que presenta características de optimización dinámica. También se ha definido un nuevo operador de mutación que mejora la explotación del algoritmo genético en los puntos prometedores.
- Se ha aplicado el algoritmo genético tanto para optimizar los parámetros de distintos procedimientos de distribución de carga, considerando cuatro problemas de prueba, como para encontrar las características del mejor procedimiento para un problema de prueba o para un conjunto de problemas de prueba. El comportamiento observado para la convergencia del algoritmo genético ha sido satisfactorio obteniéndose conclusiones que confirman los resultados obtenidos en otros trabajos [ANT04].

En cuanto a las conclusiones relativas a las características de los procedimientos de equilibrado de carga, a las que se ha llegado a partir de la metodología que se ha aplicado, cabe citar las siguientes:

- En todos los casos analizados se ha observado la superioridad de la política de transferencia simétrica con política de información periódica frente a la iniciada por el emisor y el receptor ambas con la política de información en demanda. Estos resultados coinciden con las conclusiones obtenidas en otros trabajos [ANT04].
- En cuanto a las políticas de localización, las más eficientes suelen ser las basadas en uno o dos umbrales, y en cuanto a las políticas de selección, en algunos casos las mejores son las de corte forzoso (*preemption*) y en otros las que no utilizan corte forzoso (*no-preemption*). Los procedimientos con política de localización aleatoria son los que proporcionan las peores prestaciones.
- En los experimentos se ha puesto de manifiesto la buena convergencia de los parámetros GRN, LBF, TSD1, y TSD2. En el caso de los problemas de prueba correspondientes a la multiplicación de matrices, el cálculo de la integral, y el cálculo del número pi, las mejores prestaciones se obtienen cuando GRN se fija entorno a P^2, y para ese valor, se obtienen convergencias de LBF, TSD1, y TSD2 a valores en entornos bastante estrechos. En el caso del algoritmo de ramificación y poda también se observa la convergencia para LBF, TSD1, y TDS2, aunque en este caso es más amplio el entorno al que convergen, dependiendo de las políticas que utilice el procedimiento de distribución de carga.
- En los experimentos realizados en los que el algoritmo genético explora todo el espacio de diseño generado por la taxonomía que hemos propuesto, tanto si se fija la aplicación paralela (y el tamaño del problema), como si se consideran también varias aplicaciones al mismo tiempo, los resultados obtenidos muestran convergencia del algoritmo genético hacia soluciones coherentes (con los experimentos en los que se fija el procedimiento de distribución de carga) en cuanto a las políticas mejores (la política de transferencia simétrica) y a los valores de los parámetros LBF, GRN, TSD1, y TSD2.

Algunas de las aportaciones que se han descrito se han publicado en el trabajo:

[ALD04] Aldasht, M.; Ortega, J.; Puntonet, C.G.; Díaz, A.F.:"A Genetic Exploration of Dynamic Load Balancing Algorithms". IEEE Conference on Evolutionary Computation, Portland, Oregon, 2004.

Trabajo futuro. El trabajo de investigación realizado y que se describe en esta Memoria no agota el tema de investigación que hemos iniciado. Las limitaciones de tiempo que han condicionado el desarrollo de nuestro trabajo también han implicado limitaciones en cuanto a la posibilidad de completar el trabajo experimental en plataformas diversas de las que no se dispone en nuestro grupo, pero a las que sí tenemos acceso. Así pues, se pueden citar una serie de trabajos que nos permitirán aprovechar las posibilidades de la metodología que proponemos, y que se pueden agrupar en tres frentes que corresponden a las aplicaciones paralelas, las plataformas paralelas, y la propia metodología de optimización dinámica basada en algoritmos genéticos:

- Por un lado habría que analizar el comportamiento del procedimiento de optimización de parámetros en nuevas aplicaciones paralelas de interés, fundamentalmente en aquellas que tienen un comportamiento más irregular, y donde resulta crucial disponer de un procedimiento eficiente de distribución dinámica de carga.
- También habría que considerar plataformas con diversas topologías de interconexión, y nodos con capacidad de cómputo heterogénea. De esta forma se podría estudiar la optimización de las políticas de distribución y algunas de las alternativas de las políticas de información. La aplicación de la metodología que hemos propuesto a plataformas distribuidas, constituidas por diferentes arquitecturas conectadas a través de redes de área local y área amplia, tendría una clara incidencia en el ámbito de la computación en GRID, donde el equilibrado de carga dinámico y distribuido cobra especial interés.
- Por último, en cuanto al procedimiento genético de optimización que se ha empleado, existen aspectos interesantes que deben estudiarse. Por un lado está la implementación de otras alternativas que se han propuesto [BRA01] para los problemas de optimización dinámica. En la implementación que hemos utilizado aquí se ha utilizado una aproximación de memoria explícita basada en el elitismo. Si bien consideramos que los resultados obtenidos por el algoritmo

genético son satisfactorios, habría que explorar la incorporación de mecanismos de mantenimiento de la diversidad [SAR98] que, según ciertos trabajos [BRA99], afectan a las prestaciones que proporcionan los enfoques elitistas. En este ámbito, también puede ser una línea prometedora la utilización de planteamientos de optimización genética multiobjetivo, para abordar situaciones en las que se pone de manifiesto la necesidad de un cierto compromiso entre la ganancia y la utilización que se consigue con una distribución de carga.

De esta forma, damos por concluida esta Memoria, donde se ha descrito el trabajo de investigación que, financiado a través de los proyectos TIC2000-1348 y TIC2001-2845, realiza aportaciones significativas e inicia aproximaciones prometedoras en los ámbitos de la distribución de carga en plataformas paralelas y la computación evolutiva.

Apéndice A

Código C-MPI para el algoritmo EFP descrito en capítulo 3

```
/*************************************************************************
* este programa implementa el algoritmo EFP propuesto en Capítulo 2 de    *
* tesis. El algoritmo está implementado para la multiplicación de 2       *
* matrices A y B, entonces el resultado será C = AXB, para facilitar la   *
* implementación suponemos que las matrices A y B son matrices cuadradas y*
* de tamaño NXN cada una. Como descrito en el capítulo 2 el algoritmo EFP *
* funciona en modo maestro trabajador. Donde el maestro lleva la          *
* implementación del algoritmo de distribución de carga, y al mismo tiempo*
* realiza cálculos en su tiempo ocioso. Mientras un proceso trabajador    *
* recibe instrucciones y datos del proceso maestro, realiza el cálculo    *
* necesario sobre los datos y manda el resultado al maesrtro junto con    *
* la información necesaria para el maestro para la evaluación de su carga.*
* entonces el maestro calcula la carga que se tiene que enviar al         *
* trabajador, y se la manda.                                              *
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#define N 1000            // tamaño de la matriz
#define FROM_MASTER 1    // tipo de mensaje
#define FROM_WORKER 2    // tipo de mensaje

/*************************************************************************
* esta función se puede utilizar por los diferentes procesos para         *
* chequear si hay algún mensaje pendiente esperandole.                    *
*************************************************************************/
int msg_pending(void)
{
    MPI_Status status;
    int flag;
```

```
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag,
&status);
    return (flag);
}

/*************************************************************************
* esta función multiplica una fila de la matriz A por la matriz B y la    *
* devuelve en su offset en C.                                             *
*************************************************************************/
void mastermult(int row, float *a, float *b, float *c)
{
  int j,k;
  for (j = 0; j < N; j++)
   {
    c[row*N+j] = 0.0;
    for (k = 0; k < N; k++)
     {
      c[row*N+j]+=a[row*N+k]*b[k*N+j];
     }
   }
}

/*************************************************************************
* esta función devuelve la potencia del CPU donde reside un proceso en    *
* terminos de MIPS. Según viene en el archivo /proa/cpuinfo creado por el *
* sistema operativo LINUX. Esta información se utiliza para poder asignar *
* a un proceso como maestro o trabajador. El proceso maestro reside en el *
* procesador más rápido.                                                  *
*************************************************************************/
float get_cpumips()
{
  FILE* fp;
  char buffer[1024];
  size_t bytes_read;
  char* match;
  float cpumips;

  /* lee el contenido del archivo /proc/cpuinfo en el buffer  */
  fp = fopen ("/proc/cpuinfo", "r");
  bytes_read = fread (buffer, 1, sizeof (buffer), fp);
  fclose (fp);
  if (bytes_read == 0 || bytes_read == sizeof (buffer))
    return;
  /* NUL-terminate the text.  */
  buffer[bytes_read] = '\0';

  /* localizar la línea que comienza con "bogomips".  */
  match = strstr (buffer, "bogomips");
  sscanf (match, "bogomips : %f", &cpumips);

  return cpumips;
}
MPI_Status status; /* un variable del comunicador de MPI */

main(int argc, char *argv[])
{
int numprocs,np,   /* Número de procesos */
```

```
    taskid, MASTER,/* identificador del proceso */
    numworkers,    /* número de trabajadores */
    source,   /* identificador del proceso fuente en caso de send, recv */
    dest,     /* identificador del proceso destino en caso de send, recv */
    mtype,    /*    tipo del mensaje */
    rows, rowcont, /* contadores de filas */
    averow, extra,    /* Sirve para calcular el número de filas a enviar */
    offset, tmpoff,   /* para determinar la dirección de una fila       */

    i, j, k,            /* misc */

    count, granularity, /* contador y variable de la granularidad */
    rond, tmprond, /* para determiner las rondas que hace el maestro */
    rondcont, nronds;  /* contadores de rondas */

    float *a, *b, *c,  /* las matrices para multiplicar el la de resul. */
          *pmips;       /* potencia de un procesador */
    int *wspace;        /* puntero para los procesos activos */

double starttime,
       endtime, t1, t2,
       tex, texr,
       tcom, waittime;  /* variables para medir el tiempo */

FILE* fptr;       /* puntero para el fichero de resultados */

MPI_Init(&argc,&argv); /* Inicializa MPI */

MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);


/*************************************************************************
* para ejecutar este programa hay que introducirle el número de procesos  *
* que van a calcular la multiplicación de las matrices. Entonces para     *
* llevarlo a cabo el proceso 0 recibe la información sobre la potencia    *
* de todos los procesadores. Luego se ordenan los procesadores según su   *
* potencia y se utilizan los primeros np más potentes. Donde np es el     *
* número de procesadores indicado por le usuario para la ejecución del    *
* problema.                                                               *
*************************************************************************/
if (taskid == 0)
 {
  if (argc != 2)
      {    printf("Utilización: %s <Numero de procesos>\n", argv[0]);
           MPI_Finalize();
           exit(0);
      }
  np=atoi(argv[1]); // número de procesadores que se van a utilizar


  if (np < 2 || numprocs < np)
      {    printf("como mínimo se utiliza 2 procesos y como máximo
%d\n",numprocs);
           MPI_Finalize();
           exit(0);
      }
```

```
  pmips = (float *)malloc(numprocs*sizeof(float));
  pmips[0] = get_cpumips();

  for (i=1; i<numprocs; i++)
    MPI_Recv(&pmips[i], 1, MPI_FLOAT, i, 2, MPI_COMM_WORLD, &status);

  float mips=0.0;
  int p;

  wspace = (int *)malloc(np*sizeof(int));
  for (i=0; i<np; i++)
   {
    for (j=0; j<numprocs; j++)
      if (pmips[j] > mips)
        { p = j;
          mips = pmips[j];
        }
    pmips[p] = 0.0;
    mips = 0.0;
    wspace[i] = p;
   }
  for (i=0;i<np;i++)
    printf("seleccionar proceso %d\n",wspace[i]);
  for (i=1; i<numprocs; i++)
   {
    MPI_Send(&np, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
    MPI_Send(&wspace[0], np, MPI_INT, i, 1, MPI_COMM_WORLD);
   }
 }

if (taskid > 0)
 {
  pmips = (float *)malloc(sizeof(float));
  pmips[0] = get_cpumips();
  MPI_Send(&pmips[0], 1, MPI_FLOAT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(&np, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
  wspace = (int *)malloc(np*sizeof(int));
  MPI_Recv(wspace, np, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
 }


MASTER = wspace[0];
numworkers = np-1;




/***********************       proceso maestro    ************************/

/*************************************************************************
* apartir de aquí empieza el proceso maestro, donde los procesos están    *
* ordenados en el vector wspace según su potencia. Entonces como hemos    *
* comentado antes el procesador más potente se asigna como maestro.       *
```

```
* Según la oden del vector wspace el procesador más potente queda en      *
* el lugar 0.                                                              *
****************************************************************************/
if (taskid == MASTER) {

  starttime = MPI_Wtime();
  a= (float *)malloc(N*N*sizeof(float));
  b= (float *)malloc(N*N*sizeof(float));
  c= (float *)malloc(N*N*sizeof(float));

  srand48((long)100);
  for ( i=0; i<N*N; i++)
      a[i] = b[i] = drand48();

/****************************************************************************
* granrand es un número aleatorio entre 1 y numprocs. Esto se utiliza      *
* para calcular la granularidad que será igual a                           *
* (n*granrand*((numprocs*numprocs+numprocs)/2))/1; donde n pertenece a     *
* grupo {1,2,3}.                                                           *
****************************************************************************/
  double granrand = ceil(numprocs*drand48());

  granularity = (2*granrand*((numprocs*numprocs+numprocs)/2))/1;
  averow = N/granularity;
  extra = N%granularity;

  nronds = (granularity < N) ? granularity : N;

  rond = 0;       // inicializar ronda
  offset = 0;

  fptr = fopen ("central.txt", "a");

  fprintf(fptr,"Multiplicacion de dos (%dX%d) Matrices\n", N,N);
  fprintf(fptr,"Maestro-Trabajador, hay %d trabajadores\n",numworkers);
  fprintf(fptr,"Una distribución escalonada, granularidad = %d
\n",granularity);

  mtype = FROM_MASTER;




/****************************************************************************
* primero el proceso maestro calcula número total de rondas que se van     *
* a realizar comprobando si la granularidad es mayor que el tamaño del     *
* problema, si es mayor entonces el número de rondas total pasa a ser      *
* igual que N. sino entonces el número total de rondas pasa a ser igual    *
* que la granularidad. Luego el proceso maestro empieza realizar las       *
* primeras rondas de distribución. La primera distribución se realiza en   *
* una manera ecalonada de manera que proceso P1 recibe un número de filas  *
* igual a (1*rows), el proceso P2 recibe un número de filas igual a        *
* (2*rows) hasta Pn que recibe (n*rows) filas. Esta distribución es        *
```

```
* necesaria para evitar que los procesos vuelven a pedir más trabajo al   *
* mismo tiempo.                                                            *
***************************************************************************/

  for (dest=1; dest<=numworkers; dest++) {
    tmprond = rond;
    rond += dest;
    rows = (rond <= extra) ? dest*(averow+1) : dest*averow;
    if ((extra > tmprond)&&((extra-tmprond)<dest))
            rows += (extra-tmprond);

    MPI_Send(&offset, 1, MPI_INT, wspace[dest], mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, wspace[dest], mtype, MPI_COMM_WORLD);

    count = N*N;
    MPI_Send(&b[0], count, MPI_FLOAT, wspace[dest],
              mtype, MPI_COMM_WORLD);

    count = rows*N;
    MPI_Send(&a[offset], count, MPI_FLOAT, wspace[dest],
              mtype, MPI_COMM_WORLD);

    offset = offset + rows*N;
    }


/***************************************************************************
* aquí empieza el proceso maestro a distribuir el trabajo dinámicamente   *
* entre los procesos trabajadores. En un bucle cada vez que el proceso    *
* maestro realiza cálculo de una fila chequea si hay algún mensaje        *
* pendiente de algún trabajador. Si hay mensajes pendientes serán         *
* resultados de los trabajadores, entonces el maestro recibe los          *
* resultados, junto con la información de velocidad del trabajador,       *
* según esta información el maestro calcula el número de filas que se     *
* van a mandar a este trabajador. Se calcula un factor de velocidad (nr)  *
* para cada proceso. nr = ceil(Tl/Tp) donde Tl es el tiempo empleado en   *
* el procesador más lento para calcular una unidad de trabajo, Tp es el   *
* tiempo empleado por el trabajador actual para calcular una unidad de    *
* trabajo. luego a este trabajador se le envía nr*averow filas. donde     *
* averows es el promedio del número de filas calculado según la           *
* granularidad. Esta operación se repite por el maestro hasta que no      *
* haya más filas para multiplicar.                                        *
***************************************************************************/

  double maxtime = 0.0;
  int nx, range, row, done;
  rowcont = 0;
  done = 1;
  while (rond < nronds) {
    if (done)
     {
      rond ++;
      done = 0;
      rows = (rond <= extra) ? averow+1 : averow;
      rowcont += rows;
      row = offset/N;
      range = (offset/N)+rows;
      offset = offset + rows*N;
```

```
     }
    if (!done)
     {
      if (row<range)
       {
        mastermult(row,a,b,c);
        row++;
       }
      else done = 1;
     }
    if (rond == nronds) break;
    if (!msg_pending()) continue;
    tmpoff = offset;
    mtype = FROM_WORKER;
    MPI_Recv(&texr, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
            mtype, MPI_COMM_WORLD, &status);
    source = status.MPI_SOURCE;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype,
            MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype,
            MPI_COMM_WORLD, &status);
    count = rows*N;
    MPI_Recv(&c[offset], count, MPI_FLOAT, source,
              mtype, MPI_COMM_WORLD,&status);

    mtype = FROM_MASTER;
    offset = tmpoff;
    dest = source;

    tex = texr/(rows/averow);
    maxtime = (tex > maxtime) ? tex : maxtime;

    nx = (int)ceil(maxtime/tex);

    rondcont = ((rond+nx)>nronds) ? (nronds-rond) : nx;
    tmprond = rond;
    rond += rondcont;

    rows = (rond <= extra) ? rondcont*(averow+1) : rondcont*averow;

    if ((extra > tmprond)&&((extra-tmprond)<rondcont)) rows += (extra-
tmprond);

            round %d\n",rows,dest,rond);
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);

    count = rows*N;
    MPI_Send(&a[offset], count, MPI_FLOAT, dest, mtype, MPI_COMM_WORLD);

    offset = offset + rows*N;
  }

  while (row<range)
   {
    mastermult(row,a,b,c);
    row++;
   }
```

```
  fprintf(fptr,"Total number of rows Multiplied at the Master:
                %d \n", rowcont);
  fprintf(fptr, "Proc.\tRows\tTtot\tTex\tTcomm\tTwait\n");
  rond = 1;

/*************************************************************************
* aquí el proceso maestro después de haber acabado todo el calculo y la   *
* distribución, tiene que recibir los resultados del último trabajo que   *
* está en realización en los procesos trabajadores.                       *
*************************************************************************/
  while (rond<=numworkers) {

    mtype = FROM_WORKER;

    MPI_Probe(MPI_ANY_SOURCE, mtype, MPI_COMM_WORLD, &status);
    source = status.MPI_SOURCE;
    MPI_Recv(&texr, 1, MPI_DOUBLE, source, mtype,
            MPI_COMM_WORLD, &status);
    MPI_Recv(&offset, 1, MPI_INT, source, mtype,
            MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype,
            MPI_COMM_WORLD, &status);
    count = rows*N;
    MPI_Recv(&c[offset], count, MPI_FLOAT, source, mtype,
             MPI_COMM_WORLD,&status);

    offset = -1;
    mtype = FROM_MASTER;

    MPI_Send(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD);

     mtype = FROM_WORKER;
    MPI_Recv(&rowcont, 1, MPI_INT, source, mtype,
            MPI_COMM_WORLD, &status);
    MPI_Recv(&tex, 1, MPI_DOUBLE, source, mtype,
            MPI_COMM_WORLD, &status);
    MPI_Recv(&tcom, 1, MPI_DOUBLE, source, mtype,
            MPI_COMM_WORLD, &status);
    MPI_Recv(&waittime, 1, MPI_DOUBLE, source, mtype,
            MPI_COMM_WORLD,&status);

    fprintf(fptr,"%d\t%d\t%6.3f\t%6.3f\t%6.3f\t%6.3f\n",
                 source,rowcont,tex+tcom+waittime,tex,tcom,waittime);
    printf("Time Information Recieved From Task %d\n",source);
    rond++;
    }

  free(a);
  free(b);
  free(c);

endtime = MPI_Wtime();

/* Escribir resultados */

fprintf(fptr,"At the Master, The Parallel Time was: %6.3f \n",
              endtime-starttime);
```

```
fclose(fptr);

}  /* Final del proceso maestro */


                        //          Proceso Trabajador
/*************************************************************************
* el proceso trabajador, ejecuta un bucle de modo que cada vez que recibe *
* trabajo se queda calculando hasta que termina el calculo del trabajo que*
* tiene, entonces manda el resultado para el maestro junto con el tiempo  *
* empleado en el calculo. Luego recibe una cantidad de trabajo            *
* proporcional a su velocidad.                                            *
*************************************************************************/
if (taskid != MASTER) {
  int exists = 0;
  for (i=1; i<=numworkers; i++)
    if (taskid == wspace[i]) exists = 1;
 if (exists == 1) {
  rowcont = 0;
  tcom = 0.0;
  tex = 0.0;
  waittime = 0.0;

  mtype = FROM_MASTER;
  source = MASTER;

// Rcibiendo mensaje del maestro
  t1 = MPI_Wtime();
  MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  t2 = MPI_Wtime();
  waittime += t2-t1;
  t1 = MPI_Wtime();

  MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  rowcont += rows;
    count = N*N;

  b = (float *)malloc(count*sizeof(float));
  MPI_Recv(b, count, MPI_FLOAT, source, mtype, MPI_COMM_WORLD, &status);


    count = rows*N;
  a = (float *)malloc(count*sizeof(float));
  MPI_Recv(a, count, MPI_FLOAT, source, mtype, MPI_COMM_WORLD, &status);

  t2 = MPI_Wtime();
  tcom += t2-t1;

  while (1)
   {
    texr = 0.0;
    t1 = MPI_Wtime();
    c = (float *)malloc(count*sizeof(float));
    for (i = 0; i < rows; i++)
       for (j = 0; j < N; j++)
         {
         c[i*N+j] = 0.0;
         for (k = 0; k < N; k++)
```

```
            {
              c[i*N+j]+=a[i*N+k]*b[k*N+j];
            }
          }

    t2 = MPI_Wtime();
    tex += t2-t1;
    texr = t2-t1;
    mtype = FROM_WORKER;

    t1 = MPI_Wtime();

    MPI_Send(&texr, 1, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);

    t2 = MPI_Wtime();
    waittime += t2-t1;
    t1 = MPI_Wtime();

    MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&c[0], rows*N, MPI_FLOAT, MASTER, mtype, MPI_COMM_WORLD);

    free(a);
    free(c);

    mtype = FROM_MASTER;
    source = MASTER;

    MPI_Recv(&offset, 1, MPI_INT, source, mtype,
            MPI_COMM_WORLD, &status);

//  salir de esto al recibir offset = -1,
//  esto indica que el maestro no dispone de más trabajo
    if (offset == -1) break;
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    rowcont += rows;
       count = rows*N;
    a = (float *)malloc(count*sizeof(float));
    MPI_Recv(a, count, MPI_FLOAT, source, mtype,
            MPI_COMM_WORLD, &status);
    t2 = MPI_Wtime();
    tcom += t2-t1;
   }

// Finalizar
  free(b);
  mtype = FROM_WORKER;
  MPI_Send(&rowcont, 1, MPI_INT, MASTER, mtype,  MPI_COMM_WORLD);
  MPI_Send(&tex, 1, MPI_DOUBLE, MASTER, mtype,  MPI_COMM_WORLD);
  MPI_Send(&tcom, 1, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&waittime, 1, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
 }
}  /* end of worker process*/

  MPI_Finalize();
  exit(0);
} /* of main */
```

Apéndice B

Código C-MPI para el procedimiento generico de equilibrio de carga descrito en capítulo 4

```
/**************************************************************************
* Este programa implementa el procedimiento genérico de distribución de   *
* carga descrito en el capítulo 4.                                        *
**************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <limits.h>
#include <mpi.h>

#define N 1000

#define WORK 1     /* Tipo de mensaje */
#define RESULT 2   /* Tipo de mensaje */
#define INFO 3     /* Tipo de mensaje */

typedef struct {
   long bogomips;
   long availmem;
   char hostname[MPI_MAX_PROCESSOR_NAME];
} pinfo;

typedef struct {
   int rows;          /* número de filas que corresponden a una tarea */
   int xrow;        /* la fila que se va a ejecuatr */
   int offset;        /* el índice de las filas en la matriz A */
   int tmpoffset;   /* un índice temporario */
   int state;         /* el estado de la tarea */
   int migselect;   /* para la selección de la tarea para ser migrada */
} taskinfo;

MPI_Status status;
```

```
/*************************************************************************
* esta función multiplica una fila de la matriz A, por la matriz B y      *
* devuelve el resultado en la matriz C.                                   *
*************************************************************************/
void multiply(int row, float *a, float *b, float *c)
{
  int j,k;
  for (j = 0; j < N; j++)
   {
    c[row*N+j] = 0.0;
    for (k = 0; k < N; k++)
     {
      c[row*N+j]+=a[row*N+k]*b[k*N+j];
     }
   }
}

long getbogomips(void)

/* para saber la potencia de un nodo en función de MIPS */
{
  FILE* fp;
  char buffer[1024];
  size_t bytes_read;
  char* match;
  long mips;

  /* Read the entire contents of /proc/cpuinfo into the buffer.  */
  fp = fopen ("/proc/cpuinfo", "r");
  bytes_read = fread (buffer, 1, sizeof (buffer), fp);
  fclose (fp);

  if (bytes_read == 0 || bytes_read == sizeof (buffer))
    return;
  buffer[bytes_read] = '\0';

  /* Locate the line that starts with "bogomips".  */
  match = strstr (buffer, "bogomips");
  sscanf (match, "bogomips : %d", &mips);

  return(mips);
}

long getavailmem(void)

/* devuelve la memoria libre en un nodo*/
{
  FILE* fp;
  char buffer[1024];
  size_t bytes_read;
  char* match;
  long availmem;

 /* leer el contenido de /proc/meminfo  */
  fp = fopen ("/proc/meminfo", "r");
```

```
  bytes_read = fread (buffer, 1, sizeof (buffer), fp);
  fclose (fp);

  if (bytes_read == 0 || bytes_read == sizeof (buffer))
    return;
  buffer[bytes_read] = '\0';

  /* localizar la linea que comienza con la palabra "MemFree".  */
  match = strstr (buffer, "MemFree");
  sscanf (match, "MemFree : %d", &availmem);

  return(availmem);
}

/*************************************************************************
* esta función elimina una tarea de la cola de trabajo. Se puede utilizar *
* para eliminar las tareas que migran de la cola de un proceso hasta otro *
* proceso.                                                                *
*************************************************************************/
void taskrem(int m, int tasks, taskinfo *migtask)
{
 int i;
 taskinfo *taskp;
 for(i=m, taskp = migtask, taskp += m; i<tasks-1; i++){
  *taskp = *(taskp+1); taskp++;
 }
}

/*************************************************************************
* esta función sirve para insertar una tarea recién llegada en la cola de *
* trabajo.                                                                *
*************************************************************************/
void taskinsert(int m, int tasks, taskinfo *migtask)
{
 int i;
 taskinfo *taskp;

 taskp = migtask;
 taskp += tasks-1;
 for(i=tasks-1; i>m; i--){
  *taskp = *(taskp-1); taskp--;
 }
}

/*************************************************************************
* devuelve el proceso cuyo índice de carga es el menor entre todos los    *
* procesos                                                                *
*************************************************************************/
int minimum(int tasks, int *ldindex)
{
 int i, min = 0;
 for(i=1;i<tasks;i++) if (ldindex[i] < ldindex[min]) min = i;
 return(min);
}
```

```
/*************************************************************************
* devuelve el proceso cuyo índice de carga es el mayor entre todos los    *
*procesos                                                                 *
*************************************************************************/
int maximum(int tasks, int *ldindex)
{
 int i, max = 0;
 for(i=1;i<tasks;i++) if (ldindex[i] > ldindex[max]) max = i;
 return(max);
}

/*************************************************************************
* esta función se utiliza para detectar el fin de trabajo en todos los    *
* procesos                                                                *
*************************************************************************/
int done(int ldindex, int procid, int tasks, int *ldindeces)
{
 int i, ok;
 ok = 1;
 for(i=0;i<tasks;i++)
   if (ldindeces[i] != 0) ok = 0;

 if (ldindex != ldindeces[procid]) ok = 0;

 return(ok);
}

/*************************************************************************
* esta función se utiliza por la política de localización basada en       *
* sondeo. Lo que hace está función es escoger un proceso aleatoriamente.  *
*************************************************************************/
int pollnode(int procid, int nprocs)
{
  int i, randnum, n;
  int fd = open("/dev/urandom", O_RDONLY);
  randnum = procid;
  if (nprocs > 1)
    while (randnum == procid) {
      read(fd, &n, sizeof(n));
      randnum = abs((double)(nprocs)*n/INT_MAX);
    }
  close(fd);

  return(randnum);
}

/*************************************************************************
* esta función es para localizar procesos como emisores o receptor para   *
* la política de transferencia simétrica. Esta función dispone de 5       *
* alternativas  de localización. Según la alternativa indicada por el     *
* usuario a traves el parámetro introducido LP esta función localiza      *
* proceso/s emisor/es y receptor/es para la política de transferencia     *
* simétrica. Las 5 alternativas de localización implementadas aquí son:   *
* 1. según un umbral, si el índice de carga de un proceso es menor que    *
* este valor umbral entonces este proceso se localiza como receptor, en   *
* caso contrario se localiza como emisor. 2. utilizando dos umbrales      *
* superior y inferior, un proceso se localiza como emisor si su índice    *
* de carga es mayor que el umbras superior, mientras que un proceso       *
```

```
* se localiza como receptor si su índice de carga es menor que el        *
* umbral inferior. 3. según máximo y mínimo, donde el proceso que tiene  *
* el índice de carga máximo se localiza como emisor y el proceso que     *
* tiene el índice de carga mínimo se localiza como receptor. 4. el       *
* máximo se localiza como emisor y éste localiza a un receptor           *
* aleatorio. 5. el mínimo se localiza como receptor y éste localiza a    *
* un emisor aleatorio.                                                   *
*************************************************************************/
int Processor_Locate(int procid, int *located,int *loadindex,int lp,int
      nprocs,int OTSD,int LTSD,int UTSD)
{
  int i,j,loc,randnum;
  int rangeflag = 1;
  MPI_Win locwin;     /* a shared memory window for the randomized location
of processes */

  switch (lp) {
      case 0: /* Threshold-based Location */
            for (i=0;i<nprocs;i++) {
              if (loadindex[i] < OTSD)
                located[i] = -1;
              if(loadindex[i] >= OTSD)
                located[i] = 1;
              }
            break;

      case 1: /* Threshold-Based Location Policy (Using tow ) */
            for (i=0;i<nprocs;i++) {
              if (loadindex[i] <= LTSD)
                located[i] = -1;
              if(loadindex[i] >= UTSD)
                located[i] = 1;
              if(((loadindex[i] < UTSD)&&(loadindex[i] >
LTSD))||(LTSD>UTSD))
                located[i] = 0;
              }
            break;

      case 2: /* Shortest-Based Location Policy */
            for (i=0;i<nprocs;i++)  located[i] = 0;
            i = minimum(nprocs,loadindex);
            j = maximum(nprocs,loadindex);
            if ((loadindex[j] - loadindex[i]) > 0){
            located[i] = -1; located[j] = 1;}
            break;

      case 3: /* Random-Based Location Policy (Randomize the Receiver) */
            for (i=0;i<nprocs;i++)  located[i] = 0;
            j = maximum(nprocs,loadindex);
            if(procid == j) {
              randnum = pollnode(procid,nprocs);
              MPI_Win_create(&loc, sizeof(int), 1, MPI_INFO_NULL,
MPI_COMM_WORLD, &locwin);
              loc = randnum;
            }
            else {
              MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
MPI_COMM_WORLD, &locwin);}
```

```
            MPI_Win_fence(0, locwin);
            if(procid != j) MPI_Get(&loc, 1, MPI_INT, j, 0, 1, MPI_INT,
locwin);
            MPI_Win_fence(0, locwin);
            i = loc;
            if ((loadindex[j] - loadindex[i]) > 0){
            located[i] = -1; located[j] = 1;}

            MPI_Win_free(&locwin);
            break;

      case 4: /* Random-Based Location Policy (Randomize the Sender) */
            for (i=0;i<nprocs;i++)  located[i] = 0;
            j = minimum(nprocs,loadindex);
            if(procid == j) {
              randnum = pollnode(procid,nprocs);
              MPI_Win_create(&loc, sizeof(int), 1, MPI_INFO_NULL,
MPI_COMM_WORLD, &locwin);
              loc = randnum;
            }
            else MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
MPI_COMM_WORLD, &locwin);

            MPI_Win_fence(0, locwin);
            if(procid != j) MPI_Get(&loc, 1, MPI_INT, j, 0, 1, MPI_INT,
locwin);
            MPI_Win_fence(0, locwin);
            i = loc;
            if ((loadindex[i] - loadindex[j]) > 0){
            located[i] = 1; located[j] = -1;}

            MPI_Win_free(&locwin);
            break;

      default: rangeflag = 0;
  }
  return(rangeflag);
}

/*********************************************************************
* esta función es para localizar procesos como emisores o receptor para   *
* la política de transferencia iniciada por el emisor. Esta función       *
* dispone de 3 alternativas  de localización. Según la alternativa        *
* indicada por el usuario a traves el parámetro introducido LP esta       *
* función localiza proceso/s receptor/es para la política de transferencia*
* iniciada por el emisor. Las 3 alternativas de localización implementadas*
* aquí son: 1. según un umbral, sondea hasta que encuentre algún proceso  *
* con índice de carga menor que el valor umbral para asignarlo como       *
* receptor o hasta un límite de sondeo predeterminado. 3. según un mínimo,*
* donde el proceso emisor sondea hasta un límite de sondeo y localiza al  *
* proceso con el índice de carga más corto, entre todos los procesos      *
* sondeados, como receptor.                                               *
*********************************************************************/
int Locate_Receiver(int procid,int *located,int *loadindex,int lp,int
nprocs,int OTSD,int pollimit,int sndrcv)
{
  int i,j,randnode;
```

```
  int rangeflag = 1;
  for (i=0;i<nprocs;i++)  located[i] = 0;
  if(sndrcv == 1) {
    switch (lp) {
      case 0: /* Random-based Location */
            randnode = pollnode(procid,nprocs);
            located[procid] = 1;
            located[randnode] = -1;
            break;

      case 1: /* Threshold-Based Location Policy */
            randnode = pollnode(procid,nprocs);
            i=0;
            while (i<pollimit) {
              if(loadindex[randnode]<OTSD) break;
              randnode = pollnode(procid,nprocs);
              i++;
            }
            located[procid] = 1;
            located[randnode] = -1;
            break;

      case 2: /* Shortest-Based Location Policy */
            j = pollnode(procid,nprocs);
            i=0;
            while (i<pollimit) {
              randnode = pollnode(procid,nprocs);
              if(loadindex[randnode]<loadindex[j]) j = randnode;
              i++;
            }
            located[procid] = 1;
            located[j] = -1;
            break;

      default: rangeflag = 0;
    }
  }
  return(rangeflag);
}

/*************************************************************************
* esta función se utiliza para evaluar la carga y seleccionar tas         *
* tareas que deben migrar. Básicamente se introduce la localización,      *
* realizada por las funciones de localización implementadas arriba,       *
* junto con los índices de carga de el/los proceso/s emisor/es y          *
* receptor/es. Como primer pasa la  función calcula la sobrecarga del     *
* proceso emisor según la formúla:                                        *
* overld = loadindex[procid] - ((sumhload+sumlload)/(senders+receivers)); *
* es decir la sobrecarga de un proceso emisor es la diferencia entre su   *
* índice de carga y la carga media de todos los procesos emisores y       *
* receptores. Luego, después de haber calculado la sobrecarga, el proceso *
* emisor va a seleccionar tareas de su cola de trabajo para transferirlas.*
* Estas tareas seleccionadas deben presentar trabajo igual a la sobrecarga*
* del proceso emisor. La función implementa dos políticas de selección que*
* son, 1. (preemption) donde una tarea que está ejecutande puede ser      *
* seleccionada para migrar. 2. (no-preemption) donde sólo las tareas que  *
* están en espera pueden ser seleccionadas para migrar.                   *
*************************************************************************/
```

```
int Task_Select(int procid, int *located, int *loadindex, taskinfo
*migtask,  int sp, int ctask, int nprocs, int ntasks, int *recnum, int
*sendnum, int *nmigtasks)
{
  int rangeflag = 1;
  int i, sumhload, sumlload, overld, tasks, senders, receivers, nmigs;

  sumhload = 0; sumlload = 0; senders = 0; receivers = 0; nmigs = 0;
  for (i=0;i<nprocs;i++){
    if (located[i] == 1){
      senders++;
      sumhload += loadindex[i];}
    else if (located[i] == -1){
      receivers++;
      sumlload += loadindex[i];}
  }

  switch (sp) {
      case 0: /* Preemption selection of Migrant Tasks */
            if (located[procid] == 1) {
              overld = loadindex[procid] -
((sumhload+sumlload)/(senders+receivers));
              tasks = 0;
              nmigs = 0;
              if ((overld <= 0)||(receivers==0)) break;
              i = ctask;
              while((tasks<=overld)&&(i<ntasks)) {
                if (migtask[i].state != 2) {
                  tasks += 1;
                  migtask[i].migselect = 1; /* select it to migrate */
                  nmigs++;
                }
                i++;
              }
              if(tasks > overld) {
                migtask[i-1].migselect = 0; /* undo the selection */
                nmigs--;
              }
            }
            break;

      case 1: /* Non-Preemption selection of Migration Tasks */
            if (located[procid] == 1) {
              overld = loadindex[procid] -
((sumhload+sumlload)/(senders+receivers));
              tasks = 0;
              nmigs = 0;
              if ((overld <= 0)||(receivers==0)) break;
              i = ntasks;
              while((tasks<=overld)&&(i>ctask)){
                i--;
                if (migtask[i].state != 2) {
                  tasks += 1;
                  migtask[i].migselect = 1; /* select it to migrate */
                  nmigs++;
                }
              }
              if(tasks > overld) {
```

```
                migtask[i].migselect = 0; /* undo the selection */
                nmigs--;
              }
            }
            break;

      default: rangeflag = 0;
    }
  *sendnum = senders; *recnum = receivers; *nmigtasks = nmigs;
  return(rangeflag);
}

int main(int argc, char *argv[])
{
  int   numprocs,        /* Número de procesos */
      procid,            /* Identificador de proceso */
      dest,              /* proceso destino */
      source,            /* proceso fuente */
      tmpload,           /* índice de carga temporanio */
      randnum,
      sendnum,
      recnum,
      LTSD,              /* umbral inferior */
      UTSD,              /* umbral superior */
      OTSD,              /* un umbral */
      count,
      i, j, k, m,        /* misc */
      fin,
      namelen,
      mtype,             /* tipo de mensaje */
      ntasks,            /* número de tareas en un proceso */
      tasktot,           /* número total de tareas */
      averow, extra,
      nmigtasks,   /* número de tareas migradas */
      offset,
      rows,
      tmpoffset,
      avetasks, tasks,
      ctask,crow,  /* tarea actual bajo ejecución */
      rcvdrows,tmprows,
      LP, SP, TP,        /* parámetros par alas poílicas de localización,
selección y transferencia*/
      LBF, DLB,    /* frequencia de equilibrado de carga */
      rangeflag,
      randnode;

  int *sgnal;
  MPI_Win sgwin;   /* ventana para acceso remoto de memoria */
  MPI_Group Group;
  int pollimit, *finkey;
  int *loadindex;        /* índice de carga */
  int *located;
  pinfo *procinfo;
  taskinfo *migtask;
  float *a, *b, *c, *rc;
  float *pac1, *pac2;
  double starttime, endtime,
  t1, t2, tcomp, putilization,
```

```
       granp, LBP, TSDP1, TSDP2;

  FILE* fptr;           /* puntero al fichero de resultados */
/*********************************************************************/

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &procid);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  procinfo = (pinfo *)malloc(numprocs*sizeof(pinfo));

  procinfo[procid].bogomips = getbogomips();
  procinfo[procid].availmem = getavailmem();
  MPI_Get_processor_name(procinfo[procid].hostname,&namelen);

  b = (float *)malloc(N*N*sizeof(float));

  located = (int *)calloc(numprocs,sizeof(int));
  loadindex = (int *)calloc(numprocs,sizeof(int));

  rcvdrows = 0;
  ntasks = 0;


  LP = atoi(argv[1]);
  SP = atoi(argv[2]);
  TP = atoi(argv[3]);

  LBP = atof(argv[4]);
  granp = atof(argv[5]);
  TSDP1 = atof(argv[6]);
  TSDP2 = atof(argv[7]);

  tasktot = numprocs* numprocs;     // calcula la granularidad
  OTSD = (int)ceil(TSDP1*tasktot);  // calcula el umbral
  LTSD = OTSD;
  LBF = (int)ceil(LBP*tasktot);     // calcula la frecuencia del equilibrio
de carga
  UTSD = (int)ceil(TSDP2*tasktot);  // calcula el umbral superior

  migtask = NULL;
  a = NULL;
  c = NULL;
  rc = NULL;
  tcomp = 0.0;

 if (procid == 0) {
    a = (float *)realloc(a,N*N*sizeof(float));
    c = (float *)realloc(c,N*N*sizeof(float));

    srand48((long)5);
    for ( i=0; i<N*N; i++){
      a[i] = b[i] = drand48();
    }
    if (argc != 8) /* Exit, if wrong number of command arguments */
      {    printf("Usage: %s <The 6 Arguments>\n", argv[0]);
           MPI_Finalize();
```

```
           exit(0);
      }
%d\n",LTSD,UTSD,OTSD);
    printf("Total number of tasks is %d\n",tasktot);
    printf("Load Balancing Freq. = %d ,,, TSDP2 = %6.3f\n", LBF, TSDP2);*/

    averow = N/tasktot;
    extra = N%tasktot;
    migtask = (taskinfo *)realloc(migtask,tasktot*sizeof(taskinfo));
    offset = 0;
    for(i=0;i<tasktot;i++) {
      rows = (i < extra) ? averow+1 : averow;
      migtask[i].rows = rows;
      migtask[i].xrow = 0;
      migtask[i].offset = offset;
      migtask[i].tmpoffset = offset;
      migtask[i].state = 0;   /* nueva tarea */
      migtask[i].migselect = 0; /* no seleccionada para migrar */
      offset += rows*N;
    }

    rcvdrows = N;
    ntasks = tasktot;
    loadindex[0] = tasktot;
    starttime = MPI_Wtime();
}

  count = N*N;
  MPI_Bcast(b,count,MPI_FLOAT,0,MPI_COMM_WORLD);

  for(i=0;i<numprocs;i++) {
    MPI_Bcast(&loadindex[i],1,MPI_INT,i,MPI_COMM_WORLD);
    MPI_Bcast(&procinfo[i].bogomips,1,MPI_LONG,i,MPI_COMM_WORLD);
    MPI_Bcast(&procinfo[i].availmem,1,MPI_LONG,i,MPI_COMM_WORLD);

MPI_Bcast(&procinfo[i].hostname,MPI_MAX_PROCESSOR_NAME,MPI_CHAR,i,MPI_COMM_
WORLD);}

  ctask = 0;
  crow = 0;
  DLB = 1;

/*************************************************************************
* política de transferencia inciada por el emisor.                       *
* si el índice de carga de la tarea es mayor que el umbral superior      *
* entonces este proceso en emiso.                                        *
* el proceso emisor localiza un receptor según una de las políticas de   *
* localización indicadas arriba.                                         *
* el emisor pide la información de carga al receptor.                    *
* el emisor realiza selección de tareas.                                 *
* por último las tareas seleccionadas en el emisor se transfieren al hasta*
* el receptor.                                                           *
*************************************************************************/

  switch (TP) {
      case 0:
      /*****************************************************************/
      sgnal = (int *)malloc((numprocs+1)*sizeof(int));
```

```
      finkey = (int *)malloc(numprocs*sizeof(int));
      pollimit = (int)(numprocs/2);

      MPI_Win_create(&sgnal[0], (MPI_Aint)((numprocs+1)*sizeof(int)),
sizeof(int),
                  MPI_INFO_NULL, MPI_COMM_WORLD, &sgwin);

      finkey[procid] = loadindex[procid];
      for(i=0;i<numprocs;i++)
MPI_Bcast(&finkey[i],1,MPI_INT,i,MPI_COMM_WORLD);

      do {
      tmprows = rcvdrows;
      tmpload = loadindex[procid];

    if (DLB==0) {
      finkey[procid] = loadindex[procid];
      for(i=0;i<numprocs;i++)
MPI_Bcast(&finkey[i],1,MPI_INT,i,MPI_COMM_WORLD);

      for(i=0;i<numprocs;i++) sgnal[i] = -1;

      if (loadindex[procid] > UTSD) sgnal[numprocs] = 1; /* Send */
      else sgnal[numprocs] = -1; /* Receive */

      /* Locate Some Receiver */
      Locate_Receiver(procid,located,loadindex,LP,numprocs,OTSD,pollimit,sg
nal[numprocs]);

      for(i=0;i<numprocs;i++) if(located[i] == -1) {
        sgnal[i] = i;
        break;
      }

      j = 0;
      m = -1;

      MPI_Win_fence(0, sgwin);
      if (i<numprocs)
        MPI_Get(&j, 1, MPI_INT, i, (MPI_Aint)numprocs, 1, MPI_INT, sgwin);
      MPI_Win_fence(0, sgwin);
      if (i<numprocs){
        if (j==1){
          located[i] = 0; located[procid] = 0;
        }
        if (located[procid]){
          m = procid;
          MPI_Put (&m, 1, MPI_INT, i, (MPI_Aint)m, 1, MPI_INT, sgwin);
        }
      }
      MPI_Win_fence(0, sgwin);

      if((i<numprocs)&&(j==1)) sgnal[i] = -1;

      /* On-Demand Information Exchange */
      for(i=0;i<numprocs;i++) if(sgnal[i]>-1){
        source = procid; dest = sgnal[i]; mtype = INFO;
        if(sgnal[numprocs] == -1) {
```

```
          located[source] = -1;
          located[dest] = 1;
          MPI_Send(&loadindex[source], 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD);
        }
        if(sgnal[numprocs] == 1) {
          MPI_Recv(&loadindex[dest], 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD, &status);
        }
      }

      /* Do Task Selection */
      rangeflag =
Task_Select(procid,located,loadindex,migtask,SP,ctask,numprocs,ntasks,
                        &recnum,&sendnum,&nmigtasks);

      /* Transfer Tasks */
      if ((recnum > 0)&&(located[procid] == 1)) {

  /*  printf("\n*** Redistributing from process %d \n",procid);*/
      for(m=0;m<ntasks;m++) if (migtask[m].migselect==1){
        if (migtask[m].state==1) {
               ntasks++; migtask = (taskinfo
*)realloc(migtask,ntasks*sizeof(taskinfo));
               taskinsert(m,ntasks,migtask);
               rows = migtask[m].rows;
          migtask[m].rows = migtask[m].xrow;
          rows -= migtask[m].rows;
          migtask[m].state = 2;          /* executed task */
          migtask[m].migselect = 0;       /* not selected to migrate */
          offset = migtask[m].offset;
          offset += migtask[m].rows*N;
          tmpoffset = migtask[m].tmpoffset;
          tmpoffset += migtask[m].rows*N;
          m++; ctask++;
          migtask[m].rows = rows;
          migtask[m].xrow = 0;
          migtask[m].offset = offset;
          migtask[m].tmpoffset = tmpoffset;
          migtask[m].state = 0;          /* a new task */
          migtask[m].migselect = 1;       /* select it to migrate */
          break;
        }
        break;
      }
      tmpoffset = migtask[m].tmpoffset;
      avetasks = nmigtasks/recnum;
      extra = nmigtasks%recnum;
      j = 0; mtype = WORK;
      for (i=0; i<numprocs; i++) if (located[i] == -1) {
        dest = i; j++;
        tasks = (j <= extra) ? avetasks+1 : avetasks;
      /*  printf("*** sending %d tasks from proc. %d, to proc.
%d\n",tasks,procid,dest);*/
        MPI_Send(&tasks, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
        if(tasks == 0) continue;
          for(k=0;k<tasks;k++){
```

```
            MPI_Send(&migtask[m].offset, 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD);
            rows = migtask[m].rows; rcvdrows -= rows; loadindex[procid] -=
1;
            MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            count = rows*N;
            offset = migtask[m].tmpoffset;
            MPI_Send(&a[offset], count, MPI_FLOAT, dest, mtype,
MPI_COMM_WORLD);
            migtask[m].migselect = 2; /* Task m has been migrated */
            m++;
          }
        }
        for(m=0;m<ntasks;m++) if (migtask[m].migselect == 2) break;
        rows = 0;
        for(i=0;i<ntasks;i++) {
          if (migtask[i].migselect == 2) {
            rows += migtask[i].rows;
            taskrem(i, ntasks, migtask); ntasks--; i--;
            migtask = (taskinfo *)realloc(migtask,ntasks*sizeof(taskinfo));
          }
        }
        count = rows*N;
        for(i=m;i<ntasks;i++) migtask[i].tmpoffset -= count;
          if(rows>0) {
            pac1 = a; pac2 = a;
            pac1 += tmpoffset; pac2 += tmpoffset+count;
            for(i=tmpoffset+count; i<tmprows*N; i++) { *pac1 = *pac2;
pac1++; pac2++; }
            pac1 = c; pac2 = c;
            pac1 += tmpoffset; pac2 += tmpoffset+count;
            for(i=tmpoffset+count; i<tmprows*N; i++) { *pac1 = *pac2;
pac1++; pac2++; }
            a = (float *)realloc(a,rcvdrows*N*sizeof(float));
            c = (float *)realloc(c,rcvdrows*N*sizeof(float));
          }
      /*  printf("*** Sending from %d ,  DONE !!!\n",procid);*/
      }
      if ((sendnum > 0)&&(located[procid] == -1)){
        mtype = WORK;
        for(i=0;i<sendnum;i++) {
          MPI_Recv(&tasks, 1, MPI_INT, MPI_ANY_SOURCE, mtype,
MPI_COMM_WORLD, &status);
          source = status.MPI_SOURCE;
          if (tasks == 0) continue;
          for(k=0;k<tasks;k++){
            ntasks++;
            migtask = (taskinfo *)realloc(migtask,ntasks*sizeof(taskinfo));
            MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
            MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
            migtask[ntasks-1].offset = offset;
            migtask[ntasks-1].rows = rows;
            tmpoffset = rcvdrows*N; rcvdrows += rows;
            a = (float *)realloc(a,rcvdrows*N*sizeof(float));
            count = rows*N;
```

```
MPI_Recv(&a[tmpoffset],count,MPI_FLOAT,source,mtype,MPI_COMM_WORLD,&status)
;
            loadindex[procid] += 1;
            c = (float *)realloc(c,rcvdrows*N*sizeof(float));
            migtask[ntasks-1].tmpoffset = tmpoffset;
            migtask[ntasks-1].xrow = 0;
            migtask[ntasks-1].state = 0;         /* a new task */
            migtask[ntasks-1].migselect = 0;   /* not selected to migrate
*/
          }
        }
      /*  printf("*** Receiving at %d ,  DONE !!!\n",procid);*/
      }
    }
      if (loadindex[procid] > 0 ){
        t1 = MPI_Wtime();
        DLB++;
        while(migtask[ctask].xrow != migtask[ctask].rows) {
          multiply(crow,a,b,c); crow++;
          migtask[ctask].xrow += 1;
        }
        loadindex[procid] -= 1;
        if (migtask[ctask].xrow == 1) migtask[ctask].state = 1; /* Task
under Execution */
        if (migtask[ctask].xrow == migtask[ctask].rows) {
          migtask[ctask].state = 2;             /* executed task */
          ctask++;
        }
        t2 = MPI_Wtime();
        tcomp += t2-t1;
      }

      if ((DLB>=LBF)||(loadindex[procid] == 0 )) DLB = 0;

      } while (!done(tmpload,procid,numprocs,finkey));

      free(sgnal);
      free(finkey);
      MPI_Win_free(&sgwin);

      break;
      /*****************************************************************/
/*************************************************************************
* política de transferencia inciada por el receptor.                    *
* si el índice de carga de la tarea es menor que el umbral inferio       *
* entonces este proceso es receptor.                                     *
* el proceso receptor localiza un emisor según utilizando sondeo hasta un *
* límite y localiza el proceso con el mayor índice de carga como emisor  *
* el receptor pide la información de carga al emisor.                    *
* el emisor realiza selección de tareas.                                 *
* por último las tareas seleccionadas en el emisor se transfieren al hasta*
* el receptor.                                                           *
*************************************************************************/

      case 1: /* Receiver-Initiated Transfer Policy */
      sgnal = (int *)malloc((numprocs+1)*sizeof(int));
      finkey = (int *)malloc(numprocs*sizeof(int));
```

```
      pollimit = (int)(numprocs/2);

      MPI_Win_create(&sgnal[0], (MPI_Aint)((numprocs+1)*sizeof(int)),
sizeof(int),
                  MPI_INFO_NULL, MPI_COMM_WORLD, &sgwin);

      finkey[procid] = loadindex[procid];
      for(i=0;i<numprocs;i++)
MPI_Bcast(&finkey[i],1,MPI_INT,i,MPI_COMM_WORLD);

      do {
      tmprows = rcvdrows;
      tmpload = loadindex[procid];

    if (DLB==0) {

      finkey[procid] = loadindex[procid];
      for(i=0;i<numprocs;i++)
MPI_Bcast(&finkey[i],1,MPI_INT,i,MPI_COMM_WORLD);

      for(i=0;i<numprocs;i++) sgnal[i] = -1;

      if (loadindex[procid] < OTSD) sgnal[numprocs] = -1;   /* Receive */
      else sgnal[numprocs] = 1; /* Send */

      /* Locate Some Receiver */
      for (i=0;i<numprocs;i++)  located[i] = 0;
      if (sgnal[numprocs] == -1) {
        j = pollnode(procid,numprocs);
        i=0;
        while (i<pollimit) {
          randnode = pollnode(procid,numprocs);
          if(loadindex[randnode]>loadindex[j]) j = randnode;
          i++;
        }
        located[procid] = -1;
        located[j] = 1;
      }

      for(i=0;i<numprocs;i++) if(located[i] == 1) {
        sgnal[i] = i;
        break;
      }

      j = 0;
      m = -1;

      MPI_Win_fence(0, sgwin);
      if (i<numprocs)
        MPI_Get(&j, 1, MPI_INT, i, (MPI_Aint)numprocs, 1, MPI_INT, sgwin);
      MPI_Win_fence(0, sgwin);
      if (i<numprocs){
        if (j==-1){
          located[i] = 0; located[procid] = 0;
        }
        if (located[procid]==-1){
          m = procid;
          MPI_Put (&m, 1, MPI_INT, i, (MPI_Aint)m, 1, MPI_INT, sgwin);
```

```
        }
      }
      MPI_Win_fence(0, sgwin);

      if((i<numprocs)&&(j==-1)) sgnal[i] = -1;

      /* On-Demand Information Exchange */
      for(i=0;i<numprocs;i++) if(sgnal[i]>-1){
        source = procid; dest = sgnal[i]; mtype = INFO;
        if(sgnal[numprocs] == 1) {
          located[source] = 1;
          located[dest] = -1;
          MPI_Send(&loadindex[source], 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD);
        }
        if(sgnal[numprocs] == -1) {
          MPI_Recv(&loadindex[dest], 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD, &status);
        }
      }

      /* Do Task Selection */
      rangeflag =
Task_Select(procid,located,loadindex,migtask,SP,ctask,numprocs,ntasks,
                        &recnum,&sendnum,&nmigtasks);

      /* Transfer Tasks */
      if ((recnum > 0)&&(located[procid] == 1)) {

  /*  printf("\n*** Redistributing from process %d \n",procid);*/
      for(m=0;m<ntasks;m++) if (migtask[m].migselect==1){
        if (migtask[m].state==1) {
                ntasks++; migtask = (taskinfo
*)realloc(migtask,ntasks*sizeof(taskinfo));
                taskinsert(m,ntasks,migtask);
                rows = migtask[m].rows;
          migtask[m].rows = migtask[m].xrow;
          rows -= migtask[m].rows;
          migtask[m].state = 2;          /* executed task */
          migtask[m].migselect = 0;       /* not selected to migrate */
          offset = migtask[m].offset;
          offset += migtask[m].rows*N;
          tmpoffset = migtask[m].tmpoffset;
          tmpoffset += migtask[m].rows*N;
          m++; ctask++;
          migtask[m].rows = rows;
          migtask[m].xrow = 0;
          migtask[m].offset = offset;
          migtask[m].tmpoffset = tmpoffset;
          migtask[m].state = 0;          /* a new task */
          migtask[m].migselect = 1;       /* select it to migrate */
          break;
        }
        break;
      }
      tmpoffset = migtask[m].tmpoffset;
      avetasks = nmigtasks/recnum;
      extra = nmigtasks%recnum;
```

```
      j = 0; mtype = WORK;
      for (i=0; i<numprocs; i++) if (located[i] == -1) {
        dest = i; j++;
        tasks = (j <= extra) ? avetasks+1 : avetasks;
      /*  printf("*** sending %d tasks from proc. %d, to proc.
%d\n",tasks,procid,dest);*/
        MPI_Send(&tasks, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
        if(tasks == 0) continue;
          for(k=0;k<tasks;k++){
            MPI_Send(&migtask[m].offset, 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD);
            rows = migtask[m].rows; rcvdrows -= rows; loadindex[procid] -=
1;
            MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            count = rows*N;
            offset = migtask[m].tmpoffset;
            MPI_Send(&a[offset], count, MPI_FLOAT, dest, mtype,
MPI_COMM_WORLD);
            migtask[m].migselect = 2; /* Task m has been migrated */
            m++;
          }
        }
        for(m=0;m<ntasks;m++) if (migtask[m].migselect == 2) break;
        rows = 0;
        for(i=0;i<ntasks;i++) {
          if (migtask[i].migselect == 2) {
            rows += migtask[i].rows;
            taskrem(i, ntasks, migtask); ntasks--; i--;
            migtask = (taskinfo *)realloc(migtask,ntasks*sizeof(taskinfo));
          }
        }
        count = rows*N;
        for(i=m;i<ntasks;i++) migtask[i].tmpoffset -= count;
          if(rows>0) {
            pac1 = a; pac2 = a;
            pac1 += tmpoffset; pac2 += tmpoffset+count;
            for(i=tmpoffset+count; i<tmprows*N; i++) { *pac1 = *pac2;
pac1++; pac2++; }
            pac1 = c; pac2 = c;
            pac1 += tmpoffset; pac2 += tmpoffset+count;
            for(i=tmpoffset+count; i<tmprows*N; i++) { *pac1 = *pac2;
pac1++; pac2++; }
            a = (float *)realloc(a,rcvdrows*N*sizeof(float));
            c = (float *)realloc(c,rcvdrows*N*sizeof(float));
          }
      /*  printf("*** Sending from %d ,  DONE !!!\n",procid);*/
      }
      if ((sendnum > 0)&&(located[procid] == -1)){
        mtype = WORK;
        for(i=0;i<sendnum;i++) {
          MPI_Recv(&tasks, 1, MPI_INT, MPI_ANY_SOURCE, mtype,
MPI_COMM_WORLD, &status);
          source = status.MPI_SOURCE;
          if (tasks == 0) continue;
          for(k=0;k<tasks;k++){
            ntasks++;
            migtask = (taskinfo *)realloc(migtask,ntasks*sizeof(taskinfo));
```

```
            MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
            MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
            migtask[ntasks-1].offset = offset;
            migtask[ntasks-1].rows = rows;
            tmpoffset = rcvdrows*N; rcvdrows += rows;
            a = (float *)realloc(a,rcvdrows*N*sizeof(float));
            count = rows*N;

MPI_Recv(&a[tmpoffset],count,MPI_FLOAT,source,mtype,MPI_COMM_WORLD,&status)
;
            loadindex[procid] += 1;
            c = (float *)realloc(c,rcvdrows*N*sizeof(float));
            migtask[ntasks-1].tmpoffset = tmpoffset;
            migtask[ntasks-1].xrow = 0;
            migtask[ntasks-1].state = 0;         /* a new task */
            migtask[ntasks-1].migselect = 0;   /* not selected to migrate
*/
          }
        }
      /*  printf("*** Receiving at %d ,  DONE !!!\n",procid);*/
      }
    }
      if (loadindex[procid] > 0 ){
        t1 = MPI_Wtime();
        DLB++;
        while(migtask[ctask].xrow != migtask[ctask].rows) {
          multiply(crow,a,b,c); crow++;
          migtask[ctask].xrow += 1;
        }
        loadindex[procid] -= 1;
        if (migtask[ctask].xrow == 1) migtask[ctask].state = 1; /* Task
under Execution */
        if (migtask[ctask].xrow == migtask[ctask].rows) {
          migtask[ctask].state = 2;             /* executed task */
          ctask++;
        }
        t2 = MPI_Wtime();
        tcomp += t2-t1;
      }
      if ((DLB>=LBF)||(loadindex[procid] == 0 )) DLB = 0;

      } while (!done(tmpload,procid,numprocs,finkey));

      free(sgnal);
      free(finkey);
      MPI_Win_free(&sgwin);

      break;

/*************************************************************************
* política de transferencia simétrica.                                  *
* Primero: intercambia la información.                                  *
* Segundo: localiza procesos, esmisores o receptores según una de       *
las políticas de localización implementadas arriba.                     *
* Tercer: los procesos emisores seleccionan tareas.                     *
* Cuarto: realizar la transferencia de la tareas seleccionadas desde los  *
```

```
* emisores hasta los receptores.                                        *
*************************************************************************/
      case 2:
      /*****************************************************************/
      do {

      tmpload = loadindex[procid];
      tmprows = rcvdrows;

    if (DLB==0) {
      /* Periodic Information Policy */
      for(i=0;i<numprocs;i++)
MPI_Bcast(&loadindex[i],1,MPI_INT,i,MPI_COMM_WORLD);

      /* Do Processor Location */
      rangeflag = Processor_Locate(procid, located, loadindex, LP,
                        numprocs, OTSD, LTSD, UTSD);

      /* Do Task Selection */
      rangeflag = Task_Select(procid,located,loadindex,migtask,SP,ctask,
                        numprocs,ntasks,&recnum,&sendnum,&nmigtasks);

      /* Transfer Tasks */
      if ((recnum > 0)&&(located[procid] == 1)) {

  /*  printf("\n*** Redistributing from process %d \n",procid);*/
      for(m=0;m<ntasks;m++) if (migtask[m].migselect==1){
        if (migtask[m].state==1) {
                ntasks++; migtask = (taskinfo
*)realloc(migtask,ntasks*sizeof(taskinfo));
                taskinsert(m,ntasks,migtask);
                rows = migtask[m].rows;
          migtask[m].rows = migtask[m].xrow;
          rows -= migtask[m].rows;
          migtask[m].state = 2;          /* executed task */
          migtask[m].migselect = 0;       /* not selected to migrate */
          offset = migtask[m].offset;
          offset += migtask[m].rows*N;
          tmpoffset = migtask[m].tmpoffset;
          tmpoffset += migtask[m].rows*N;
          m++; ctask++;
          migtask[m].rows = rows;
          migtask[m].xrow = 0;
          migtask[m].offset = offset;
          migtask[m].tmpoffset = tmpoffset;
          migtask[m].state = 0;          /* a new task */
          migtask[m].migselect = 1;       /* select it to migrate */
          break;
        }
        break;
      }
      tmpoffset = migtask[m].tmpoffset;
      avetasks = nmigtasks/recnum;
      extra = nmigtasks%recnum;
      j = 0; mtype = WORK;
      for (i=0; i<numprocs; i++) if (located[i] == -1) {
        dest = i; j++;
        tasks = (j <= extra) ? avetasks+1 : avetasks;
```

```
      /*  printf("*** sending %d tasks from proc. %d, to proc.
%d\n",tasks,procid,dest);*/
        MPI_Send(&tasks, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
        if(tasks == 0) continue;
          for(k=0;k<tasks;k++){
            MPI_Send(&migtask[m].offset, 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD);
            rows = migtask[m].rows; rcvdrows -= rows; loadindex[procid] -=
1;
            MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            count = rows*N;
            offset = migtask[m].tmpoffset;
            MPI_Send(&a[offset], count, MPI_FLOAT, dest, mtype,
MPI_COMM_WORLD);
            migtask[m].migselect = 2; /* Task m has been migrated */
            m++;
          }
        }
        for(m=0;m<ntasks;m++) if (migtask[m].migselect == 2) break;
        rows = 0;
        for(i=0;i<ntasks;i++) {
          if (migtask[i].migselect == 2) {
            rows += migtask[i].rows;
            taskrem(i, ntasks, migtask); ntasks--; i--;
            migtask = (taskinfo *)realloc(migtask,ntasks*sizeof(taskinfo));
          }
        }
        count = rows*N;
        for(i=m;i<ntasks;i++) migtask[i].tmpoffset -= count;
          if(rows>0) {
            pac1 = a; pac2 = a;
            pac1 += tmpoffset; pac2 += tmpoffset+count;
            for(i=tmpoffset+count; i<tmprows*N; i++) { *pac1 = *pac2;
pac1++; pac2++; }
            pac1 = c; pac2 = c;
            pac1 += tmpoffset; pac2 += tmpoffset+count;
            for(i=tmpoffset+count; i<tmprows*N; i++) { *pac1 = *pac2;
pac1++; pac2++; }
            a = (float *)realloc(a,rcvdrows*N*sizeof(float));
            c = (float *)realloc(c,rcvdrows*N*sizeof(float));
          }
/*      printf("*** Sending from %d ,  DONE !!!\n",procid);*/
      }
      if ((sendnum > 0)&&(located[procid] == -1)){
        mtype = WORK;
        for(i=0;i<sendnum;i++) {
          MPI_Recv(&tasks, 1, MPI_INT, MPI_ANY_SOURCE, mtype,
MPI_COMM_WORLD, &status);
          source = status.MPI_SOURCE;
          if (tasks == 0) continue;
          for(k=0;k<tasks;k++){
            ntasks++;
            migtask = (taskinfo *)realloc(migtask,ntasks*sizeof(taskinfo));
            MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
            MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
            migtask[ntasks-1].offset = offset;
```

```
            migtask[ntasks-1].rows = rows;
            tmpoffset = rcvdrows*N; rcvdrows += rows;
            a = (float *)realloc(a,rcvdrows*N*sizeof(float));
            count = rows*N;

MPI_Recv(&a[tmpoffset],count,MPI_FLOAT,source,mtype,MPI_COMM_WORLD,&status)
;
            loadindex[procid] += 1;
            c = (float *)realloc(c,rcvdrows*N*sizeof(float));
            migtask[ntasks-1].tmpoffset = tmpoffset;
            migtask[ntasks-1].xrow = 0;
            migtask[ntasks-1].state = 0;         /* a new task */
            migtask[ntasks-1].migselect = 0;   /* not selected to migrate
*/
          }
        }
/*      printf("*** Receiving at %d ,  DONE !!!\n",procid);*/
      }
    }
      if (loadindex[procid] > 0 ){
        t1 = MPI_Wtime();
        DLB++;
        while(migtask[ctask].xrow != migtask[ctask].rows) {
          multiply(crow,a,b,c); crow++;
          migtask[ctask].xrow += 1;
        }
        loadindex[procid] -= 1;
        if (migtask[ctask].xrow == 1) migtask[ctask].state = 1; /* Task
under Execution */
        if (migtask[ctask].xrow == migtask[ctask].rows) {
          migtask[ctask].state = 2;  /* executed task */
          ctask++;
        }
        t2 = MPI_Wtime();
        tcomp += t2-t1;
      }
      if ((DLB>=LBF)||(loadindex[procid] == 0 )) DLB = 0;
      } while (!done(tmpload,procid,numprocs,loadindex));
      break;
      /*************************************************************/
      default: printf("The argument of TP is out of range\n");
  }

  mtype = RESULT;

  if (procid != 0) {
    MPI_Send(&tcomp, 1, MPI_DOUBLE, 0, mtype, MPI_COMM_WORLD);
    MPI_Send(&ntasks, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
    for(i=0;i<ntasks;i++) {
      offset = migtask[i].offset;
      rows = migtask[i].rows;
      tmpoffset = migtask[i].tmpoffset;
      MPI_Send(&offset, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
      MPI_Send(&c[tmpoffset], rows*N, MPI_FLOAT, 0, mtype, MPI_COMM_WORLD);
    }
  }
  else if(procid==0) {
```

```
    double totaltime;
    putilization = tcomp;

    rc = (float *)realloc(rc,N*N*sizeof(float));
    for(i=0;i<ntasks;i++) {
      rows = migtask[i].rows; count = rows*N;
      pac1 = c; pac2 = rc;
      pac1 += migtask[i].tmpoffset; pac2 += migtask[i].offset;
      for(j=0; j < count; j++) { *pac2 = *pac1; pac1++; pac2++; }
    }
  /*  printf("*** At the Root %d Tasks executed\n",ntasks);*/
    if(numprocs > 1) for (i=1; i<numprocs; i++) {
      MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      source = status.MPI_SOURCE;
      MPI_Recv(&tcomp, 1, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD,
&status);
      putilization += tcomp;
      MPI_Recv(&ntasks, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
 /*     printf("*** receiving info C, %d tasks from pro.
%d\n",ntasks,source);*/
      for(j=0;j<ntasks;j++){
        MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
        MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
        count = rows*N;
        MPI_Recv(&rc[offset], count, MPI_FLOAT, source, mtype,
MPI_COMM_WORLD,&status);
      }
    }
    endtime = MPI_Wtime();
    totaltime = endtime - starttime;
    putilization = putilization/numprocs; // average comp. time
    fptr = fopen("/home/mohammed/genetics/result","w");
    fprintf(fptr,"ExecTime : %f\n",totaltime);
    fprintf(fptr,"Putilize : %f\n",putilization/totaltime);
    fclose(fptr);
    printf("Total Time = %f\n",totaltime);

  }
  MPI_Barrier(MPI_COMM_WORLD);
/*  printf ("\n   DONE! At Process %d\n", procid);*/
  free(loadindex);
  free(procinfo);
  free(located);
  free(migtask);
  free(a);
  free(b);
  free(c);
  free(rc);
  MPI_Finalize();
  exit(0);
} /* of main */
```

Apéndice C

Código C para el algoritmo Genético descrito en capítulo 4

```
/*************************************************************************
* Este programa es una implementación en el lenguaje de programación C    *
* para el algoritmo genético discrito en el capítulo 4.                   *
* este algoritmo trata de explorar el espacio de búsqueda de los tres     *
* parámetros LBF, TSD1 y TSD2. el cuarto parámetro GRN está fijado para   *
* facilitar la convergencia del los otros tres parámetros.                *
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>
#include <math.h>

#define T 1000          /* el tamaño del problema que implementa el la
                           aplicación paralela */
#define S 12            /* número de procesadores*/
#define INDIVIDUALS 10  /* número de individuos, este número de individuos
                           fué eligido tras unos experimentos que
demuestran
                            que es el número más adecuado  */
#define GENES      4    /* número de genes, 1º LBF, 2º GRN, 3º TSD1 y 4º
TSD2,
                           aquí GRN aúnque está incluido pero no influye
                           porque está fijado*/
#define XOVER      0.6  /* esta probabilidad de cruce fue eligida tras unos
                           experimentos que demuestran que esta
probabilidad
                            es la más adecuada*/

#define N   3     /* número de ejecuciones de la aplicación paralela para
                     la evaluación del fitness */
#define PI 3.14159265358979323846264643

#define WORSTFITNESS 0.0
#define FALSE 0
#define TRUE  1
```

```
/*************************************************************************
* Una cadena de caracteres para llamar a la aplicación paralela        *
*            a la hora de evaluar el fitness                            *
*************************************************************************
**/
char param[88] = "mpirun -np 12 /home/mohammed/benchmarks/matgrn ";
char param1[60] = "mpirun -np 1 /home/mohammed/benchmarks/matgrn 0 0 2 1 1
1 1";

/*************************************************************************
*           una función que devuelve el tiempo del sistema              *
*************************************************************************/
double get_uptime() /* Read the system uptime from /proc/uptime.  */
{
  FILE* fp;
  double uptime, idle_time;

  fp = fopen ("/proc/uptime", "r");
  fscanf (fp, "%lf %lf\n", &uptime, &idle_time);
  fclose (fp);
  return(uptime);
}

/*************************************************************************
* La aplicación paralela devuelve su tiempo de ejecución y la utilización *
* media de los procesadores durante su ejecución y los escribe en un     *
* fichero se llama result. Esta función es para leer el tiempo de        *
* ejecución de la aplicación paralela del fichero result                 *
*************************************************************************/

float get_exectime(void)
{
  FILE* fp;
  float ExecTime;

  fp = fopen ("result", "r");
  fscanf(fp,"ExecTime : %f",&ExecTime);
  fclose (fp);
  return ExecTime;
}

/*************************************************************************
* Esta función es para leer el tiempo de ejecución y la utilización del  *
* fichero result escrito por la aplicación paralela                      *
*************************************************************************/
void get_time_util(float *extime, float *putilize)

{
  FILE* fp;

  fp = fopen ("result", "r");
  fscanf(fp,"ExecTime : %f\n",extime);
  fscanf(fp,"Putilize : %f",putilize);
  fclose (fp);
}
```

```
/*************************************************************************
* es possible durante la exploración que realiza el algoritmo genético    *
* que se repitan algunos individuos, para evitar las repeticiones         *
* posibles y ahorrar el tiempo se guardan todos los individuos            *
* explorados en un historial junto con su fitness. Esta función busca     *
* el individuo en el historial de individuos antes que evalúe su fitness  *
*************************************************************************/
int FindIndiv(int tot, int **indivs, int a, int b, int c, int d) {
  int i, found = -1;
  for(i=0;i<tot;i++)
    if
((indivs[i][0]==a)&&(indivs[i][1]==b)&&(indivs[i][2]==c)&&(indivs[i][3]==d)
) found = i;
  return(found);
}

/*************************************************************************
*          esta función muestra la población en la pantalla              *
*************************************************************************/
void printpob(float **CurrPob)
 {
  int i,j;

  printf("Poblacion Actual: \n");
  for(i=0; i<INDIVIDUALS; i++) {
    for(j=0; j<GENES; j++)
      printf("%f ",CurrPob[i][j]);
    printf("\n");
  }
 }

/*************************************************************************
* una función que devuelve un múmero entero aleatorio entre un dado       *
* mínimo y máximo                                                         *
*************************************************************************/
int my_random(int min, int max)
 {
      int fd = open("/dev/urandom", O_RDONLY);
      int nr;
      read(fd, &nr, sizeof(nr));
      close(fd);
      return abs(min + (double)(max - min) * nr / INT_MAX);
 }


/*************************************************************************
*     Inicializar los Individuos de la población inicial aleatoriamente   *
*************************************************************************/
void InitIndividuos(float **CurrPob, int lp, int tp){
  int indiv, gene, p, Max;
  float tmp;

  for(indiv=0; indiv<INDIVIDUALS; indiv++) {
    srand48((long)time(NULL)*indiv);
    for (gene=0;gene<GENES;gene++) {
      CurrPob[indiv][gene] = drand48();
      Max = S*S;
      p = (int)((double)Max*CurrPob[indiv][gene]);
```

```
      while ((p<1)||(p>Max)) {
      printf("%d\n", p);
      CurrPob[indiv][gene] = drand48();
      p = (int)((double)Max*CurrPob[indiv][gene]);
      }
    }
  }

  printpob(CurrPob);

}
```

```
/*************************************************************************
* Aquí se realiza la evaluación del fitness de los individuos. Los       *
* parámetros están codificados en número reales entre 0 y 1, entonces se *
* multiplica cada parámetro en su máximo y se coge el CEIL(), en este    *
* caso el máximo es igual para los tres parámetros y es igual a GRN.     *
* Entonces el valor que representa cada parámetro será                   *
* CEIL(PARAMETRO*MÁXIMO).                                                *
* Después de esta transformación de los parámetros se busca el individuo *
* en el historial de los individuos, por si haya sido evaluado antes.    *
* Si no lo encuentra, entonces se realiza una llamada al sistema para    *
* ejecutar la aplicación paralela introduciéndole los parámetros LBF,    *
* GRN, TSD1 y TSD2. Luego se recoge el tiempo de ejecución y la          *
* utilización media del archivo result, se multiplican estas dos         *
* cantidades para obtener el fitness del individuo.                      *
*************************************************************************/
void EvalPopulation(float sertime, float *IndivFitness, float *CumFitness,
float **CurrPob,int **indivs,float **speedup, int totindiv, int gener, int
LP, int SP, int TP){
  int indiv, curind;
  int gene, i, j, n, x;
  int LBF, GRN, TSD, TSD2, found;
  float *X;
  float TotFitness, sumx, sumu, utilize, extime, meu, sigma2, badfitness,
bestfitness;
  float CurrIndivFitness;
  int FitnessOrder[INDIVIDUALS];
  FILE *fp, *fput;

  X = (float *)malloc(N*sizeof(float));

  fput = fopen("tracing","a");
  TotFitness = 0.0;
  badfitness = S;
  bestfitness = WORSTFITNESS;

  for(indiv=0; indiv<INDIVIDUALS; indiv++){

    GRN = S*S;
    LBF = (int)ceil(CurrPob[indiv][0]*GRN);
    TSD = (int)ceil(CurrPob[indiv][2]*GRN);
    TSD2 = (int)ceil(CurrPob[indiv][3]*GRN);
    found = FindIndiv(totindiv,indivs,LBF,GRN,TSD,TSD2);
```

```
    if (found != -1) {
      meu = speedup[found][0];
      utilize = speedup[found][1];
      sigma2 = speedup[found][2];
      CurrIndivFitness = meu*utilize;
      if(CurrIndivFitness<badfitness) badfitness = CurrIndivFitness;
      if(CurrIndivFitness>bestfitness) bestfitness = CurrIndivFitness;
      fprintf(fput,"%d\t%d\t%d\t%d\t%4.2f\t%4.2f\t%4.2f\n", LBF, GRN, TSD,
TSD2, meu, utilize, sigma2);
    }
    else {
      i=47;
      for(j=i;j<88;j++) param[j] = 0;

      fp = fopen("individuo","w");
      fprintf(fp,"%d %d %d ", LP, SP, TP);
      for(gene=0; gene<GENES; gene++)
      fprintf(fp,"%f ",CurrPob[indiv][gene]);
      fclose(fp);

      printf("Execute as:\n");
      fp = fopen("individuo","r");
      for(j=i;j<88;j++) param[j] = fgetc(fp);
      fclose(fp);

      printf("%s\n",param);
      sumx = 0.0;
      sumu = 0.0;
      for(x=0;x<N;x++) {
        n = system(param);
        n = system("lamclean");
      get_time_util(&extime, &utilize);
        X[x] = extime;
      X[x] = sertime/X[x];
        sumx += X[x];
      sumu += utilize;
      }
      meu = sumx/N;
      sumx = 0.0;
      for(x=0;x<N;x++) {
        sumx = (X[x]-meu)*(X[x]-meu);
      }
      sigma2 = sumx/N;
      utilize = sumu/N;
      CurrIndivFitness = meu*utilize;
      if(CurrIndivFitness<badfitness) badfitness = CurrIndivFitness;
      if(CurrIndivFitness>bestfitness) bestfitness = CurrIndivFitness;
      fprintf(fput,"%d\t%d\t%d\t%d\t%4.2f\t%4.2f\t%4.2f\n", LBF, GRN, TSD,
TSD2, meu, utilize, sigma2);
    }

    curind = (gener*INDIVIDUALS) + indiv;
    speedup[curind][0] = meu;
    speedup[curind][1] = utilize;
    speedup[curind][2] = sigma2;
    indivs[curind][0] = LBF;
    indivs[curind][1] = GRN;
```

```
    indivs[curind][2] = TSD;
    indivs[curind][3] = TSD2;

    IndivFitness[indiv] = CurrIndivFitness;
    CumFitness[indiv] = CurrIndivFitness;
    TotFitness += CurrIndivFitness;
    printf("Fitness of Indiv. %d is %f\n",indiv,CurrIndivFitness);
  }

  meu = TotFitness/INDIVIDUALS;
  fprintf(fput,"Max: %4.2f, Mean: %4.2f, Min:
%4.2f\n",bestfitness,meu,badfitness);

  fclose(fput);

  for(i=0; i<INDIVIDUALS; i++) {
    indiv = 0;
    for(j=1;j<INDIVIDUALS; j++) if(CumFitness[j]>CumFitness[indiv]) indiv =
j;
    FitnessOrder[i] = indiv;
    CumFitness[indiv] = WORSTFITNESS;
  }

  TotFitness = 0.0;
  for(i=0; i<INDIVIDUALS; i++) {
    indiv = FitnessOrder[i];
    CumFitness[indiv] = (INDIVIDUALS-i)*(IndivFitness[indiv]);
    TotFitness += CumFitness[indiv];
  }

  CumFitness[0] = 100*(CumFitness[0]/TotFitness);
  for(i=1; i<INDIVIDUALS; i++)
    CumFitness[i] = CumFitness[i-1] + (100*CumFitness[i]/TotFitness);
  printf("The Cummulative Fitness is %f\n",CumFitness[INDIVIDUALS-1]);

  free(X);
}

/*********************************************************************
*                       seleccionar el major individuo                *
*********************************************************************/
int SelectBest(float *BetterIndiv, float *BetterFitness, float
*IndivFitness, float **CurrPob){
  int betterind, i;

  betterind = 0;

  for(i=1; i<INDIVIDUALS; i++)
    if (IndivFitness[i] > IndivFitness[betterind]) betterind = i;

  if (IndivFitness[betterind] > *BetterFitness) {
    *BetterFitness = IndivFitness[betterind];
    printf("***** Better Fitness Found: ");

    for(i=0;i<GENES;i++) {
      BetterIndiv[i] = CurrPob[betterind][i];
      printf("%f ",BetterIndiv[i]);
    }
```

```
    printf(" ====>>> Con Fitness : %f\n",*BetterFitness);
  }

  IndivFitness[betterind] = WORSTFITNESS;

  return(betterind);
}

/*************************************************************************
* realiza selección, cruce y mutación para generar la nueva población →  *
* Explotación                                                            *
* 1. la selcción se raliza según el método de la ruleta                  *
* 2. se utiliza un cruce con probabilidad 2 padres → 2 hijos, la         *
* probabilidad  de cruzar dos individuos está fijada en 0.6 . si dos     *
* individuos van a cruzar, se genera un punto aleatorio y los individuos *
* se cruzan a partir de este punto.                                      *
* 2. una vez están seleccionados los cruzados los nuevos individuos,     *
* pasan a la fase de la mutación y se mutan según el operador de mutación *
* descrito en el capítulo 4 de la tesis                                  *
*************************************************************************/
void FindNextPop(int gener, int numgen, float **CurrPob, float
*BetterIndiv, float *BetterFitness, float *IndivFitness, float *CumFitness)
{
  float **NextPob;
  int indiv, sindiv, Max;
  int gene;
  int XoverPoint;
  int CruceIndiv, xover;
  int MutaGene, mutation;
  int elmejor;
  float rw, temp, gss, bord;

  NextPob = (float**)malloc(INDIVIDUALS*sizeof(float*));
  for(indiv=0; indiv<INDIVIDUALS; indiv++)
    NextPob[indiv] = (float *)malloc(GENES*sizeof(float));

  for(indiv=0; indiv<INDIVIDUALS; indiv++)
    for(gene=0; gene<GENES; gene++)
      NextPob[indiv][gene] = CurrPob[indiv][gene];

  elmejor = SelectBest(BetterIndiv, BetterFitness, IndivFitness, CurrPob);

                /* S E L E C T I O N  (RW)*/
  printf("Doing Selection\n");
  for(indiv=0; indiv<INDIVIDUALS; indiv++) {
    rw = my_random(0, 100);
    sindiv = 0;
    while (CumFitness[sindiv]<rw) sindiv++;
    for(gene=0; gene<GENES; gene++) NextPob[indiv][gene] =
CurrPob[sindiv][gene];
  }

  for(indiv=0; indiv<INDIVIDUALS; indiv+=2){
    CruceIndiv = my_random(1,11);
    xover = (int)(10*XOVER);
    if(CruceIndiv <= xover) {
      XoverPoint  =  1+my_random(0, GENES-1);
```

```
      printf("Doing Xover between Indivs. %d, %d, from the gene %d\n",
indiv, indiv+1, XoverPoint);
      for(gene=XoverPoint; gene<GENES; gene++) {
      temp = NextPob[indiv][gene];
      NextPob[indiv][gene] = NextPob[indiv+1][gene];
      NextPob[indiv+1][gene] = temp;
      }
    }
  }

  printpob(CurrPob);
  for(indiv=0; indiv<INDIVIDUALS; indiv++){
    /* Mutation */
    printf("Doing Mutation in Indiv. %d\n", indiv);
    for(gene=0; gene<GENES; gene++){
     /* MutaGene = my_random(0,10);
      mutation = (int)(10*MUTATION);
      if(MutaGene < 10) {*/
          temp = NextPob[indiv][gene];
        Max = S*S;
        if ((gene == 3)||(gene == 1)) bord = 1.0/S+0.0;
        else bord = 1.0/Max+0.0;
        gss = fabs(temp-bord);
        if ((gss > (1-bord))||(gss > bord)) {
          if (temp > bord) temp = temp - (gss/2);
          else temp = temp + (gss/2);
        }
          else {
          if (temp > bord) temp = temp - (gss+(gss/2));
          else temp = temp + (gss+(gss/2));
        }
              NextPob[indiv][gene] = temp;
      //}
    }
  }
           /* E L I T I S M O */
  for(gene=0; gene<GENES; gene++) CurrPob[0][gene] = BetterIndiv[gene];

  for(indiv=0; indiv<INDIVIDUALS; indiv++)
    for(gene=0; gene<GENES; gene++)
      CurrPob[indiv][gene] = NextPob[indiv][gene];

  printpob(CurrPob);

  for(indiv=0; indiv<INDIVIDUALS; indiv++) free(NextPob[indiv]);
  free(NextPob);
}


int main(int argc, char *argv[]){

  float **CurrPob;
  float *IndivFitness,
        *CumFitness;
  float *BetterIndiv, **speedup, *mins, *maxs, *sums, *minu, *maxu, *sumu;
  float BetterFitness = WORSTFITNESS;
  float sumx, sertime;
  float bettertemp, sigma2, meu, utilize;
```

```
 double starttime, endtime;

 int LBF, GRN, TSD, TSD2, LP, SP, TP;

 int indiv, **indivs;
 int x, n, i, totindiv,iter;
 int gener = 0;
 int numgen;
 FILE *fp;

 CurrPob = (float**)malloc(INDIVIDUALS*sizeof(float*));

 IndivFitness = (float*)malloc(INDIVIDUALS*sizeof(float));
 CumFitness = (float*)malloc(INDIVIDUALS*sizeof(float));
 BetterIndiv = (float*)malloc(GENES*sizeof(float));

 for(indiv=0; indiv<INDIVIDUALS; indiv++)
   CurrPob[indiv] = (float *)malloc(GENES*sizeof(float));

/* n = system("lamboot hosts");*/

 numgen = atoi(argv[1]);
 LP = atoi(argv[2]);
 SP = atoi(argv[3]);
 TP = atoi(argv[4]);

 InitIndividuos(CurrPob,LP,TP);

 fp = fopen("tracing","a");
 fprintf(fp,"Execution Results For: %s%d %d %d\n",param,LP,SP,TP);
 fprintf(fp,"LBF GRN   TSD   TDS2  SpeedUp     Utilize     SigmaSqr\n");
 fclose(fp);

 fp = fopen("resultados","a");
 fprintf(fp,"Execution Results For: %s%d %d %d\n",param,LP,SP,TP);
 fprintf(fp,"LBF GRN   TSD   TDS2  SpeedUp     Utilize     SigmaSqr\n");
 fclose(fp);

 totindiv = INDIVIDUALS*numgen;

 indivs = (int **)malloc(totindiv*sizeof(int*));
 for(i=0;i<totindiv;i++)
   indivs[i] = (int *)calloc(GENES,sizeof(int));

 speedup = (float **)malloc(totindiv*sizeof(float*));
 for(i=0;i<totindiv;i++)
   speedup[i] = (float *)malloc(3*sizeof(float));

 mins = (float *)malloc(numgen*sizeof(float));
 maxs = (float *)malloc(numgen*sizeof(float));
 sums = (float *)malloc(numgen*sizeof(float));

 minu = (float *)malloc(numgen*sizeof(float));
 maxu = (float *)malloc(numgen*sizeof(float));
 sumu = (float *)malloc(numgen*sizeof(float));

 for(i=0;i<totindiv;i++) indivs[i][0] = -1;
```

```
  starttime = get_uptime();

  printf("get the serial execution time by executing: \n");
  printf("%s\n",param1);
  sumx = 0.0;

  for(x=0;x<N;x++) {
    n = system(param1);
    n = system("lamclean");
    sumx += get_exectime();
  }
  sertime = sumx/N;

  bettertemp = BetterFitness;
  iter = 0;

                          /* realizar la búsqueda */
  while (gener<numgen){
    EvalPopulation(sertime,IndivFitness,CumFitness,CurrPob,indivs,
                   speedup,totindiv,gener,LP,SP,TP);
    FindNextPop(gener, numgen, CurrPob, BetterIndiv, &BetterFitness,
IndivFitness, CumFitness);
    gener++;
    if (BetterFitness>bettertemp) {
      bettertemp = BetterFitness;
      iter = gener;
    }
  }

  endtime = get_uptime();

  GRN = S*S;
  LBF = (int)ceil(BetterIndiv[0]*GRN);
  TSD = (int)ceil(BetterIndiv[2]*GRN);
  TSD2 = (int)ceil(BetterIndiv[3]*GRN);
  indiv = FindIndiv(totindiv,indivs,LBF,GRN,TSD,TSD2);
  meu = speedup[indiv][0];
  utilize = speedup[indiv][1];
  sigma2 = speedup[indiv][2];

  printf("Number of generations was: %d \n", gener);
  printf("***** Mejor Individuo Encontrado: ");
  printf("%d %d %d %d\n", LBF, GRN, TSD, TSD2);
  printf(" ====>>> Con Fitness : %f\n",BetterFitness);

                           /*Escribir resultados*/
  fp = fopen("resultados","a");
  fprintf(fp,"%d\t%d\t%d\t%d\t%4.2f\t%4.2f\n", LBF, GRN, TSD, TSD2, meu,
utilize, sigma2);
  fprintf(fp,"Number of generations was: %d ,,, The Best Indiv. Found in
%d:\n", gener,iter);

  for(i=0;i<gener;i++) {
    x = i*INDIVIDUALS;
    mins[i] = speedup[x][0];
    maxs[i] = speedup[x][0];
    minu[i] = speedup[x][1];
    maxu[i] = speedup[x][1];
```

```
    sums[i] = speedup[x][0];
    sumu[i] = speedup[x][1];
    for (n=x+1;n<x+INDIVIDUALS;n++) {
      if(speedup[n][0] < mins[i]) mins[i] = speedup[n][0];
      if(speedup[n][0] > maxs[i]) maxs[i] = speedup[n][0];
      if(speedup[n][1] < minu[i]) minu[i] = speedup[n][1];
      if(speedup[n][1] > maxu[i]) maxu[i] = speedup[n][1];
      sums[i] += speedup[n][0];
      sumu[i] += speedup[n][1];
    }
  }

  fprintf(fp,"minsp%d%d%d = [",LP,SP,TP);
  for(i=0;i<gener;i++) fprintf(fp,"%4.2f ",mins[i]);
  fprintf(fp,"]\n");

  fprintf(fp,"maxsp%d%d%d = [",LP,SP,TP);
  for(i=0;i<gener;i++) fprintf(fp,"%4.2f ",maxs[i]);
  fprintf(fp,"]\n");

  fprintf(fp,"meansp%d%d%d = [",LP,SP,TP);
  for(i=0;i<gener;i++) fprintf(fp,"%4.2f ",sums[i]/INDIVIDUALS);
  fprintf(fp,"]\n");

  fprintf(fp,"minut%d%d%d = [",LP,SP,TP);
  for(i=0;i<gener;i++) fprintf(fp,"%4.2f ",minu[i]);
  fprintf(fp,"]\n");

  fprintf(fp,"maxut%d%d%d = [",LP,SP,TP);
  for(i=0;i<gener;i++) fprintf(fp,"%4.2f ",maxu[i]);
  fprintf(fp,"]\n");

  fprintf(fp,"meanut%d%d%d = [",LP,SP,TP);
  for(i=0;i<gener;i++) fprintf(fp,"%4.2f ",sumu[i]/INDIVIDUALS);
  fprintf(fp,"]\n");

  fprintf(fp, "Total Algorithm Time = %4.2f\n", endtime-starttime);

  fclose(fp);

  printpob(CurrPob);

/*  n = system("lamhalt");
  n = system("wipe -v hosts");*/

  for(i=0; i<INDIVIDUALS; i++) free(CurrPob[i]);
  for(i=0; i<totindiv; i++) free(indivs[i]);
  for(i=0; i<totindiv; i++) free(speedup[i]);

  free(CurrPob);
  free(indivs);
  free(IndivFitness);
  free(CumFitness);
  free(BetterIndiv);
  free(speedup);
  free(mins);
  free(maxs);
  free(sums);
```

```
  free(minu);
  free(maxu);
  free(sumu);

}
```

Bibliografía

[AHM91] I. Ahmad and A. Ghafoor, “Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems”, IEEE Transactions on Software Engineering,vol. 17, no. 10, pp. 987-1004, October 1991.

[ALB02] ALBA, A.; TOMASINI, M.:”Parallelism and Evolutionary Algorithms”. IEEE Trans. on Evolutionary Computation, Vol. 6, No. 5, pp.443-462. Octubre, 2002.

[ALB96] Van Albada, G.D., Clinckemaillie, J. “Dynamite-blasting Obstacles to Parallel Cluster Computing”, Technical Report, Department of Computer Science, University of Amsterdam, The Netherlands, 1996.

[ALD04] Aldasht, M.; Ortega, J.; Puntonet, C.G.; Díaz, A.F.:”A Genetic Exploration of Dynamic Load Balancing Algorithms”. IEEE Conference on Evolutionary Computation, Portland, Oregon, 2004.

[AMD67] Amdahl, G.M., “Validity of single-processor approach to achieving large-scale computing capability”, Proceedings of AFIPS Conference, Reston, VA. 1967.

[AMO01] M. Amor, F. Argüello, J. López3, O. Plata, and E. L. Zapata, “A Data-Parallel Formulation for Divide and Conquer Algorithms”, Computer Journal, Volume 44, Issue 4, pp. 303-320, April 2001.

[ANA03] Aris Anagnostopoulos, Adam Kirsch, Eli Upfal, “Stability and Efficiency of a Random Local Load Balancing Protocol”, 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03, pp. 472-481), October 2003.

[AND91] J. Andrews and C. Polychronopoulos, “An Analytical Approach to Performance/Cost Modelling of Parallel Computers”, Journal of Parallel and Distributed Computing, no. 12, pp. 343-356, 1991.

[AND95] Anderson, T.E.; Culler, D.E.; Patterson, D.A.:"A case for NOW (Networks of Workstations)". IEEE Micro, pp.54-64. Febrero, 1995.

[ANT04] ANTONIS, K.; GAROFALAKIS, J.; MOURTOS, I. ; SPIRAKIS, P. : “A hierarchical adaptive distributed algorithm for load balancing”. J. Parallel Distrib. Comput., 64, pp.151-162, 2004.

[ANT98] Antonis, K.; Garofalajis, J.; Spirakis, P.: “A comparative symmetrical transfer policy for load sharing “. Proceedings of the 1998 Int. Euro-Par Conference, Springer, 1998.

[ARN89] E. A. Arnould, F.J.Bitz, E. C. Cooper,H.T.Kung, R. D. Sansom, and P.A.Steenkiste, “The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers”, Third Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pp. 205-216, 1989.

[BAA88] Baase, Sara, “Computer Algorithms, Introduction to Design and Analysis, second edition” Addison-wesley, 1988.

[BAC97] Bäck, T.; Hamel, U.; Schwefel, H.-P.:”Evolutionary computation: Comments on the History and Current State”. IEEE Trans. on Evolutionary Computation, Vol.1, No.1, pp.3-17. Abril, 1997.

[BAR04] BARKER, K.; CHERNIKOV, A.; CHRISOCHOIDES, N.; PINGALI, K.:”A Load Balancing Framework for Adaptive and Asynchronous Applications”. IEEE Trans. on Parallel and Distributed Systems, Vol.15, No. 2, pp.183-192. Febrero, 2004.

[BEN91] K. BenMohammed, “An Evaluation of Load Balancing Algorithms for Distributed Systems”, PhD thesis, School of Computer Studies, University of Leeds, September 1991.

[BEO04] (http://www.beowulf.org/), Proyecto Original de Computación de Altas prestaciones con Linux.

[BER00] Berzins, M.:”A new metric for dynamic load balancing”. Applied Mathematical Modeling, Vol.25, pp.141-151, 2000.

[BEV70] Beveridge , S.G and Schetchter . S, “Optimization : Theory and Practice” McGraw-Hill Book Company, 1970.

[BIS88] R. Bisiani and A. Forin, “Multilanguage Parallel Programming of Heterogeneous Machines”, IEEE Transactions on Computers,vol. 37, no. 8, pp. 930-945, August 1988.

[BOI90] J. E. BOILLAT, “Load balancing and Poisson equation in a graph”, Concurrency: Practice and Experience, 2, pp. 289-313, 1990.

[BRA01] Branke, J.:”Evolutionary Optimization in Dinamic Environments”. Kluwer, 2001.

[BRA01] T. D. Braun ET AL., “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems”, J. of Parallel and Distributed Computing, Vol. 61, No. 6, pp. 810-837, June 2001.

[BRA99] Branke, J.:”Memory enhanced evolutionary algorithms for changing optimization problems”. IEEE Congress on Evolutionary Computation, CEC99, Vol.3, pp.1875-1882, 1999.

[BUT91] M. Butler,T.-Y.Yeh, Y.Patt, M. Alsup, H. Scales, and M. Shebanow, “Single Instruction Stream Par-allelism is Greater than Two”, International Symposium on Computer Architecture, pp. 276-286, 1991.

[BUY99] Rajkumar Buyya, 1999, “High Performance Cluster Computing volume 1, Architectures and systems”, Prentice Hall PTR.

[CAR92] N. Carriero, D. Gelernter,and T.G.Mattson, “Linda in Heterogeneous Computing Environments”, Workshop on Heterogeneous Processing,International Parallel Processing Symposium,pp. 43-46, 1992.

[CAS88] T.L. Casavant and J. G. Kuhl, “A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems”, IEEE Transactions on Software Engineering, vol. 14, no. 2, pp. 141-154, February 1988.

[CAS94] J. Casas, et. al.. “Adaptive Load Migration Systems for PVM ” Proc. of Supercomputing ‘94, pp. 390-399, IEEE Computer Society Press, 1994.

[CER97] M. Cermele, M. Colajanni and G.Necci, “Dynamic load balancing of distributed SPMD computations with explicit message passing”, Proc. of IEEE Heterogeneous Computing Workshop, Geneva, pp. 2-13, Apr. 1997.

[CHA94] S. Chakrabarti, A. Ranade, and K. Yelick, “Randomized Load Balancing for Tree-structured Computation”, In Proceeding of the IEEE Scalable High Performance Computing, pages 666-673, 1994.

[CHA97] Huah Yong Chan, Claude Daval-Frerot “Dynamic load balancing using MPI Implementation on ATM” Fifteenth IASTED International Conference Applied Informatics, Feb. 1997.

[CHE02] Hongtu Chen, Muthucumaru Maheswaran, “Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems”, International Parallel and Distributed Processing Symposium , Fort Lauderdale, Florida, pp. 15-19, April 2002.

[CHE93] S. Chen, M. M. Eshaghian, A. Khokhar,and M. E. Shaaban, “A Selection Theory and Methodology for Heterogeneous Supercomputing”, Workshop on Heterogeneous Processing,International Parallel Processing Symposium,pp. 15-22, 1993.

[CHE94] Lin Chengjiang and Li Sanli, “Strategy and Simulation of Adaptive RID for Distributed Dynamic Load Balancing in Parallel Systems”, IEEE International Symposium on Parallel Architectures, Algorithms and Networks, pp. 406-412, 1994.

[CHO90] S. Chowdhury, “The Greedy Load Sharing Algorithm”, Journal of Parallel and Distributed Computing, no. 9, pp. 93-99, May 1990.

[COM02] Nicholas Comino and V. Lakshmi Narasimhan, “A Novel Data Distribution Technique for Host-Client Type Parallel Applications”, IEEE Transactions On Parallel And Distributed Systems, Vol. 13, No. 2, pp. 97-110, February 2002.

[COR00] Corne, D.W.; Knowles, J.D.; Oates, M.J.:”The Pareto envelope-based selection algorithm for multiobjective optimization”. Proc. of the Parallel Problem Solving from Natures VI Conference, pp.839-848, 2000.

[COR90] Thomas H. Cormen, Charles E. Leiseron, and Ronald L. Rivest, “Introduction to Algorithms”, 1990.

[COR99] Corrêa, R.C.; Ferreira, A.; Rebreyend, P.:”Scheduling multiprocessor tasks with genetic algorithms”. IEEE Tansactions on Parallel and Distributed Systems, Vol. 10, No.8, pp.825-837. Agosto, 1999.

[COR99a] A. Corradi, L. Leonardi, F. Zambonelli, “Diffusive Load Balancing Policies for Dynamic Applications”, IEEE Concurrency, vol. 7, no. 1, 22-31, 1999.

[COR99b] A. Cortés, A. Ripoll, M.A. Senar, P. Pons, E. Luque, “On the Performance of Nearest-Neighbors Load Balancing Algorithms in Parallel Systems”, Seventh Euromicro Workshop on Parallel and Distributed Processing, pp. 170-177, February 1999.

[CUL99] David E. Culler and Jaswinder Pal Dingh, with Anoop Gupta, “Parallel Computer Architecture a Hardware/Software Approach”, Morgan Kaufmann Publishers, Inc. USA, 1999.

[CYB89] CYBENKO, G.:”Dynamic Load Balancing for Distributed Memory Multiprocessor”. J. Parallel and Distributed Computing, Vol.19, pp.279-301, 1989.

[DAN97] Dandamudi, S., Piotrowski, A. ,“A Comparative Study of Load Sharing on Networks of Workstations”, Proceedings of the International Conference on Parallel and Distributed Computing Systems, New Orleans.

[DAV91] L. Davis, “Handbook of Genetic Algorithms”, van Nostrand Reinhold, New York, 1991, October 1997.

[DEJ75] DeJong, K.A.:”An analysis of the behaviour of a class of genetic adaptive systems”. Tesis Doctoral, Universidad de Michigan, Ann Arbor, 1975.

[DIE93] H. G. Dietz, W.E.Cohen, and B. K. Grant, “Would You Run it Here... or There? (AHS: Automatic Heterogeneous Supercomputing)”, International Conference on Parallel Processing, Volume II:Software,pp. 217-221, 1993.

[DON93] Dongarra J., A. Geist, R. Manchek, V. Sunderam, “Integrated pvm framework supports heterogeneous network computing”, Computers in Physics, (7)2, pp. 166-175, April 1993.

[DOS96] Luis Paulo Peixoto Dos Santos, “Load Distribution: A Survey”, Technical Report, Universidade do Minho, Departamento do Informática Oct. 1996.

[DRI95] DRIESSCHE, R. van; ROOSE, D.:”An Improved Spectral Bisection Algorithm and its Application to Dynamic Load Balancing”. Parallel Computation, Vol. 21, pp.29-48, 1995.

[EAG86] D. L. Eager, E. D. Lazowska, and J. Zahorjan, “Adaptive Load Sharing in Homogeneous Distributed Environments”, IEEE Transactions on Software Engineering, Vol. 12 no. 5, pp. 662-675, May 1986.

[ELS00] Robert Elsasser, Burkhard Monien and Robert Preis “Diffusive Load Balancing Schemes on Heterogeneous Networks”, ACM Symposium on Parallel Algorithms and Architectures, pp. 30-38, 2000.

[EVA93] D. Evans and W. Butt. “Dynamic Load Balancing Using Task-Transfer Probabilities.” Parallel Comp., 19:897–916, 1993.

[FER99] Fernández, L.; García, J.M.:”Una aproximación a las Redes de Altas Prestaciones”. X Jornadas de Paralelismo, pp.225-233, 1999.

[FOG00] Fogel, D.B.:”What is evolutionary computation?. IEEE Spectrum, pp.26-32. Febrero, 2000.

[FOG62] Fogel, L.J.:”Autonomous Automata”. Ind. Research, Vol. 4, pp.14-19,1962.

[FOG66] Fogel, L.J.; Owens, A.J.; Walsh, M.J.:"Artificial Intelligence through Simulated Evolution". Wiley, 1966.

[FOG92] Fogel, D.B.:"Evolving Artificial Intelligence". Ph.D. Dissertation. Universidad de California, San Diego, 1992.

[FOS95] Ian Foster, An Online Publishing Project of Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation, Designing and Building Parallel Programs v1.3 / dbpp@mcs.anl.gov, 1995.

[FRE89] R. F. Freund, "Optimal Selection Theory for Superconcurrency", Supercomputing '89, pp. 699-703, 1989.

[GAR79] M. R. Garey and D. S. Johnson, "Computers and Intractability", W. H.Freeman and Company, New York, 1979.

[GHO96a] Bhaskar Ghosh and S. Muthukrishnan, "Dynamic load balancing by random matchings", Journal of Computer and System Sciences, Vol. 53 I. 3, pp. 357-370, December 1996.

[GHO96b] Bhaskar Ghosh, S. Muthukrishnan, Martin H. Schultz, "First and Second order diffusive methods for rapid, coarse, distributed load balancing", SPAA, pp. 72-81, Italy 1996.

[GIL02] GIL, C.; ORTEGA, J.; MONTOYA, M.G.; R. BAÑOS: "A Mixed Heuristic for Circuit Partitioning". International Journal on Computational Optimization and Applications, 23, pp.321-340, 2002.

[GIL98] GIL, C.; ORTEGA, J.; DÍAZ, A.F.; MONTOYA, M.G.: " Annealing-based heuristics and genetic algorithms for circuit partitioning in parallel test generation". Future Generation Computer Systems Journal (FGCS), Elsevier, Vol. 14, pp.439-451, 1998.

[GOL89] Goldberg, D.E., "Genetic Algorithms in Search, Optimization, and Machine Learning", Addison-Wesley: Reading, MA, 1989

[GON94] González, C.A.; Wainwright, R.L.:"Dynamic Scheduling of Computer Tasks using Genetic Algorithms". Proc. First Conf. Evolutionary Computation, IEEE World Congress Computational Intelligence, Vol.2, pp. 829-833, 1994.

[GOR03] J.M.Górriz, C.G.Puntonet, M.Aldasht, M.Salmerón, M.Damas: “Neural AR Prediction model using exogenous series implemented in parallel programming languages”. Jornadas de Paralelismo, Leganés, Madrid, 2003. ISBN: 84-89315-34-5.

[GOR04] J.M.Górriz, C.G.Puntonet, M.Salmerón, J.Ortega, M.Aldasht: "Time series forecasting based on parallel neural network", EIS-2004, 29-febrero al 2 de Marzo, Madeira-Portugal. 2004. (IEEE)

[GRI92] A. S. Grimshaw, “Meta-Systems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems”, Workshop on Heterogeneous Processing,International Parallel Processing Symposium, pp. 54-59, 1992.

[GUS88] Gustafson, J.L., Reevaluating Amdahl's Law, CACM, 31(5). pp. 532-533, 1988.

[HAC87] Hac, A.; Jin, X.:”Dynamic load balancing in a distributed system using a decentralized algorithm”. Proc. IEEE International Conference on Distributed Computing Systems, pp.170-177, 1987.

[HAD93] E. Haddad, “Load Distribution Optimization in Heterogeneous Multiple Processor Systems”, Workshop on Heterogeneous Processing, International Parallel Processing Symposium, pp. 42-47, 1993.

[HAD97] Hadad, B.S.; Eick, C.F.:”Supporting polyploidy in genetic algorithms using dominance vectors”. 6th Int. Conf. In Evolutionary Programming, LNCS 1213, pp.223-234, 1997.

[HAM94] B. Hamidzadeh and D. J. Lilja, “Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing, Volume II: Software”, International Conference on Parallel Processing,pp. 39-46, 1994.

[HAM96] Babak Hamidzadeh and David J. Lilja, “Dynamic Scheduling Strategies for Shared-Memory Multiprocessors”, International Conference on Distributed Computing Systems, pp. 208-215, May 1996.

[HAR96] Mor Harchol-Balter and Allen B. Downey, “Exploiting process lifetime distributions for dynamic load balancing”, Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Philadelphia, Pennsylvania, United States , pp.13 - 24 ,1996

[HEI94] A. Heirich and S. Taylor, “AParabolic Load Balancing Algorithm”, Technical Report, Cal-tech Computer Science Department, Caltech-CS-TR-94-13, 1994.

[HEI95] HEIRICH, A.; TAYLOR, S.:”A Parabolic Load Balancing Algorithm”. Proc. 24th Int. Conf. on Parallel Programming, Vol.3, pp.192-202, 1995.

[HOL75] Holland, J.H.:”Adaptation in Natural and Artificial Systems”. University of Michigan Press., 1975.

[HOL86] Holland, J.H.; Holyoak, K.J.; Nisbett, R.E.; Thagard, P.R.:”Induction: Processes of Inference, Learning, and Discovery”. MIT Press, Cambridge, Mass., 1986.

[HOR93] HORTON, G.:”A Multi-Level Difussion Method for Dynamic Load Balancing”. Parallel Computing, Vol.19, pp.209-218, 1993.

[HOS89] H. Hosseini, B. Litow, M. Malkawi, J. McPherson and K. Vairavan “System Theory Modeling and Performance Analysis of a Distributed Load Balancing Algorithm”, Proc. of the 32nd IEEE Midwest Symposium on Circuits and Systems, pp. 648-652, 1989.

[HOS90] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, “Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm”, Journal of Parallel and Distributed Computing, Vol. 10, pp. 160-166, 1990.

[HOV03] Hovey, L.; Volper, D.E.; Oh, J.C.:”Evolution of optimal server clusters for dynamic load-balancing systems”. IEEE Conference on Evolutionary Computation, pp.528-535, 2003.

[HU82] Hu, T.C.:”Combinatorial algorithms”. Addison Wesley, 1982.

[HUM92] S. F. Hummel, E. Schonberg, and L. E. Flynn, “Factoring: A Method for Scheduling Parallel Loops”, Communications of the ACM,vol. 35, no. 8, pp. 90-101, August 1992.

[IQB93] M. A. Iqbal, “Partitioning Problems in Heterogeneous Computer Systems”, Workshop on Heterogeneous Processing, International Parallel Processing Symposium, pp. 23-28, 1993.

[KHO93] A. A. Khokhar,V.K.Prasanna, M. E. Shaaban, and C.-L. Wang, “Heterogeneous Computing: Chal-lenges and Opportunities”, Computer,vol. 26, no. 6, pp. 18-27, June 1993.

[KID94] Kidwell, M.D.; Cook, D.J.:”Genetic Algorithm for Dynamic Task Sceduling”. Proc. 13th IEEE Int. Phoenix Conf. Computers and Communication, pp.61-67, 1994.

[KLE94] Klein; Tavangarian; Hipper; Koch, “A New Concurrent Network Architecture (CNA) for an Efficient Parallel Computing in Workstation Clusters”, Workshop Parallele Datenverarbeitung im Verbund von Hochleistungs-Workstations, Fachberichte Informatik, Universitat Koblenz, sep 1994.

[KOH95] KOHRONG, G.:”Dynamic Load Balancing for Parallelized Particle Simulations on MIMD Computers”. Parallel Computing, Vol.21, pp.683-693, 1995.

[KOR03] Jens-Christian Korth, “Diffusive Load Balancing” Advanced seminar “Load Balancing for Massive Parallel Systems” , Department for Simulation of Large Systems, University of Stuttgart, June, 2003, http://www.informatik.uni-stuttgart.de/ipvr/sgs/lehre/seminare/DLD/03_reports.pdf.

[KOZ92] Koza, J.R.:”Genetic Programming: On the Programming of Computers by Means of Natural Selection”. MIT Press. Cambridge, Mass., 1992.

[KOZ92] Koza, J.R.:”Genetic Programming: On the Programming of Computers by Means of Natural Selection”. MIT Press, 1992.

[KUC90] H. Kuchen, A. Wagener: “Comparison of Dynamic Load Balancing Strategies”, AIB 90-5, RWTH Aachen, 1990.

[KUC96] KUCK, D.J.:”High Performance Computing. Challenges for Future Systems”. Oxford University Press, 1996.

[KUM88] M. Kumar, “Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications”, IEEE Transactions on Computers, vol. 37, no. 9, pp. 1088-1098, September 1988.

[KUM94a] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, “Introduction To Parallel Computing, Design and Analysis of Algorithms”, by the Benjamin/Cummings Publishing Company, 1994.

[KUM94b] J. Mohan Kumar, L.M. Patnaik, and A. Das, “Load Balancing Algorithms for an Extended Hypercube”, Computer Digital Technology, 141(5), pp. 298-306, September 1994.

[KUM94c] V. Kumar, A. Grama, and N. Vempaty, “Scalable load balancing techniques for parallel computers”, Journal of Parallel and Distributed Computing, 22: pp. 60-79, 1994.

[KUN91] T. Kunz, “The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme”, IEEE Transactions on Software Engineering, vol. 17, no. 7, 725-730, July 1991.

[LEE95] Lee, B., “Dynamic Load Balancing in a Message Passing Virtual Parallel Machine”, Technical Report, Division of Computer Engineering, School of Applied Science, Nanyang Technological University, Singapore, 1995.

[LEE97] Gil-Haeng Lee, “Using system state information for adaptive state polling policy in distributed load balancing”, 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis (pAs '97), pp. 166-173, March 1997.

[LEE99] Gil-Haeng Lee, “Issues of the State Information for Location and Information Policies in Distributed Load Balancing Algorithm”, 25th Euromicro Conference (EUROMICRO '99)-Vol. 1, pp. 1067-1070, September, 1999.

[LEW98] Lewis, J.; Hart, E.; Ritchie, G.:”A comparison of dominance mechanisms and simple mutation on non-stationary problems”. Proc. Parallel Problem from Nature Conference, LNCS 1498, pp.139-148, 1998.

[LIL92] D. J. Lilja (ed.), “Architectural Alternatives for Exploiting Parallelism”, IEEE Computer Society Press, Los Alamitos, CA, ISBN 0-8186-2642-9, 1992.

[LIL93] D. J. Lilja, “Experiments with a Task Partitioning Model for Heterogeneous Computing”, Workshop on Heterogeneous Processing, International Parallel Processing Symposium, pp. 29-35, April 1993.

[LIL94a] D. J. Lilja, “A Multiprocessor Architecture Combining Fine-Grained and Coarse-Grained Parallelism Strategies”, Parallel Computing, vol. 20, no. 5, pp. 729-751, May 1994.

[LIL94b] D. J. Lilja, “Exploiting the Parallelism Available in Loops”, Computer, vol. 27, no. 2, pp. 13-26, February 1994.

[LIN87] LIN, F.; KELLER, R.:”The Gradient Model Load Balancing Method”. IEEE Trans Software Eng., Vol.1, No.1, pp.32-38. Enero, 1987.

[LIU93] J. Liu and V.A.Saletore, “Self-Scheduling on Distributed-Memory Machines”, Supercomputing ’93, pp. 814-823, 1993.

[MAH92] J. Mahdavi, G. L. Huntoon, and M. B. Mathis, “Deployment of a HiPPI-based Distributed Super computing Environment at the Pittsburgh Supercomputing Center”, Workshop on Heterogeneous Processing, International Parallel Processing Symposium,pp. 93-96, 1992.

[MAL00] Shahzad Malik, “Dynamic Load Balancing in a Network of Workstations”, 95.515F Research Report, Nov. 2000.

[MAR94] E. P. Markatos and T.J.LeBlanc, “Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors”, IEEE Transactions on Parallel and Distributed Systems,vol. 5, no. 4, pp. 379-400, April 1994.

[MAT04] http://mathworld.wolfram.com/NP-Problem.html, © 1999 CRC Press LLC, © 1999-2004 Wolfram Research, Inc.

[MEL96a] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel, “Adaptive Load Balancing of Irregular Applications”, In 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR `96), August 1996.

[MEL96b] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel, “An Adaptive Load Information Collection Policy”, In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA `96), August 1996.

[MEN90] D. Menasce and V.Almeida, “Cost-Performance Analysis of Heterogeneity in Supercomputer Archi tectures”, Proc. Supercomputing ’90,pp. 169-177, 1990.

[MEN91] D. Menasce and V.Almeida, “Heterogeneous Supercomputing: Is It Cost-Effective?”, Supercomputing Review,vol. 4, no. 8, pp. 39-41, August 1991.

[MIC00] Michalewicz, Z.; Fogel, D.B.:”How to Solve It: Modern Heuristics”. Springer, 2000.

[MIC92] Zbigniew Michalewicz, “Genetic Algorithms + Data Structures = Evolution Programs”, Second, Extended Edition, Springer-Verlag, 1992.

[MIT94] Mitchell, M., Holland, J.H., and Forrest, S., “When will a genetic algorithm outperform hill climbing”, In J. Cowan, G. Tesauro, and J. Alspector (Eds.), Advances in Neural Information Processing Systems. Morgan Kaufman, 1994.

[MOR97] Mori, N.; Imanishi, S.; Kita, H.; Nushikawa, Y.:”Adaptation to changing environments by means of the memory based thermodynamical genetic algorithm”. Conference on Genetic Algorithms, pp.299-306, 1997.

[MOY99] Jeffrey Moyer, Transparent Process Migration on the Beowulf System, Copyright © 1993, 1994, 1995, 1996, 1997, Nikos Drakos, Computer Based Learning Unit, University of Leeds. http://segfault.dhs.org/ ProcessMigration/Proposal/ Jan. 1999.

[MPI94] Message Passing Interface Forum, MPI: A Message Passing Interface standard, International Journal of Supercomputer Applications, Volume 8 (3/4), 1994.

[MUN95] MUNIZ, F.; ZALUSKA, E.:”Parallel Load-Balancing: An extension to the Gradient Model”. Parallel Computing, Vol. 21, pp.287-301, 1995.

[NAS96] J. M. Nash, P. M. Dew, J. R. Davy, and M. E. Dyer, “Scalable Dynamic Load Balancing Using a Highly Concurrent Shared Data Type”, In The 2nd European School of Computer Science: Parallel Programming Environments for High Performance Computing, pages 123--128, April 1996.

[NI85] L. M. Ni and K. Hwang, “Optimal Load Balancing in a Multiple Processor System with Many Job Classes”, IEEE Transactions on Software Engineering, SE11(5):491--496, May 1985.

[NIS93] Nishikawa, H.; Steenkiste, P., “A general architecture for load balancing in a distributed-memory environment”, Proceedings the 13th International Conference on Distributed Computing Systems, pp. 47-54, 1993.

[OZD93] Banu Ozden, Aaron J. Goldberg, and Avi Silberschatz, “Scalable and Non-Intrusive Load-Sharing In Owner-Based Distributed Systems”, In 5th IEEE Symposium on Parallel and Distributed Processing, pp. 690-699, December 1993.

[PAP94] PAPADIMITRIOU, C.:”Computational Complexity”. Addison Wesley, 1994.

[PAP94] Papadimitriou, C.:”Computational Complexity”. Addison Wesley, 1994.

[PAR95] Chulhye Park, J.G. Kuhl , “A fuzzy-based distributed load balancing algorithm for large distributed systems”, Second International Symposium on Autonomous Decentralized Systems, pp. 266-273, April 1995.

[PFI98] Pfister, G.F.:”In Search of Cluster. The Ongoing Battle in Lowly Parallel Computing”. Prentice Hall, 1998.

[POL87] C. Polychronopoulos and D. Kuck, “Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers”, IEEE Transactions on Computers,vol. C-36, no. 12, pp. 1425-1439, December 1987.

[QUI04] Michael J. Quinn, “Parallel Programming in C with MPI and OpenMP”, McGraw-Hill Companies, Inc. New York, USA 2004.

[REC73] Rechenberg, I.:”Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution”. Frommann-Holzboog, Stuttgart, 1973.

[REN96] Renders, J.-M.; Flasse, S.P.:"Hybrid Methods using Genetic Algorithms for Global Optimization". IEEE Trans. on Systems, Man, and Cybernetic, Part B, Vol.26, No.2, pp.243-258. Abril, 1996.

[RUD89] D. C. Rudolph and C. Polychronopoulos, "An Efficient Message-Passing Scheduler Based on Guided Self Scheduling", ACM International Conference on Supercomputing,pp. 50-60, 1989.

[SAL93] V.A. Saletore, J. Liu, and B. Y.Lam, "Scheduling Non-uniform Parallel Loops on Distributed Memory Machines", Hawaii International Conference on System Sciences, Vol. II,pp. 516-525, January 1993.

[SAN94] P. Sanders. "A detailed analysis of random polling dynamic load balancing", In International Symposium on Parallel Architectures, Algorithms and Networks, pp. 382-389, Kanazawa-Japan 1994.

[SAR98] Sareni, B.; Krähenbühl, L.:"Fitness sharing and niching methods revisited". IEEE Transactions on Evolutionary Computation, Vol.2, No.3, 1998.

[SCH95] Schwefel, H.-P.:"Evolution and Optimum Seeking". Wiley, New York, 1995.

[SER90] Serra, R.; Zanarini, G.:"Complex systems and cognitive processes". Springer, 1990.

[SER97] Seredynski, F.:"Competitive Coevolutionary Multi-Agent Systems: The Application to Mapping and Scheduling Problems". Lournal of Parallel and Distributed Computing, Vol.47, pp.39-57, 1997.

[SER98] Seredynski, F.:"Discovery with Genetic Algorithm Scheduling Strategies for Cellular Automata". Parallel Problem Solving from Nature PPSNV, Lecture Notes in Computer Science 1498, pp.643-652, 1998.

[SER99] Seredynski, F.; Koronacki, J.; Janikow, Z.:"Distributed Scheduling with Decomposed Optimization Criterion: Genetic Programming Approach". Proc. 11th IPPS/SPDP'99, pp.192-200, 1999.

[SHE02] Kai Shen, Tao Yang, Lingkun Chu, "Cluster Load Balancing for Fine-Grain Network Services", IEEE International Parallel and Distributed Processing Symposium , Fort Lauderdale, California, April 2002.

[SHI90] Shivaratri, N.G.; Krueger, P.:"Two adaptive location policies for global scheduling algorithms". Proc. IEEE International Conference on Distributed Computing Systems, pp.502-509, 1990.

[SHU96] W. Shu and M. Wu, "Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers", IEEE Transactions on Parallel and Distributed System, Vol. 7 no. 6, pp. 637-649, June 1996.

[SIE94] Siegell, B., Steenkiste, P. "Automatic Generation of Parallel Programs with Dynamic Load Balancing", IEEE Symposium on High Performance Distributed Computing, August 1994.

[SMA92] L. Smarr and C. E. Catlett, "Metacomputing", Communications of the ACM,vol. 35, no. 6, pp. 45-52, June 1992.

[SON94] SONG, J.:"A Partially Asynchronous and Iterative Algorithm for Distributed Load Balancing". Parallel Computing, Vol.20, pp.853-868, 1994.

[SUN90] V.S.Sunderam, "PVM: A Framework for Parallel Distributed Computing", Concurrency: Practice and Experience,vol. 2, no. 4, pp. 315-339, December 1990.

[SUN93] Xian-He Sun and Lionel Ni, "Scalable Problems and Memory-Bounded Speedup", Journal of Parallel and Distributed Computing, 19, pp.27-37, 1993.

[TAO93] L. Tao, B. Narahari, and Y.C.Zhao, "Heuristics for Mapping Parallel Computation to Heterogeneous Parallel Architectures", Workshop on Heterogeneous Processing, International Parallel Pro-cessing Symposium, pp. 36-41, 1993.

[THA01] Thyagaraj Thanalapati and Sivarama Dandamudi, "An Efficient Adaptive Scheduling Scheme for Distributed Memory Multicomputers", IEEE Trans. Parallel and Distributed Systems, Vol. 12, no. 7, pp. 758-768, July 2001.

[THE01] Theys, M.D., et al.:"What are the Top Ten most Influential Parallel and Distributed Processing concepts of the past millenium?". Journal of Parallel and Distributed Computing, Vol.61, pp.1827-1841, 2001.

[THE89] Marvin M. Theimer and Keith A. Lantz, “Finding Idle Machines in a Workstation-Based Distributed Systems”, IEEE Trans. on Software Engineering Vol. 15, No. 11, pp. 1444-1457, Nov. 1989.

[TIN00] Fernando G. Tinetti, “Performance of Scientific Processing in Networks of Workstations: Matrix Multiplication Example”, Journal of Computer Science & Technology Vol. 1 No. 4, 2000.

[TZE93] T.H.Tzen and L. M. Ni, “Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers”, IEEE Transactions on Parallel and Distributed Systems,vol. 4, no. 1, pp. 87-98, Jan-uary 1993.

[VEN02] David A. Vengerov, Hamid R. Berenji, Alexander B. Vengerov , “Adaptive Coordination Among Fuzzy Reinforcement Learning Agents Performing Distributed Dynamic Load Balancing”, In proceedings of the 11th IEEE International Conference on Fuzzy Systems, pp. 179-184, 2002.

[VET92] R. J. Vetter, D.H.C. Du, and A. E. Klietz, “Network Supercomputing: Experiments with a Cray-2 to CM-2 HiPPI Connection”, Workshop on Heterogeneous Processing, International Parallel Processing Symposium, pp. 87-92, 1992.

[WAL91] D. W. Wall, “Limits of Instruction-Level Parallelism”, International Conference on Architectural Support for Programming Languages and Operating Systems,pp. 176-188, 1991.

[WAL95] WALSHAW, C.; BERZINS, M.:”Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes”. Concurrency: Practice and Experience, Vol.7, pp.17-28, 1995.

[WAN92] M.-C. Wang, S.-D. Kim, M. A. Nichols, R. F.Freund, H. J. Siegel, and W.G.Nation, “Augmenting the Optimal Selection Theory for Superconcurrency”, Workshop on Heterogeneous Processing, International Parallel Processing Symposium, pp. 13-21, 1992.

[WAT98] Jerrell Watts and Stephen Taylor. “A Practical Approach to Dynamic Load Balancing”. IEEE Trans. on Parallel and Distributed Systems, Vol. 9, No. 3,pp. 235-247 March 1998

[WEE93] C. C. Weems, “Image Understanding: A Driving Application for Research in Heterogeneous Parallel Processing”, Workshop on Heterogeneous Processing,International Parallel Processing Symposium, pp. 119-126, 1993.

[WEI00] Weiss, R.:”Next-Generation Interconnects will drive Multiprocessing”. Electronic Design, pp.81-90. Diciembre, 2000.

[WIL05] Wilkinson, B.; Allen, M.:”Parallel Programming. Techniques and Applications using Networked workstations and Parallel Computers” (Segunda Edición). Pearson Prentice Hall, 2005.

[WIL91] WILLIAMS, R.:”Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations”. Concurrency: Practice and Experience, Vol.3, pp.457-481, 1991.

[WIL93] M. Willebeek-LeMair and A. Reeves. “Strategies for Dynamic Load Balancing on Highly Parallel Computers”, IEEE Transactions on Parallel and Distributed Systems, 4:979-993, 1993.

[WIL99] Wilkinson B., Allen M., “Parallel Programming: Techniques and Applications Using Networking Workstations”, Prentice-Hall, Inc., 1999.

[WIS03] [9] Yair Wiseman and Dror G. Feitelson, “Paired Gang Scheduling”, IEEE Trans. Parallel and Distributed Systems, Vol. 14, no. 6, pp. 581-592, June 2003.

[WIT89] L. Wittie and C. Maples, ‘‘MERLIN: Massively Parallel Heterogeneous Computing,’’ International Conference on Parallel Processing,Vol I: Architecture,pp. 142-150, 1989.

[WU91] I-C Wu and H. T. Kung. “Communication Complexity of Parallel Divide and Conquer”, In 32nd Annual IEEE Conference on Foundations of Computer Science, October 1991.

[WU96] M. Y. Wu and W. Shu. “The Direct Dimension Exchange Method for Load Balancing in k-ary n-cubes”, In Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, pp. 366-369, October 1996.

[XU90] Jian Xu and Kai Hwang, “Heuristic methods for dynamic load balancing in a message-passing supercomputer”. Proceedings of the 1990 conference on supercomputing, NY, USA, pp. 888 – 897, 1990.

[XU93] J. Xu, K. Hwang, “Heuristic Methods for Dynamic Load Balancing in a Message-Passing Multicomputer”, Journal of Parallel and Distributed Computing”, Vol. 18, no. 1, may 1993.

[XU95] C.Z. Xu and F.C.M. Lau, “The Generalised Dimension Exchange Method for Load Balancing in k-ary n-Cubes and Variants”, Journal of Parallel and Distributed Computing, 24:72--85, January 1995.

[XU97] XU, C.; LAU, F.:”Load Balancing in Parallel Computers”. Kluwer Academic Publishers, 1997.

[ZAK95] Mohammed Javeed Zaki, Wei Li and Srinivasan Parthasarathy, 1995 “Customized Dynamic Load Balancing for a Network of Workstations”, Technical Report 602.

[ZAK96] Zaki, M.; Li, W.; Parthasarathy, S.:”Customized dynamic load balancing for a Network Of Workstations”. Proc. 5th Int. Symp. on High Performance Distributed Computing, pp.282-291, 1996.

[ZAK97] Mohammed Javeed Zaki, Wei Li, Srinivasan Parthasarathy, “Customized Dynamic Load Balancing for a Network of Workstations”, J. of Parallel and Distributed Computing, Vol. 43, I. 2 , pp. 156-162, June 1997.

[ZHO88] S. Zhou. “A Trace-Driven Simulation Study of Dynamic Load Balancing”, IEEE Trans. on Software Engineering, Vol. 14, No. 9, pp. 1327-1341, Sept. 1988.

[ZIT98] Zitzler, E.; Thiele, L.:”An evolutionary algorithm for multiobjective optimization: the strength Pareto approach”. Technical Rep. No. 43, Computer Engineering and Networks Laboratiry, Zürich, 1998.

[ZNA94] ZNATI, T.F.; MELHEM, R.G.:"A Uniform Framework for Dynamic Load Balancing Strategies in Distributed Processing Systems". J. of Parallel and Distributed Computing, Vol. 23, pp.246-255, 1994.

[ZOM01] Albert Y. Zomaya, and Yee-Hwei Teh, "Observations on Using Genetic Algorithms for Dynamic Load-Balancing", IEEE Trans. Parallel and Distributed Systems, Vol. 12, no. 9, pp. 899-911, Sep. 2001.

[ZOM01b] Zomaya, A.Y.; Ercal, F.; Olariu, S.:"Solutiond to Parallel and Distributed Computing Problems: Lessons from Biological Sciences". Wiley, New York, 2001.

[ZOM99] Albert Y. Zomaya, Chris Ward and Ben Macey, "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues", IEEE Trans. Parallel and Distributed Systems, Vol. 10, no. 8, pp. 795-812, Aug. 1999.